Automated Control of Aggressive Prefetching for HTTP Streaming Video Servers

Jim Summers
University of Waterloo
jasummer@cs.uwaterloo.ca

Tim Brecht
University of Waterloo
brecht@cs.uwaterloo.ca

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

Tyler Szepesi
University of Waterloo
stszepesi@cs.uwaterloo.ca

Ben Cassell
University of Waterloo
becassel@cs.uwaterloo.ca

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

ABSTRACT

Past work has shown that disk prefetching can be an effective technique for improving the performance of disk bound workloads. However, the performance gains are highly dependent on selecting a prefetch size that is appropriate for a specific system and workload. Using a prefetch size that is too small can lead to poor overall disk throughput, whereas prefetch sizes that are too large can lead to data being evicted before it can be used by a subsequent request.

This paper looks at disk prefetch sizing for HTTP video streaming servers, such as those used by Apple, Adobe, Netflix, YouTube and Microsoft. We evaluate various representative streaming video workloads and show that the prefetch size that produces the best throughput can vary from 2 MB to 12 MB, and can depend on workload and system characteristics such as video bitrate, hard drive specifications, and memory capacity. A good choice of prefetch size can result in substantial performance gains, for example up to 3 times higher throughput than when using a prefetch size that is too large. We also find that application-level prefetching using the best prefetch size can provide up to 4 times higher throughput. In order to take full advantage of disk prefetching without extensive workload specific experimentation, we introduce an adaptive algorithm that dynamically selects an appropriate prefetch size. Most importantly, our results show our adaptive algorithm selects prefetch sizes that provide performance rivaling the best sizes determined through manual tuning, which requires extensive testing over different possible sizes.

Categories and Subject Descriptors

H.3.5 [**INFORMATION STORAGE AND RETRIEVAL**]: Online Information Services—*Web-based services*; D.4.3 [**OPERATING SYSTEMS**]: File Systems Management—*Access methods*; D.4.8 [**OPERATING SYSTEMS**]: Performance—*Measurements*

SYSTOR'14, June 10–12, 2014, Haifa, Israel.
Copyright 2014 ACM 978-1-4503-2920-0/14/06 ...$15.00.
DOI 10.1145/2611354.2611371

General Terms

Algorithms, Experimentation, Measurement, Performance

Keywords

HTTP, video streaming, web server, automation, prefetching, disk I/O

1. INTRODUCTION

The popularity of HTTP streaming video is rapidly increasing. Netflix currently accounts for 33% of peak fixed-line US traffic, YouTube represents 31% of mobile peak North American traffic, and the proportion of video traffic on the Internet in the United States is projected to rise to 68% by the end of 2018 [29]. Video workloads cannot typically fit in memory due to the large average size of video files and the tendency for video collections to have long tail popularity distributions. Furthermore, because of the vast number of videos in these video collections, it is currently not economically viable for all videos to be stored on SSDs. As a result, the majority of requests arriving at HTTP streaming video servers must be serviced from disk.

Although HTTP streaming video is largely sequential, and hard drives provide their highest throughput with sequential workloads, in previous work we found that the operating system alone may not provide sufficiently efficient disk access to service the volume of requests seen by video web servers [32, 33]. In those papers the application serializes disk requests and aggressively prefetches data using 2 MB reads. This is done above and beyond whatever the operating system is doing to improve system throughput (e.g., prefetching and/or read ahead). In this paper we examine two important questions: what is a good prefetch size for these workloads and what factors affect that choice.

As expected, our experiments show that prefetching too little or too much can cause dramatic decreases in throughput. For example, we find that appropriately-sized prefetching can perform up to three times better than overly aggressive prefetching. However, choosing this "appropriate" size typically requires considerable knowledge of system configurations and workload characteristics and a large amount of experimentation. Furthermore, this analysis must be repeated any time one of the relevant factors changes.

To simplify the task of providing high performance for HTTP video streaming, we propose the use of an automated algorithm for determining the prefetch size. We present one such algorithm that monitors and uses system measurements related to memory and disk usage and periodically adjusts the prefetch size used by the application. The goal is to improve the overall web server throughput, which increases the number of clients that can be serviced simultaneously. The contributions of this paper are:

- We examine the impact of various factors on the best prefetch size: Amount of system memory, video popularity distribution, video bitrates, and hard drive characteristics.
- We demonstrate the impact that prefetch size has on HTTP streaming video workloads. The best prefetch size provides up to 4 times higher throughput than without application-level prefetching, and up to 3 times higher throughput than a prefetch size that is too large.
- We introduce a novel analysis that shows that a prefetch size proportional to the square root of the bitrate of each video minimizes the number of disk seeks.
- We develop an algorithm that automatically determines a prefetch size conducive to high throughput based on the collection and analysis of system and application performance statistics. Our design is validated through a full implementation on a testbed system and experiments with representative streaming video workloads. Our results show that the server throughput obtained using the automated algorithm compares favourably with the best manual tuning.

Our study is conducted by modifying the application rather than the operating system for a number of reasons: (1) It is easier to modify, test and debug an application than the operating system. (2) It is easier to obtain useful application level information directly from the application rather than attempt to infer it in the operating system. (3) Changes made in the application are portable to other operating systems. We treat the operating system as a gray box and use techniques that augment what the operating system does with respect to prefetching and/or read ahead. (4) Once we better understand how to effectively prefetch HTTP streaming video workloads we can see if the techniques generalize to other applications and workloads and to then implement them in the operating system.

2. BACKGROUND AND RELATED WORK

Because video files are usually large and the popularity distribution of the videos is often such that there is a long tail of infrequently accessed content, HTTP streaming video server workloads are commonly disk-bound [17]. There are two techniques for using system memory to improve performance for disk-bound applications, both of which take advantage of the locality of reference in workloads. Caching attempts to take advantage of the temporal locality of data accesses. Prefetching attempts to take advantage of spatial locality by reading data into memory before an application needs to access it. When prefetch sizes are too small or too large, however, prefetching can have a negligible or even negative effect on throughput.

Caching can provide some benefits for a streaming video workload, as measurement studies show that video popularity has a Zipf-like distribution, where some videos are highly popular and caching these videos can significantly reduce demand on the disk [4, 13]. However, the effectiveness of caching is limited because most videos are seldom watched; studies report that over the course of a day or week, 70-80% of videos are watched only a single time [13, 37].

Prefetching, the act of reading beyond data that is requested by a user and storing it in memory for future access, is a technique for translating the access patterns of applications into a form that is more efficient for the hard drive to service. Prefetching can improve both disk and application performance in a variety of ways [30], three of which are relevant to improving throughput: minimizing CPU stalls by avoiding file system cache misses, reducing the length of disk seeks by scheduling in batches, and improving throughput by reducing the number of seeks made by the disk.

2.1 Minimizing CPU Stalls

Most of the early work on prefetching was designed to reduce application stalls due to disk access [6, 28, 18, 7]. Much of this work concentrates on reducing a single application's execution time by trying to predict what data will be requested and copying that data from disk into memory before it is accessed by the application. This work is not directly applicable to the problem we are trying to solve, as web servers are designed to cope with CPU stalls, typically by using threads [35], events [24], or a combination of the two [36, 26]. Additionally, HTTP streaming video clients employ large buffers to compensate for network delays that are much longer than disk I/O delays. In fact, as we will show, it is occasionally better for HTTP streaming video servers to accept increases in file system cache misses in exchange for higher useful disk throughput.

2.2 Reducing the Length of Disk Seeks

Another area of past research has concentrated on reordering sequences of disk requests to improve the efficiency with which the disk is used [34, 9, 25]. Through a combination of libraries and kernel changes, these studies focus on issuing fewer, larger requests, or requests that can take advantage of reading consecutive disk blocks. These papers show that disk seeks, which dominate I/O time, can be reduced in both volume and length by using intelligent request scheduling.

Similar approaches can also involve delaying disk requests. The Linux anticipatory scheduler for instance, refrains from switching between request streams for short periods of time to avoid the problem of *deceptive idleness* [15], where the disk head moves (seeks to a new location) away from a region before an application has a chance to issue its next request (which should have been serviced sequentially for better performance). Other methods issue more intelligent requests by utilizing file layout information [22], providing early reads through application direction [34], or using access history to request additional, likely to be read data [9].

2.3 Maximizing Disk Throughput

The main purpose of prefetching within the context of HTTP video streaming servers is the maximization of disk throughput. While reducing average seek distances can increase disk throughput, we believe it is more effective to focus on eliminating seeks entirely through prefetching than it is to reduce the cost of seeks through scheduling. Our work in this paper considers how much data to prefetch (the *prefetch size*) in order to maximize disk throughput, recognizing that this choice is constrained by the amount of available system memory.

Early studies in servicing video streams were based on scenarios where the server pushes data to the clients, as opposed to HTTP streaming video where clients are responsible for pulling content from the server. Many of these studies examined scheduling as a means of maximizing throughput [23, 11, 31] while using minimal system memory.

This problem has also been studied for modern systems with large amounts of memory [20, 21, 27, 25]. These studies are not directly applicable to our work, however, because they focus on different workloads. Some make the simplifying assumption that each client requests a unique video [20, 25], while others depend on specific changes to the cache algorithm in the kernel. Our workload uses a Zipf popularity distribution for video files, meaning only a few of the hottest files benefit from in-memory caching. Additionally, we provide an application-level prefetching implementation that does not modify the kernel.

2.4 Prefetching and Caching Interactions

There is a limited amount of memory available in a server, and the two competing uses for it, prefetching and file system caching, demonstrate interesting behaviours because of this constrained nature. Butt, et al. [5] observed that there are interactions between prefetching and caching that can have a significant impact on the effectiveness of memory management algorithms in the kernel.

One approach to addressing this issue is to integrate prefetching and caching strategies in the kernel memory management algorithms [6, 28]. In practice, however, kernels do not integrate prefetching and caching, but instead make prefetching decisions independent of the cache management algorithms. Because of this, methods for adapting the cache algorithms to take prefetching into account were devised [19, 16]. These techniques assume that prefetching decisions are made in a layer above the memory management subsystem, and adjust memory management to best service the output of the prefetch algorithm.

We take a different approach in this paper. We use the memory management algorithm in the kernel as-is, and adapt an application level prefetching algorithm accordingly. This approach has the benefits of avoiding kernel modifications and possibly exploiting information that may be more easily obtained in the application (e.g., the video bitrate).

3. MOTIVATION

Because many HTTP video servers contain a large amount of content that is viewed infrequently, many of those requests must be serviced from disk. Therefore, a key factor in improving HTTP video server throughput is improving the throughput obtained from disk reads. We now motivate the need for using serialized aggressive prefetching and for an algorithm that dynamically adjusts the prefetch size in order to obtain good disk throughput while also ensuring that as many clients as possible can be serviced at their desired video bitrates. We assume that each video is stored as a single file rather than in a series of files containing fixed size chunks. This mimics how YouTube and NetFlix store video files, allows for a larger range of prefetch sizes and provides servers with opportunities for higher throughput [33]. As one would expect there is a delicate balance between prefetch sizes that are large enough to support high disk throughput, and sizes that are too large resulting in adverse consequences such as eviction of useful data from memory. With this balance in mind, we also describe some of the problems and metrics that were considered when designing our automated algorithm.

Figure 1 shows results obtained while servicing workloads with requests for standard definition (SD) and high-definition (HD) video files. The details of the experimental methodology are given in Section 6. The individual bars represent the throughput obtained using a vanilla web server (labeled “V”) and using web servers that have been modified to perform their own prefetching (labeled 2, 4, 6, 8, 10 and 12 to denote the prefetch size used in MB). The throughput obtained from the disk (Disk Tput) and the actual throughput observed by all clients (Actual Tput) are also shown.

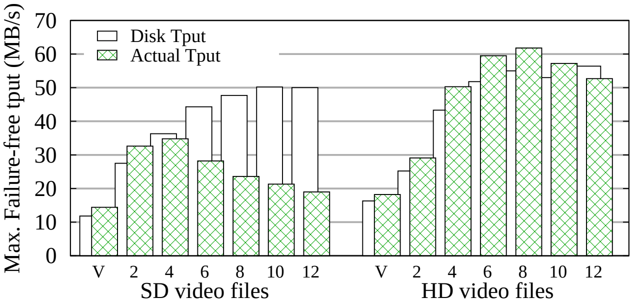

Figure 1: Throughput versus prefetch size.

These results reveal that there is a significant performance difference between the vanilla web server and the modified web server using the best prefetch size, with a factor of 2.5 improvement with SD videos (prefetch size 4 MB), and a factor of 3.4 improvement with HD videos (prefetch size 8 MB). They also show that the 2 MB prefetch size used in our previous work [32, 33] may have been significantly undersized for HD video. Additionally, these results demonstrate that although larger prefetch sizes do increase disk throughput, at some point the benefits become minimal or non-existent. Furthermore, the increase in disk throughput does not necessarily translate into improved client throughput (Actual Tput). As prefetch sizes grow too large, memory pressure results in reduced server throughput. As the figure illustrates, however, what is too large for one workload and system configuration (e.g., 8 MB for SD videos in this figure), may be the best prefetch size for another, motivating the need for an automated tuning algorithm.

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

As seen in Figure 1, for SD videos prefetch sizes larger than 4 MB reduce total server throughput. Table 1 presents some statistics, gathered during the execution of the SD video experiments, using columns with the following meanings: *Size*: prefetch size; *Disk/Requested*: ratio of the total bytes read from disk versus the total bytes requested (values less than one indicate cache hits and values greater than one indicate that re-reads were required); *Cached/Requested*: ratio of bytes read from the file system cache versus bytes requested[1]; *Evicted/Requested*: ratio of bytes evicted versus bytes requested; *Wasted/Requested*: ratio of bytes wasted versus bytes requested. The amount of wasted prefetch bytes may be slightly inaccurate when the same file is requested by more than one viewer, as it is difficult in this case to determine if it was truly a wasted prefetch.

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

Table 1: Extra data read due to prefetching.

As seen in Table 1, once prefetch sizes are too large (in this case greater than 4 MB) the system does a lot of "extra work" to service requests. Significantly more data must be read than is being requested because it must either be re-read due to file cache or prefetch evictions or because although a viewer stops watching a video the system has already prefetched data beyond the point at which the viewer stopped. Although, cache evictions, prefetch evictions, and wasted prefetches serve as the limiting factors to the performance of increasingly aggressive prefetching, in this experiment the prefetch evictions (Evicted/Requested) cause the most harm. For example, with a prefetch size of 6 MB the total number of bytes that have been prefetched and evicted (and therefore need to be re-read from disk) is greater than half of the total number of bytes requested. As a result, our automated algorithm tries to avoid excessive prefetch evictions and the results shown in our evaluations in Section 7 include information about prefetch eviction rates.

4. AUTOMATIC PREFETCH SIZING

Our automated algorithm relies on an underlying prefetcher that performs *serialized aggressive prefetching*, in which the prefetcher reads more data than was requested, and requests are serviced one at a time by a separate thread. Prefetches for all files of the same bitrate (the handling of different bitrates is described in Section 5) are done using the same prefetch size p except at the end of a file, where only the remainder of the file is read. The role of the automated sizing algorithm is to monitor the state of the server and use this feedback to periodically adjust p to improve the throughput of the system.

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

4.1 Algorithm for Adjusting Prefetch Size

While the web server is running, we continually calculate a score S, which represents the amount of work done, and use a gradient descent algorithm to find the prefetch size that minimizes S. Past work suggests using file cache misses to represent the effort required by the server [6, 12, 3]. However, our objective is to maximize throughput to the client, which involves both file cache misses and disk transfer times. Minimizing cache misses irrespective of disk transfer times can result in poor overall throughput. Therefore, we define our score as the product of both the time to read from disk and the file cache miss ratio.

Figure 2 illustrates the trade-off between cache misses and disk throughput for different choices of prefetch size, with a prefetch size of "0" representing the results of using the vanilla version of the web server (i.e., without application-level prefetching) and relying on the operating system (FreeBSD) mechanisms for obtaining good throughput. The left y-axis is used to show the number of milliseconds per transaction (*mspt*), which measures the time to read from disk, and the right y-axis is used to show the *file cache miss ratio*, which measures the ratio of data read from disk (note that ratios greater than 1 are possible when data must be read into memory multiple times due to evictions). From this figure we can see that for some prefetch sizes, accepting a small increase in the cache miss ratio can greatly reduce the disk transfer time, and therefore improve overall throughput to the clients.

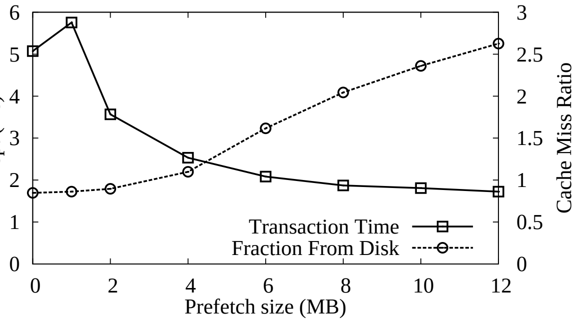

Figure 2: Transaction times versus prefetch size.

Our automated algorithm starts with an initial prefetch size and continually measures and tries to minimize the score while the server is running. At regular intervals, the algorithm compares the score at the current prefetch size (p) to the scores that were previously measured for the next larger (p_l) and smaller (p_s) prefetch sizes. If either p_l or p_s has a lower score than p, the prefetch size is adjusted towards that size. If the score has not yet been measured for p_l and the measured score for p_s is higher than the score for p, we try the unmeasured prefetch size, p_l (and vice versa if p_s has not been measured). If neither p_l nor p_s have been measured we try p_l. As will be discussed in Section 4.2, the prefetch size must be changed gradually. Therefore, we slowly change p

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

until it is equal to the desired prefetch size.

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

While the gradient descent method is suitable under most conditions, it is not effective when the system is under either very light or heavy memory pressure. In the case that the server is under light memory pressure, increasing the prefetch size will improve the disk transfer time, but it may not cause a corresponding decrease in the cache miss ratio. The result is an oversized p that will cause the server to perform very poorly when the load increases. Therefore, we turn off the automated algorithm when the load is deemed too light. Notice that there is no harm in turning off the automated algorithm because the server is under loaded and is therefore able to successfully service all of its clients.

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

This algorithm is controlled with the following parameters (actual values used while conducting our experimental evaluation are provided in Section 7):

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

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

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

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

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

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

4.2 Slowly Adjusting Prefetch Size

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

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

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

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

By adapting slowly, the automated algorithm avoids server overload conditions that could otherwise occur. Despite the fact that it changes gradually, however, it remains effective at converging towards effective prefetch rates, as will be seen in Section 7.

4.3 Prefetch Algorithm in Action

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

Figure 3 shows the operation of the algorithm during the execution of an experiment using HD video files. The figure shows the throughput of data delivered to the client, the value of the score, and the prefetch sizes chosen by the algorithm over time. The throughput is plotted using the right y-axis, while both the value of the score and the prefetch size are plotted using the left y-axis (which, for clarity, has only a single axis labelling, for prefetch size). The request rate builds over the experiment, starting at 80% of the maximum load and increasing in 5% steps over 2700 seconds. It then remains at the maximum load for 1200 seconds.

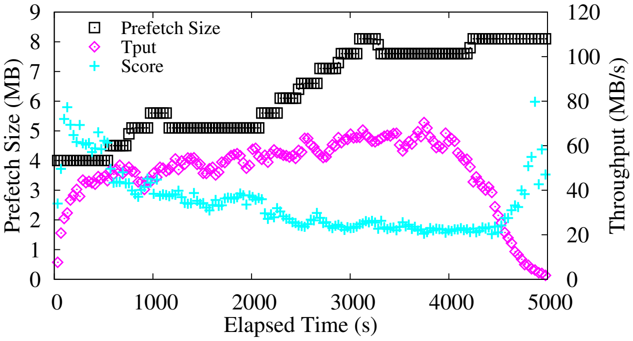

Figure 3: Dynamic prefetch size adjustments using HD videos.

The algorithm starts with a prefetch size of 4 MB, which is smaller than the best prefetch size for this workload. The prefetch size remains at 4 MB until the system experiences enough load, which begins to occur at around the 500 second mark. The algorithm increases the prefetch size in response to lower scores until the algorithm reaches a local minimum at about 1000 seconds. The stored scores are gradually deflated until the algorithm retries the higher prefetch size and measures a lower score. The algorithm continues to increase the prefetch size until it reaches 7.5 MB.

Figure 4 shows an example of an experiment where prefetch evictions are used to more quickly adjust the prefetch size. This experiment uses SD video files and starts with a prefetch size of 8 MB, a size that worked well with the HD videos in the previous experiment, but is too large for this workload. The y-axis on the right is used to show workload throughput in MB/s as well as the average prefetch eviction times in seconds. After about 500 seconds have elapsed with the increasing workload, the large prefetches begin to cause prefetch evictions. The algorithm uses the average prefetch eviction time of about 60 seconds and the average bitrate of 0.05 MB/s to calculate a new prefetch size estimate of 3 MB. After the prefetch size reaches 3 MB, the prefetch size is close to the best value, and the gradient descent algorithm reacts to the rise in workload throughput as the experiment continues.

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

5. MIXED BITRATES

For most, but not all, experiments in this paper we consider environments in which only standard definition (SD) bitrate videos are being requested. This is because YouTube workload characterization studies found that although different bitrates were available, the large majority of requests were for SD videos [13, 10] and because to our knowledge there do not exist workload characterization studies that can be used to construct representative benchmarks that include mixed bitrates. We do examine the impact of larger files by conducting some experiments with requests for only HD videos.

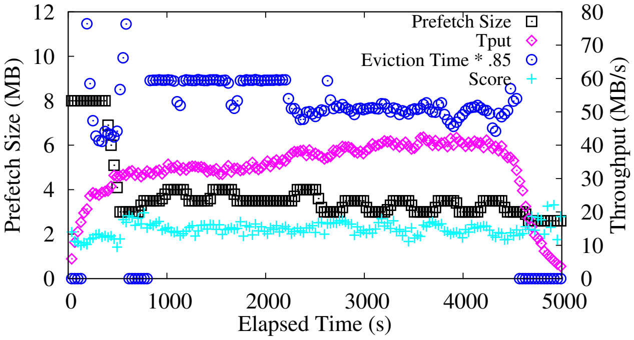

Figure 4: Dynamic prefetch size adjustments using SD videos.

However, as shown in Figure 1, the bitrate of videos can affect the choice of prefetch size. Additionally, for some workloads, different clients request videos of different bitrates, so an interesting question is what prefetch size should be used for each bitrate. Some researchers have proposed methods for handling mixed-rate workloads [12, 28], but these methods require feedback from customized kernel memory management algorithms, so they cannot be implemented in the web server.

We present a novel analysis for computing the prefetch size to use for each bitrate by minimizing the number of disk seeks. Minimizing the number of seeks allows more time for reading data, thereby increasing throughput. The number of seeks is not the only factor that affects disk throughput, but this simplified analysis is effective at improving throughput.

Let b_i be the bitrate of video i, and p_i denote the prefetch size used for that video. Let V be the average time a client spends viewing a video, which is assumed to be independent of bitrate. The average number of seeks required to read the video is then Vb_i/p_i. There is a limit to the amount of memory that can be used for prefetching, which is a fraction f of the total amount of system memory M. So the problem of determining the prefetch size for a collection of videos with different bitrates can be expressed as:

$$\text{minimize} \sum_i \frac{Vb_i}{p_i} \text{ subject to } \sum_i p_i = fM$$

where the sum is taken over the videos concurrently streaming from disk. This equation can be solved using the method of Lagrange multipliers:

$$\text{minimize } \Lambda(p_i, \lambda) = \sum_i \frac{Vb_i}{p_i} + \lambda(\sum_i p_i - fM)$$

The partial derivatives of this function are zero when:

$$\forall i, \frac{Vb_i}{p_i^2} = \lambda \text{ or } p_i = \sqrt{\frac{Vb_i}{\lambda}}$$

From this analysis, we can conclude that the number of seeks is minimized when the prefetch sizes are proportional to the square root of the video bitrates. To implement a different prefetch size for each bitrate, we specify the prefetch size for the highest bitrate video and proportionally scale down the prefetch size for lower bitrate videos.

This analysis is at odds with the work by Gill and Bathen [12], who show that the optimal prefetch size is proportional to the bitrate, rather than the square root. The goal of their analysis is essentially to optimize the cache hit rate, whereas our goal is to minimize the number of seeks. In Section 7.4 we examine workloads with different bitrates and conduct experiments to compare these two alternatives, as well as other possibilities. In all of our experiments the same prefetch size is used for all videos of the same bitrate.

6. EXPERIMENTAL METHODOLOGY

6.1 Server and Client Configurations

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. The server machine is an HP DL380 G5s containing two four-core Intel E5400 2.8 GHz processors and 32 GB of RAM. We configure the kernel to enable different amounts of physical RAM. Experiments use 4 GB of RAM unless otherwise specified. The main disk drive used for the experiments is a 1.0 TB Western Digital Red drive (model WD10EFRX). We also use a 1.0 TB Seagate drive for comparison (model ST 1000DM003). The operating system uses a separate disk from that for the video data. Bugs in the Linux implementation of `sendfile` [14] led to our use of FreeBSD 9.1-RELEASE on the server.

We use a modified `userver` web server, which implements prefetching at the application layer. This is accomplished using FreeBSD's `sendfile` flag `SF_NODISKIO`, which causes `sendfile` to return `EBUSY` if the data needs to be read from disk. When `sendfile` returns `EBUSY` we forward a prefetch request to a single worker thread and allow the main processing loop to continue with other requests. The worker thread simply issues a `read` system call to read data from disk into a single buffer using the specified prefetch size. This has the side effect of placing the prefetched data in the file system cache. After the helper thread completes the prefetch, it notifies the main processing loop which retries the `sendfile` call. At this point, it fills the socket buffer with data from the file cache and sends the data. We refer to this server architecture as asynchronous, serialized, aggressive prefetching (ASAP). Serializing requests to the disk is critical for obtaining high disk throughput by avoiding additional seeks due to request decomposition and interleaving that can otherwise occur with large, concurrent reads. Prefetching at the application level allows us to monitor data that is evicted before it is used (*prefetch eviction*), the time between prefetching and eviction (*eviction time*), and data that is prefetched but never requested (*wasted prefetches*). These and other metrics are used in our automated prefetching algorithm, described in Section 4.

We use 10 machines to generate client requests. All clients are connected to the server via multiple 1 Gbps network links through multiple 24-port switches to ensure that the network is not a bottleneck. Each system contains either dual 2.4 or dual 2.8 GHz Xeon processors and 3 GB of memory. The dual processors and memory are required to support the workload generator (`httperf`) and to emulate different types of networks used to access video servers using `dummynet`. Using `dummynet` is critical to obtaining practical results [32]. Each instance of `httperf` can potentially generate hundreds of concurrent sessions requesting streams of individual videos.

6.2 Workload

The workloads and benchmarks used in this paper are based on methodologies developed previously [32] to represent YouTube video and client characteristics that were measured in 2011 [10]. Videos have a Zipf popularity distribution with an alpha value of 0.8 being used for all experiments except where otherwise noted.

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. 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]. For our experiments we consider fixed encoding bit rates of 419 Kbps (a common bit rate observed for YouTube), and 2095 Kbps (sufficient to support higher definition video). We call these SD and HD videos, respectively. For these bitrates, ten seconds of video is equal to 0.5 MB and 2.5 MB of data, respectively. Video data is stored consecutively in one file per video and clients issue HTTP range requests for the portion of the video that will be viewed next. As we have shown in previous work [33], storing videos in a single file rather than dividing the video into chunks and storing them in separate files is essential for providing opportunities for high server throughput.

Each disk contains 20,000 SD files and 8,000 HD files, evenly distributed over the entire disk. The average length of a video is 267 seconds. Therefore, SD and HD files have an average size of 13 MB and 66 MB, respectively. The average duration spent watching a video is 162 seconds. For the SD experiments, the clients request 92 GB worth of data in 12,000 viewing sessions with the number of concurrent sessions peaking at 650. 69% of videos are only requested once and the average number of views per video is 2.1. For the HD experiments, the clients request 148 GB worth of data in 4,000 sessions with a peak of 250 concurrent sessions. 72% of videos are requested only once, and the average number of views per video is 1.9.

The primary quality criterion for video clients is the rebuffering rate of videos. We use a simple timeout as a proxy for rebuffering: `httperf` ends a connection and counts it as a failure whenever the response time exceeds 10 seconds (the time required to refill a client's buffer before it is drained). For each experiment, we determine the highest request rate that can be serviced without any client timeouts (failures). This is the *maximum failure-free rate*.

7. EXPERIMENTAL EVALUATION

In the following sections, we present results from a suite of experiments that test the effectiveness of the automated algorithm. For each experiment, we determined the maximum failure-free throughput the `userver` could achieve for a range of fixed prefetch sizes (representing a manual tuning), and compare those results to the automated algorithm, as well as the vanilla web server (labeled "V"). In all cases, we present the throughput of the `userver` (Actual Tput), the hard disk (Disk Tput), and prefetch evictions (Evictions Tput).

For the automated algorithm, in addition to the parameters shown in Table 2, we also choose a starting prefetch size, which we determined based on work done by Li et al. [20]. Their analysis shows that the prefetch size should be set so that the time required to transfer the data from disk is equal to the average seek time. Based on that analysis we experimentally determined that the WD Red drive should use a starting prefetch size of 3 and 4 MB for the SD and HD workloads, respectively. Similarly, for the Seagate drive, we use a starting prefetch size of 2 and 3 MB for the SD and HD workloads, respectively. While other starting sizes are possible, these were chosen using previously established best practices and because good estimates for the starting size help to control the duration of experiments (recall that adjustments in prefetch sizes must be done gradually over time). Note that in many cases these starting sizes are not good choices and that despite having to slowly make prefetch size adjustments, our algorithm does converge on sizes that are appropriate for the system and workload.

Parameter	Value
adjust_interval	180 seconds
step_size	0.5 MB
score_deflation_factor	5%
adjust_busy	60%
adjust_evict	5%

Table 2: Automated Algorithm Parameters.

7.1 Effect of System Memory

We performed experiments with three different amounts of system memory by changing the *hw.physmem* kernel parameter for 2 GB, 4 GB and 8 GB of physical memory. The SD workload, as shown in Figure 5, is quite sensitive to the amount of system memory. Disk throughput increases with prefetch size regardless of the amount of system memory, but if there is too little system memory available to store the prefetched data, the extra disk throughput results in evictions rather than actual throughput. The throughput when using the automated algorithm (labeled “A”) is within 5% of the throughput when using the best fixed prefetch size, demonstrating that the performance with the automated algorithm is comparable to that of hand tuning.

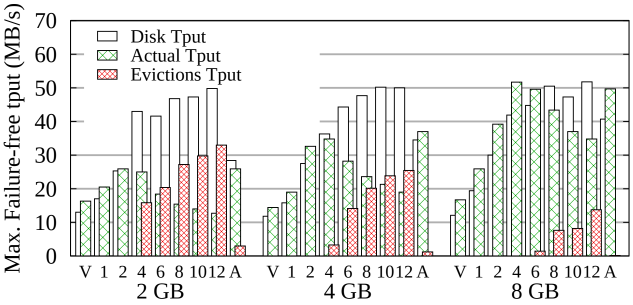

Figure 5: Comparing prefetching techniques for SD video while varying system memory.

These results also provide insight into reasons why increasing system memory improves throughput. For example, with a 2 MB prefetch size, increasing system memory from 2 GB to 8 GB increases the amount of memory available for caching, and results in a 54% increase in throughput. Furthermore, throughput increases by an additional 35%, when we increase the prefetch size from 2 MB to 4 MB.

We repeated the experiments using an HD video workload and obtained the results shown in Figure 6. Compared to the SD video workload, the eviction throughput is lower for all system memory configurations when using the same prefetch size. This is because the bitrate of HD video is 5 times higher than SD video, so only 1/5 as many HD clients can be serviced for a given throughput. Less system memory is required to store prefetched data for the fewer (HD) clients, because the amount of memory needed to store prefetched data is equal to the number of concurrent clients multiplied by the prefetch size. As a result, there are fewer evictions for a given prefetch size. Additionally, the improvements in throughput with larger system memory sizes are mainly due to increasing the number of cache hits. Comparing the best prefetch size of 6 MB using 2 GB of system memory to the best prefetch size of 12 MB using 8 GB of system memory, there was a 28% improvement in throughput from improved caching and an additional 15% improvement due to the increase in prefetch size.

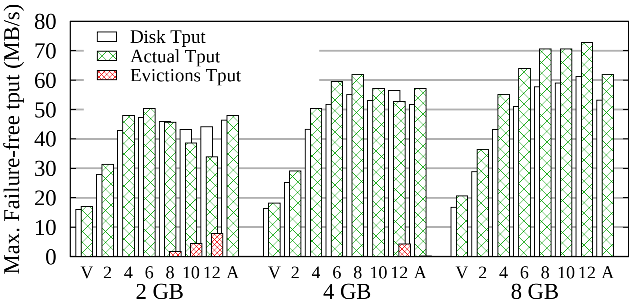

Figure 6: Comparing prefetching techniques for HD video while varying system memory.

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

7.2 Effect of Popularity Distribution

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

Although prefetching is not directly affected by the popularity distribution, caching and prefetching compete for system memory. In order to determine if the best prefetch size is sensitive to the popularity distribution, we generated two additional workloads: one with $\alpha = 0.6$ and the other with $\alpha = 1$. These are compared with the standard workload that uses $\alpha = 0.8$. All three workloads use the same SD video file set, and files have the same popularity rank in all the distributions. This ensures that, to the extent possible, the same videos are requested in each workload.

Figure 7 shows the maximum failure-free rates that could be achieved across the different workloads, using 4 GB of system memory. These results demonstrate that, with a larger α parameter, the actual throughput is improved across all prefetch sizes. In contrast, across the range of values for α, there is little or no change in disk and eviction throughput for a given prefetch size. For these experiments, the workload distribution has little impact on the amount of system memory available for prefetching, so the best throughput is achieved using the same 4 MB prefetch size, regardless of the α parameter. Instead, the increase in actual throughput is caused by an increased cache hit rate.

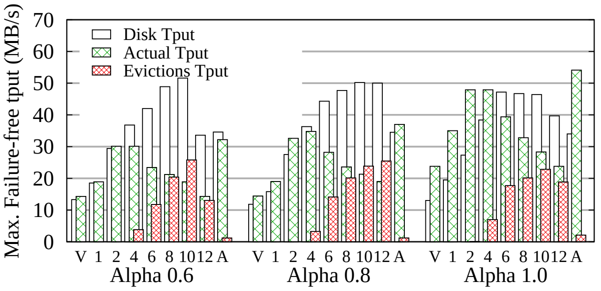

Figure 7: Different popularity distributions (α), with SD videos and 4 GB of memory.

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

7.3 Effect of Hard Drive Characteristics

The experiments presented so far use a 1.0 TB 5,400 RPM Western Digital Red drive, chosen because it is advertised to be energy efficient while still providing high throughput. As a point of comparison, we repeat the SD and HD video experiments of Section 3 using a 1.0 TB 7,200 RPM Seagate drive with lower seek latencies. The drives have the same capacity and we carefully populate the drives with files of the same size in the same locations to ensure results obtained using these drives can be directly compared [33].

The differences in the speeds of these disks are reflected in Figure 8, which shows the results of prefetch size experiments using 4 GB of system memory. The results for the Red drive were previously shown in Figure 1 and are included here for convenience.

Due to the higher transfer rate and shorter seek times, the Seagate drive is able to achieve higher throughput when using small prefetch sizes. In addition, the higher throughput of the Seagate drive allows the `userver` to support about twice as many concurrent clients. The tradeoff is that, by doubling the number of concurrent clients, the memory required to store all of the prefetched data is also doubled. The increased memory pressure causes more evictions when using the Seagate drive. The differences between the drives are smaller when using the HD workload, shown in Figure 9. When servicing HD video, there are fewer concurrent clients, which reduces memory pressure and the eviction rates when compared with the SD video case.

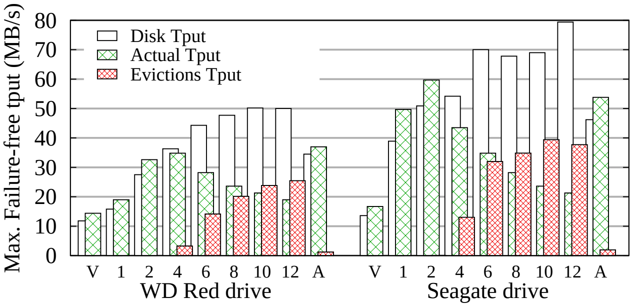

Figure 8: Comparing prefetching techniques on different disks with SD videos and 4 GB memory.

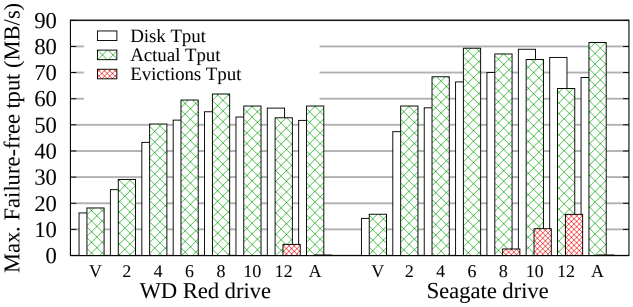

Figure 9: Comparing prefetching techniques on different disks with HD videos and 4 GB memory.

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

7.4 Effect of Mixed-Bitrate Workloads

The workloads in the experiments prior to this point have used a single bitrate for all videos. In Section 5, we devised a method for handling workloads with mixed bitrates, and we now test our method using a workload that contains 50% HD videos and 50% SD videos.

Some of the results of experiments we have conducted with this mixed workload are shown in Figure 10. The labels just below the x-axis (.1, .2, .45, .75 and 1) specify a scaling factor that is used to determine the prefetch size of the SD videos when compared with the size used to prefetch HD videos. The other labels on the x-axis (6 MB, A, and V) show the prefetch size used for the HD videos. "A" denotes the automated algorithm and "V" identifies the vanilla server.

The scaling factors of key interest are 1 and 0.45. A factor of 1 means that the same prefetch size is used for SD videos as HD videos. A factor of 0.45 is calculated using our seek-minimizing rule: the prefetch size should be proportional to the square root of the bitrate. Since HD videos have a 5 times higher bitrate than SD videos, by our analysis, the scale factor for SD videos should be $\sqrt{1/5} = 0.45$.

Therefore, when the HD video prefetch size is 6 MB the SD prefetch size is 2.7 MB. We include a scaling factor of 0.2 because previous work has determined that it is optimal to prefetch an amount proportional to the bitrate [12] and the SD bitrate is 20% of the HD bitrate. In addition, 0.1 and 0.75 are included to examine the sensitivity of the results to the scaling factor. We show only the results for a 6 MB HD prefetch size because that is the prefetch size that obtains the best throughput. Note that the best prefetch size for this workload is close to that for the purely HD workload. This is not surprising because the throughput required to service the HD requests dominates that for the SD requests; although there are equal numbers of clients requesting SD and HD videos, HD videos account for 5/6 of the total throughput.

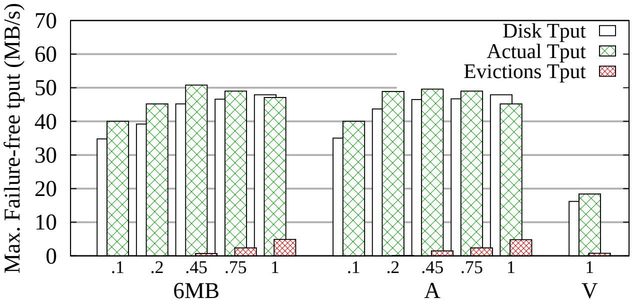

Figure 10: Varying scaling factors and prefetching technique while using 50% SD videos and 50% HD videos on WD Red drive.

We observe that the the differences in throughput are fairly small across the different scaling factors, for this workload and system configuration. However, the throughput obtained with the scaling factor of 0.45 is as good or better than that achieved with the other scaling factors. In future work it would be interesting to determine if there are circumstances under which the differences are larger. Finally, we note that the automated algorithm is also comparable to the best fixed prefetch sizes on this workload.

8. DISCUSSION

Our results show the importance of having sufficient server memory to support a large prefetch size. For example, with the scenarios considered in Figure 5, increasing the server memory from 2 to 8 GB enables a doubling of the throughput. Although these results are for a single disk system, appropriate scaling by the number of disks can yield insight applicable to the common deployment scenario in which an HTTP streaming video server is configured with many disks.

It might be possible to reduce the amount of system memory required by improving the memory management algorithms and reducing prefetch evictions. For our experiments, we use the FreeBSD memory management algorithms as-is, but other studies have shown there are benefits from treating prefetched memory specially [19, 3, 12] and by using video-specific caching algorithms such as interval caching [8]. In future work, we plan to investigate these techniques, which can be applied in concert with our automated algorithm.

Another possible area of future work concerns stream-specific adjustments to prefetch size. Our results show the potential benefit of using a prefetch size scaling factor based on the bitrate of the video being streamed. Considering additional characteristics might yield further benefits. For example, using a smaller prefetch size for new streams could be beneficial when there is a relatively high rate of termination by the user early in the video playback, as is often observed in practice [10, 1]. One might also take into account the video or user identity, for example prior work has observed that some users are “serial” early-quitters [1]. Finally, our workloads have not included use of HTTP adaptive streaming, wherein clients can adaptively switch among different versions of a video with different bitrates, since the data on video and client characteristics [10] that we use for our workloads was measured for a system not employing this technique. In HTTP adaptive streaming systems, the prefetch size for a stream might be adjusted based on the estimated likelihood that the client will soon be switching to a different version.

9. CONCLUSIONS

HTTP-based video streaming is increasingly common, and it is important that web servers be able to efficiently support this type of service. Large file sizes and content popularity characteristics that often entail a long tail of lukewarm or cold content result in HTTP streaming video servers frequently being disk-bound [17]. It is important that disks in such systems be used efficiently, and large sequential reads on video file data accomplish this.

In this paper we addressed the issue of selecting the prefetch size to use with aggressive application-level prefetching. We showed that performance is quite sensitive to this factor, with the best prefetch size providing up to 4 times higher throughput than without application-level prefetching, and up to 3 times higher throughput than with a prefetch size that is too large. Our experiments also demonstrated that the best choice of prefetch size varies considerably depending on the system characteristics as well as the video bitrates. For this latter factor, we provided an analysis that suggests that with a workload mix including videos of different bitrates, the prefetch size for each video should be chosen proportional to the square root of the video bitrate.

We devised an automated algorithm that considers both memory and disk statistics to choose prefetch sizes. We demonstrated the effectiveness of this algorithm and showed that it could successfully determine appropriate prefetch sizes for different workloads or system configurations. Aggressive application-level prefetching, using our automated algorithm for prefetch sizing, enables greatly improved performance without the need for manual tuning.

10. ACKNOWLEDGMENTS

Brecht, Eager and Wong would like to thank the Natural Sciences and Engineering Research Council (NSERC) of Canada for partial support for this project through Discovery Grants. Brecht has also received an NSERC Discovery Accelerator Supplement in support of this work. Szepesi and Cassell received partial support through NSERC, University of Waterloo President and Go Bell graduate scholarships. Summers is partially supported by a University of Waterloo Cheriton Scholarship. The authors would also like to thank the reviewers for their comments.

References

[1] A. Balachandran, V. Sekar, A. Akella, and S. Seshan. Analyzing the potential benefits of CDN augmentation strategies for Internet video workloads. In *Proc. ACM IMC*, 2013.

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

[4] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *Proc. IEEE INFOCOM 1999*, March 1999.

[5] A. R. Butt, C. Gniady, and Y. C. Hu. The performance impact of kernel prefetching on buffer cache replacement algorithms. In *Proc. ACM SIGMETRICS*, 2005.

[6] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Trans. Comput. Syst.*, 14(4):311–343, 1996.

[7] F. Chang and G. Gibson. Automatic I/O hint generation through speculative execution. 1999.

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

[9] X. Ding, S. Jiang, F. Chen, K. Davis, and X. Zhang. Diskseen: exploiting disk layout and access history to enhance I/O prefetch. In *Proc. USENIX ATC*, 2007.

[10] 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.

[11] D. Ghose and H. J. Kim. Scheduling video streams in video-on-demand systems: A survey. *Multimedia Tools Appl.*, 2000.

[12] B. S. Gill and L. A. D. Bathen. Optimal multistream sequential prefetching in a shared cache. *Trans. Storage*, 3(3), 2007.

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

[14] 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.

[15] S. Iyer and P. Druschel. Anticipatory scheduling: a disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *Proc. SOSP*, 2001.

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

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

[18] T. Kimbrel, A. Tomkins, R. Hugo, P. Brian, and B. P. Cao. A trace-driven comparison of algorithms for parallel prefetching and caching. In *Proc. OSDI*, 1996.

[19] C. Li and K. Shen. Managing prefetch memory for data-intensive online servers. In *Proc. USENIX FAST*, 2005.

[20] C. Li, K. Shen, and A. E. Papathanasiou. Competitive prefetching for concurrent sequential I/O. In *Proc. EuroSys '07*, 2007.

[21] S. Liang, S. Jiang, and X. Zhang. STEP: Sequentiality and thrashing detection based prefetching to improve performance of networked storage servers. In *Proc. ICDCS '07*, 2007.

[22] C. H. Lunde, H. Espeland, H. K. Stensland, and P. Halvorsen. Improving file tree traversal performance by scheduling I/O operations in user space. In *Proc. IEEE IPCCC '09*, 2009.

[23] B. Ozden, R. Rastogi, and A. Silberschatz. On the design of a low-cost video-on-demand storage system. *MULTIMEDIA SYSTEMS*, 1996.

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

[25] G. Panagiotakis, M. D. Flouris, and A. Bilas. Reducing disk I/O performance sensitivity for large numbers of sequential streams. In *Proc. - International Conference on Distributed Computing Systems*, 2009.

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

[27] C. Patrick, N. Voshell, and M. Kandemir. App: Minimizing interference using aggressive pipelined prefetching in multi-level buffer caches. In *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*, 2011.

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

[29] Sandvine Inc. Global internet phenomena report – 2H 2012.

[30] E. A. Shriver, C. Small, and K. A. Smith. Why does file system prefetching work? In *Proc. USENIX ATC*, 1999.

[31] R. Steinmetz. Multimedia file systems survey: Approaches for continuous media disk scheduling. *Comput. Commun.*, 18(3), 1995.

[32] J. Summers, T. Brecht, D. Eager, and B. Wong. Methodologies for generating HTTP streaming video workloads to evaluate web server performance. In *Proc. SYSTOR*, 2012.

[33] 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.

[34] S. VanDeBogart, C. Frost, and E. Kohler. Reducing seek overhead with application-directed prefetching. In *Proc. USENIX ATC*, 2009.

[35] R. von Behren, J. Condit, F. Zhou, G. Necula, and E. Brewer. Capriccio: Scalable threads for internet services. In *Proc. ACM SOSP*, 2003.

[36] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proc. ACM SOSP*, 2001.

[37] 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.