To appear, ACM Transactions on Modeling and Performance
Evaluation of Computing Systems (TOMPECS)

Disk Prefetching Mechanisms for Increasing HTTP Streaming Video Server Throughput

BENJAMIN CASSELL, University of Waterloo
TYLER SZEPESI, University of Waterloo
JIM SUMMERS, University of Waterloo
TIM BRECHT, University of Waterloo
DEREK EAGER, University of Saskatchewan
BERNARD WONG, University of Waterloo

ACM Reference format:
Benjamin Cassell, Tyler Szepesi, Jim Summers, Tim Brecht, Derek Eager, and Bernard Wong . 2018. Disk Prefetching Mechanisms for Increasing HTTP Streaming Video Server Throughput. 1, 1, Article 1 (March 2018), 31 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ABSTRACT

Most video streaming traffic is delivered over HTTP using standard web servers. While traditional web server workloads consist of requests that are primarily for small files that can be serviced from the file system cache, HTTP video streaming workloads often service a long tail of large infrequently requested videos. As a result, optimizing disk accesses is critical to obtaining good server throughput.

In this paper we explore serialized, aggressive disk prefetching, a technique which can be used to improve the throughput of HTTP streaming video web servers. We identify how serialization and aggressive prefetching affect performance and, based on our findings, we construct and evaluate Libception, an application-level shim library that implements both techniques. By dynamically linking against Libception at runtime, applications are able to transparently obtain benefits from serialization and aggressive prefetching without needing to change their source code. In contrast to other approaches that modify applications, make kernel changes, or attempt to optimize kernel tuning, Libception provides a portable and relatively simple system in which techniques for optimizing I/O in HTTP video streaming servers can be implemented and evaluated.

We empirically evaluate the efficacy of serialization and aggressive prefetching both with and without Libception, using three web servers (Apache, nginx and the userver) running on two operating systems (FreeBSD and Linux). We find that, by using Libception, we can improve streaming

XXXX-XXXX/2018/3-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

throughput for all three web servers by at least a factor of 2 on FreeBSD and a factor of 2.5 on Linux. Additionally, we find that with significant tuning of Linux kernel parameters, we can achieve similar performance to Libception by globally modifying Linux's disk prefetch behaviour. Finally, we demonstrate Libception's ability to reduce the completion time of a microbenchmark involving two applications competing for disk resources.

1 INTRODUCTION

Video streaming over HTTP is now the largest contributor to Internet traffic. The catalogue of available content from popular video streaming services has also been growing rapidly, and although memory caching and SSD can be effective for the most popular content, HTTP streaming video server workloads are often disk-bound [13]. Techniques for improving disk throughput in such systems are therefore of considerable interest.

There has been much past work on improving disk access efficiency. In the context of HTTP video streaming, however, there are two complicating factors. First, unlike early video streaming systems, HTTP-based video streaming is pull-based: the server responds to client requests for video chunks, rather than pushing video data to the client at some server-determined rate. Second, contemporary servers may be highly concurrent, responding to video chunk requests from hundreds or thousands of clients concurrently.

A well-known approach to making disk access more efficient for applications that access files sequentially is to perform larger reads from disk to prefetch data before it has been requested. In prior work we observed that *serializing* reads may also be important in some contexts [35, 36]. By modifying a web server to both aggressively prefetch *and* serialize its reads, we were able to significantly improve performance on FreeBSD. This prior work did not investigate whether the approach of combining aggressive prefetching with serialization could yield similarly large benefits for other web servers or on other operating systems.

Implementing techniques for improving disk access efficiency inside the application requires detailed knowledge of the application code, and must be repeated for each application of interest. This could be quite difficult for applications with large code bases such as Apache and nginx. On the other hand, a kernel implementation is operating system specific, requires detailed knowledge of the relevant pieces of kernel code, and has the additional problem of potential adverse impacts on other types of applications.

In this paper, we address the problem of improving disk access efficiency in HTTP video streaming servers by further exploring the impact of aggressive prefetching and serialization on web servers. Using our findings, we develop Libception, an application-level shim library that implements both techniques. We then apply Libception to evaluate the performance improvements provided by aggressive prefetching and serialization for the widely-used Apache and nginx web servers as well as a custom web server, the userver, on two operating systems (FreeBSD and Linux).

Our contributions are as follows:

- We explore the application of serialization and aggressive prefetching in HTTP streaming video web servers. We evaluate the individual contributions of these techniques towards improving throughput and provide clear evidence for why they are effective. Our evaluation shows that combining aggressive prefetching with serialization provides substantially better performance than either technique alone.
- Based on our findings, we design and implement Libception, a portable, application-level shim library that implements serialized disk access and aggressive prefetching. We demonstrate that web servers can obtain the benefits of these techniques simply by dynamically linking with Libception at runtime (using LD_PRELOAD), without the need for source code

changes. Comparing a web server that we had modified to incorporate the techniques directly to the unmodified server linked with Libception, we find essentially identical performance.
- We show that the aggressive prefetching and disk I/O serialization techniques currently implemented in Libception can approximately double the peak HTTP video streaming throughput of a variety of web servers (Apache, nginx, and the userver), both on FreeBSD, and on Linux when using the default kernel parameter settings, regardless of which Linux disk scheduler is chosen.
- We discover that there is great scope for improving HTTP video streaming performance on Linux, when not using Libception, by tuning kernel parameters. In particular, by tuning parameters to improve both prefetching and serialization, we find that throughput can be more than doubled, yielding peak throughput slightly higher than that obtained with Libception. In contrast, when using Libception, kernel parameter tuning yields only marginal additional improvements.
- Finally, we use a microbenchmark to demonstrate Libception's ability to improve performance for another workload. With two instances of the utility `diff` competing for resources, Libception is shown to reduce execution time for both instances of `diff`, cutting workload completion time by over 50%.

This paper expands upon our International Conference on Performance Engineering (ICPE 2017) paper [37]. New material in this paper includes additional background information on web server architectures; a detailed description of our modification of the userver to directly incorporate serialization and aggressive prefetching; results from a low-level analysis to determine the benefits of applying serialization and aggressive prefetching individually and together; experimental results involving multiple disks; and microbenchmark performance results demonstrating Libception's effects outside of a video streaming workload.

2 BACKGROUND AND RELATED WORK

HTTP streaming video workloads are typically disk-bound due to the large size of video files and the tendency of the popularity distributions of these files to have a very long tail (meaning the large majority of content is accessed infrequently) [13]. There are three general techniques for improving the bottleneck of disk performance: file caching, disk scheduling and prefetching. Our research is primarily interested in the effects of request scheduling and prefetching. File caching plays a significant role in improving server throughput for our workload, but competes with prefetching for system memory resources [5]. For the purposes of our experiments, we simply use the kernel file caching algorithm as-is.

In the following sections we describe prior work for disk scheduling and prefetching, and discuss how the results apply to an HTTP streaming video workload. We discuss handling concurrent I/O streams, because the bulk of the research into scheduling and prefetching assumes a single-threaded workload (which is not consistent with our workload). Finally, we discuss the architectures used to implement web servers to enable the efficient servicing of many concurrent streams of video content.

2.1 Block I/O Scheduling

Block I/O schedulers play a major role in most modern operating systems, and typically act as a layer between user I/O requests and requests to block device drivers. As of kernel version 2.6.33, Linux provides three different default options for scheduling block I/O devices: NOOP, deadline, and completely fair queuing (CFQ). All three schedulers attempt to take advantage of different

aspects of temporal and spatial locality between requests in order to yield higher throughput for disk-bound workloads.

The NOOP scheduler is the least complicated of the Linux schedulers. It provides a simple first-in-first-out (FIFO) queue for requests, and also performs basic request merging [2]. The deadline scheduler maintains sector-sorted read and write queues, as well as queues which are organized by expiration times. The deadline scheduler gives priority to expired requests in the secondary queues, and otherwise batches requests from the sector-sorted queues. It is tailored towards workloads that require latency guarantees on I/O requests [2]. CFQ divides access to the disk into time slices which are allocated between groups of per-process request queues and sized by process priority. CFQ idles shortly on empty queues whose time-slices have not expired, even if other queues contain outstanding requests [2].

Linux previously offered an anticipatory scheduler (AS), which was introduced as a means to eliminate "deceptive idleness". Deceptive idleness occurs when processes leave a small data processing gap between I/O requests. During this time, naive schedulers may switch to servicing other processes, introducing seeks that can degrade system performance [11]. The abilities of AS are mostly a subset of CFQ, and as such AS was removed in version 2.6.33 of the Linux kernel [3].

The schedulers built into the Linux kernel are necessarily designed to handle a wide range of workloads. However, there are characteristics of streaming video workloads that can be exploited by specialized scheduling algorithms. There are many studies, aimed at broadcast streaming scenarios in which servers push data to clients, where scheduling is used as a means to maximize throughput [7, 9, 31]. These studies contain valuable insights, but are not directly applicable to HTTP streaming video servers, where clients individually pull requests from the server. As we will demonstrate experimentally, the choice of block I/O scheduler is not very important for HTTP streaming video workloads. Regardless of which scheduler is chosen, significant throughput benefits are gained from the use of other techniques to improve I/O.

2.2 Prefetching

Prefetching is a well-studied technique, and refers to reading data from the disk into memory before it is actually requested by the user. This allows subsequent read requests to return immediately instead of blocking on disk operations. Prior research has shown the effectiveness of using prefetching as a means of off-setting latency and CPU stalls [6, 24, 39]. Papathanasiou and Scott demonstrated that aggressive prefetching (prefetching far beyond what is requested by the user) can be used to offset request latency [22]. However, latency of disk access is not a significant concern for HTTP streaming video workloads. Clients use buffering to cope with potentially high network latencies, so lower latencies incurred by disk I/O are unlikely to affect the quality of service experienced by end users.

Instead, HTTP streaming video would benefit from using prefetching as a vehicle for increasing disk throughput. Examples of systems that have studied prefetching within this context are DiskSeen, a system that modifies the Linux kernel to introduce history-aware prefetching into the operating system [12], and libprefetch, which uses both kernel and application modifications to provide application-directed prefetching [38]. Unlike previous work, our research considers disk request serialization in combination with highly aggressive prefetching, and does not require code changes at the user level nor in the kernel.

Prefetching is commonly performed at the hardware level, in addition to the software level. We refer to prefetching done by the disk itself as "lookahead". Ruemmler and Wilkes demonstrated that lookahead could improve read times by up to 42% on Unix-based systems [29], and subsequently

showed that effective use of on-disk caches for lookahead can yield optimal results for sequential workloads by eliminating unnecessary rotational delays [28].

An important issue in prefetching concerns how much data to request with each read operation. A larger prefetch amortizes the overhead cost of a disk access over more bytes of data but can have adverse consequences, such as the eviction of useful data from the cache. Panagiotakis, et al. [21] demonstrate that using a large fixed prefetch size to service 100 sequential streams improves throughput by up to 4 times compared to not prefetching. Li, et al. [14] provide a 2-competitive algorithm that uses hard drive performance specifications to determine a prefetch size. In prior work, we found that the best prefetch size depends on both available system resources and specific workload characteristics [34]. We provide an automated algorithm for dynamically determining a good prefetch size [34], and we demonstrate that it is possible to further increase the efficiency of servicing a streaming video workload by exploiting knowledge of specific workload characteristics to implement a prefetch algorithm [33]. For this paper, we implement a simple prefetch algorithm in Libception that uses fixed-size prefetches, with a default size of 2 MB (we experiment with different fixed prefetch sizes in Sections 6.4 and 6.5). In the future, for use in a production server or for different workloads, we could implement an automated workload-specific prefetch algorithm in Libception.

2.3 Concurrent I/O Streams

It is important to consider the trade-off between reading from disk efficiently using large prefetches and servicing concurrent disk requests fairly. Panagiotakis et. al demonstrated rapid degradation of I/O throughput as additional I/O streams are introduced across a variety of Linux schedulers [21].

The Argon system [40] focused on meeting service-level agreements by using disk-head time-slicing to ensure minimum levels of throughput to competing applications. Unlike Argon's target workload, HTTP streaming video workloads only require that clients avoid re-buffering (it is not uncommon for clients to buffer about 5 – 30 seconds of video [1, 4]). Because HTTP streaming clients are insensitive to disk latency, there is scope to reduce fairness between individual clients in exchange for higher disk throughput.

2.4 Web Server Architecture

Web servers are an important class of applications that must manage hundreds or thousands of concurrent I/O streams. To handle concurrent connections efficiently, a web server must ensure that when servicing one HTTP request, the servicing of other requests is not unnecessarily blocked. In particular, when it is necessary to issue disk I/O to service one request, it does not block the servicing of a request that can be serviced from the file system cache. There are two web server architectures designed to allow concurrent processing; *thread-per-connection* (used by Apache) where each connection is serviced using its own thread or *event-driven* that makes use of non-blocking I/O so that a single event-loop thread can be used on each core to service multiple clients (used by the userver and nginx). Unfortunately, not all operating systems provide support for non-blocking disk I/O, so the event-based AMPED architecture was developed, then used to implement the Flash web server [20]. With AMPED, a pool of helper threads is used to read data from disk into memory, and then notify the main when the data is available in the file system cache.

One issue with either thread-per-connection or the AMPED architecture is tuning; an administrator must configure the number of threads in the pool. Our approach is a variation of AMPED that we call ASAP (Asynchronous, Serialized Aggressive Prefetching) which does not require configuring a pool of threads. This architecture has been used in previous work [35, 36]. We call this architecture *asynchronous* because data is read from disk using a separate thread from the main

event loop, similar to Flash/AMPED. It is *serialized* because we use a single helper thread per disk to serialize all application read requests, to reduce the interleaving of disk accesses for the portions of different files requested from the numerous clients being served. Finally, we use *aggressive prefetching* when we access the disk, to exploit the property that requests from an individual client are highly sequential because users tend to continue watching the same video.

Our work in this paper investigates the specific reasons why the ASAP architecture is effective. We also demonstrate the utility of this approach by implementing Libception to provide similar benefits to the Apache and nginx web servers, as well as other applications that involve large amounts of concurrent I/O.

3 DESIGN AND IMPLEMENTATION

In this section, we describe two different approaches to improving the throughput of web servers for HTTP streaming video workloads. First, we modify an existing web server, the userver, and use the modified server to conduct experiments to determine the contributions of individual aspects of the ASAP architecture. Second, we develop a library shim, Libception, that provides applications with the necessary capabilities for both I/O serialization and aggressive prefetching without the need for source code or kernel modifications.

3.1 ASAP Design

The `sendfile` system call is widely used in modern web servers to obtain high throughput because it or a variant is supported on most modern operating systems and it provides "zero copy" transfers of files from the file system cache to the NIC.

One way web applications use `sendfile` is to ask it to send an entire file at a time. In this case the caller blocks until all of the bytes have been sent and multiple threads (or processes) are used to ensure that some threads can make progress while others threads are blocked. This approach is used by the Thread-per-connection architecture.

Web servers that use an event-driven architecture, such as nginx and the userver, use `sendfile` in a non-blocking mode. In this case the `sendfile` call sends as much data as can be accommodated by the socket buffer before returning the number of bytes actually written. A kernel event mechanism (e.g., `select`, `poll`, `epoll`, `kevent`) is used to obtain notifications when the socket buffer can be written to again. Unfortunately, because the `sendfile` call blocks for disk I/O when the data being sent is not found in the file system cache, a special approach is required to prevent the main event loop from blocking. Our approach is to use the `SF_NODISKIO` flag (supported by FreeBSD) when calling `sendfile`, which causes `sendfile` to return `EBUSY` rather than blocking for disk I/O [26]. When `EBUSY` is returned, a separate helper thread is used to perform disk I/O (using `read`). When the helper thread finishes accessing the disk, it signals the main event loop which then reissues the `sendfile` call. This new call does not block because the requested data is in the file system cache. In addition to serializing access to a disk, a helper thread also performs aggressive prefetching by reading more data than was requested by the client.

The use of the `SF_NODISKIO` flag makes these modifications specific to a particular version of FreeBSD. Therefore, in this paper we use our modified version of the userver mainly to evaluate the effectiveness of the ASAP architecture. For a more portable implementation of serialized aggressive prefetching, we developed the Libception shim library.

3.2 Libception Design

The Libception library is portable, operates in user space, and has been tested on FreeBSD, Linux and Mac OS X. It is comprised of two components: The first component is Deception, a dynamically linked shared object which inserts itself between the application and libc calls. Deception intercepts I/O requests from the application and forwards them to the system's second component, Reception. Reception is a server process that runs separately from applications using Deception. Reception receives, services, and responds to requests generated by applications using Deception (including prefetching data when necessary). Figure 1 shows a high-level overview of the components of Libception.

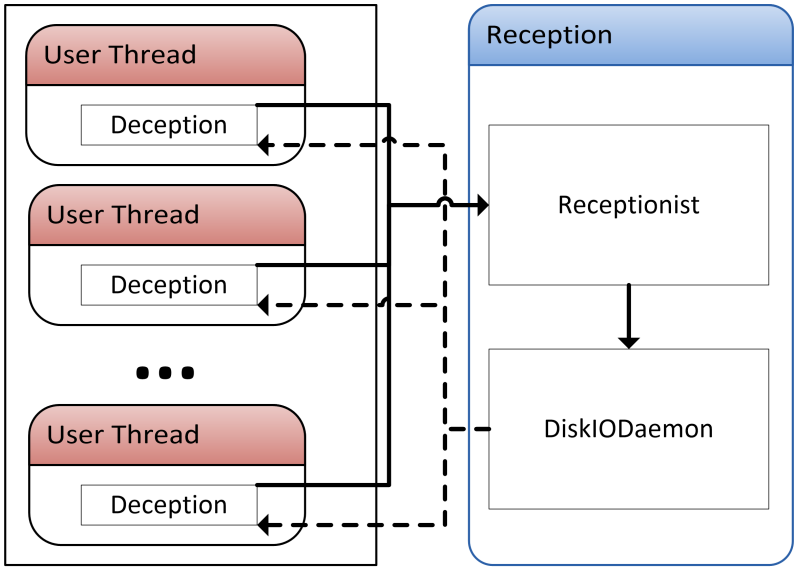

Fig. 1. Libception design

3.2.1 Deception. Deception is the primary interface for communication between user applications and the Libception library. It is implemented as a shared object that is dynamically linked at launch-time by the application that wishes to make use of it. On Unix-based operating systems, including Linux and FreeBSD, this is done by setting the `LD_PRELOAD` environment variable when launching an application. Likewise, on Mac OS X this is accomplished by setting the `DYLD_INSERT_LIBRARIES` environment variable. Once loaded, Deception begins silently intercepting calls to a variety of libc I/O-related functions including `open`, `read` and `sendfile`. This technique allows applications to take advantage of the Libception library's benefits without requiring any source code modifications. Furthermore, because Deception is implemented in user space, it requires no changes to the underlying operating system.

Most calls to Deception perform several validity checks, and then determine whether or not the application's I/O request is already resident in the system's file cache. This is done by using the system calls `mmap` and `mincore` to check all memory blocks of the request (excluding blocks that would be prefetched if the request went to disk). If a request is found to be contained entirely in memory, Deception passes the call to the original libc function (which returns without needing to go to disk). Otherwise, disk I/O is required, and a Libception I/O request is constructed and sent to Reception using Unix sockets (which duplicates any necessary file descriptors).

Should a disk read be necessary, Deception waits on a response from Reception before it proceeds. To ensure transparency, Deception always finishes by executing the application's original libc call and returning to it the appropriate return codes (even if a parameter input fails sanity checks). This allows Deception to run invisibly, without affecting the guarantees of the API for the associated libc call. This in turn means that the application can expect the same control and user-functionality for libc I/O calls that it would be afforded if Libception was not being employed.

Some additional non-I/O libc functions are also intercepted by Deception (such as `getpid` and `fork`). These functions are intercepted for functionality purposes, do not communicate with the Reception layer, and are also invisible to the user (as they terminate by making the original libc calls as well). All initialization for Deception is handled transparently at launch by GCC constructor functions and all cleanup is likewise handled unobtrusively at termination by GCC destructor functions which are run when the Libception shared object is loaded and unloaded, respectively.

The basic design principles for Libception are also applicable to other operating systems, such as Microsoft Windows. Although the process for implementing these techniques on Windows are more complicated than setting an environment variable, there are several options for re-routing the Win32 API calls that deal with disk I/O. For example, a Windows-based version of Libception could use Microsoft Detours [15] to forward I/O calls to an external Reception process. We leave the details of such an implementation to future work.

3.2.2 Reception. In our experiments, Reception is launched as a separate user space process by the user, prior to running applications with the Deception shim. It could also be launched as a daemon process in a production environment. Reception is primarily responsible for accepting, serializing and servicing incoming I/O requests from one or more Deception shims. Reception furthermore modifies requests as necessary (for example, by enlarging read sizes to introduce readahead), executes the prefetch itself, and finally responds to Deception. These tasks are divided between a single Receptionist thread, and one or more DiskIODaemon threads.

The Receptionist thread acts as a server, accepting requests over Unix sockets. I/O requests received by the Receptionist thread are sorted by device, and are placed in one of multiple queues to be serviced by the appropriate DiskIODaemon thread. The Receptionist server thread also may delegate requests to a separate maintenance thread that performs utility tasks, such as statistics collection and aggregation.

DiskIODaemon threads are responsible for performing I/O for different devices. By default, Reception runs in a single DiskIODaemon mode, with all requests being serviced sequentially in the system by one thread. Alternatively, the user may set Reception to create individual DiskIODaemon threads for individual devices in the system. In this case, the Receptionist de-multiplexes incoming I/O requests to the DiskIODaemon threads based on the underlying device for the file descriptor in the request.

Regardless of which mode is selected, DiskIODaemon threads each use their own lock-protected request queue, which is filled by the Receptionist, and drained by the DiskIODaemon thread. Only one request is removed from this queue at a time by the DiskIODaemon thread, ensuring that I/O to whichever device it is servicing is serialized. As requests are pulled from this queue, they are expanded to include any necessary prefetch information, and are then sent to disk. Once finished servicing the request, the DiskIODaemon thread sends a message to the Deception shim that made the request, allowing it to unblock and proceed.

Libception additionally contains options that allow it to perform prefetch-free serialization, serialization-free prefetching, and simple request tracking without either serialization or prefetching (useful, for instance, in latency profiling or other statistics gathering). Serialization-only mode indicates to Libception not to extend reads provided by the user and is identical to prefetching

with a prefetch size of zero. Prefetch-only mode transfers the burden of extending and performing requests from Reception to Deception. In this mode, instead of communicating with Reception, Deception shims simply immediately extend their requests and perform application-side `pread` calls before completing the original I/O request.

4 EXPERIMENTAL METHODOLOGY

We used the methodology described in [35] to generate experimental workloads and benchmarks. Our workload represents a large number of HTTP streaming video clients requesting videos with characteristics similar to requests for YouTube videos in 2011. We use a small number of client machines to generate traffic simulating thousands of concurrent sessions. Each session represents an end user viewing video. The video selected for each session is chosen using a Zipf distribution with an α value of 0.8. This video is watched for some fraction of its duration. It is an important characteristic of video workloads that users do not typically watch to the end of a video, so this property is reflected in our workloads: The percentage of a video that any given user requests is in line with the watching patterns of a typical YouTube-like workload.

Another important characteristic of our workload is that the network is not the primary bottleneck. Netflix, for example, uses Open Connect Appliance (OCA) servers that have operational throughputs between 9 Gbps and 36 Gbps [17], but are provisioned with up to 40 Gbps of network capacity [19]. Instead, the workload is heavily disk-bound. This can be seen in the findings of our recent work characterizing the properties of a Netflix video workload [33]. Video providers maintain a large catalogue of content in order to appeal to a broad audience. Netflix, for instance, has a catalogue which is approximately 2 Petabytes in size [33]. Large video libraries have long tails [13], meaning they contain a large amount of infrequently viewed content. Cost effective storage is achieved by storing videos on large, inexpensive hard disk drives and the infrequent access means that the data being requested must be serviced from disk.

Videos are stored on the server hard drives by storing each video in a separate file. We stored data for a video in a single file rather than multiple chunks because we found this approach is more efficient in prior experiments [36]. The disk is populated with 20,000 video files which have an average duration of 265 seconds, with a similar distribution as YouTube videos [8]. The files represent videos with a fixed bit rate of 420 kbps. This bit rate was chosen based on information available at the time of creation of the benchmark [10]. With these duration and bit rate characteristics, our average file size is 13 MB.

Each client session consists of a sequence of requests for 10 second intervals of video data, which are 0.5 MB in size. The first three requests in a session are made as quickly as the server can deliver the results, then subsequent requests are made on a fixed 10 second interval. This represents the filling of a playout buffer at the beginning of a session, followed by requests to refill the buffer as it is consumed at the bit rate of the video. This is a simplified model of a pull-based video client; it does not attempt to represent user actions like pausing the video, skipping to different points in the playback, or changing the quality level of the video. These actions were rare in the YouTube workload we modelled, but our methodology is flexible enough that we could represent these actions for workloads where they are significant.

The average duration of a video session is 160 seconds, using a distribution derived from real-world measurements. During experiments, video sessions are started at a chosen rate, using a Poisson distribution for session initiation. Experiments consist of 14,400 sessions, with a maximum of about 650 concurrent sessions. A total of 118 GB of video data is requested from 6581 different videos. Each video is viewed 2.2 times on average and 67.8% of videos are requested a single time during the experiment.

The clients monitor the service time for each request, and if it takes longer than 10 seconds to completely receive the data from a request, the client terminates the session and stops making further requests. For our experiments, we are interested in determining the highest aggregate client request rate that can be serviced, so that we can compare different web servers and configurations. To determine this rate, the *maximum failure-free rate*, we conduct a number of benchmark runs with a range of different aggregate rates of requests. From this, we determine the highest rate that results in fewer than 0.3% session failures. This value was chosen to permit clients to perform a small but very limited amount of re-buffering.

The clients are connected to the server over a local-area network with high bandwidth and low delay. To better represent the conditions available to real-world users, we use dummynet [25], which allows us to simulate different network types. We throttle 50% of client sessions in the workload to 3.5 Mbps, and the other 50% of client sessions to 10.0 Mbps, in order to represent a mix of end-user cable and DSL access speeds. Furthermore, we add 50 ms of delay to the network in each direction in order to model more realistic wide-area network conditions.

The equipment and environment we use to conduct our experiments were selected to ensure that network and processor resources are not a limiting factor in the experiments. We use two server machines, one for FreeBSD experiments and the other for Linux experiments. Both are HP DL380 G5 systems which contain two four-core Intel E5400 2.8 GHz processors and 8 GB of RAM. The Linux system uses Ubuntu 12.04 with a Linux 3.2.0 kernel, and a Western Digital Red (WDC WD10EFRX) 1.0 TB 5,400 RPM 3.5 inch SATA3 disk to store video files (chosen for its combination of relatively high throughput and low power consumption). The FreeBSD system uses FreeBSD 8.0 and stores videos on an HP 146 GB 10,000 RPM 2.5 inch SAS disk. Table 1 shows the raw throughput and seek times for the two different drives, obtained using the `diskinfo` command. Note that these are raw throughput numbers that do not include file system overhead so actual application throughput numbers will be lower. We use FreeBSD 8.0 so we can try to match the performance obtained previously with a modified web server [35]. Video files accessed during experiments are stored on a separate disk from the operating system.

Drive	Throughput (MB/s)			Seek Time (msec)		
	Outer	Middle	Inner	Full	1/2	1/4
HP	122	107	73	10.5	8.0	6.8
WD Red	133	115	67	32.6	23.1	18.9

Table 1. Raw Throughput and Seek Times

In all of our experiments client machines are used to generate the load of thousands of viewers on the servers. In Section 5 twelve client machines are used and in the remainder of the paper we use four clients. Therefore, the workloads are slightly different and as a consequence, care must be taken if comparing the disk throughput results shown in Table 2 and Figure 7. Each of the client systems contains either dual 2.4 or dual 2.8 GHz Xeon processors and 2 GB or 3 GB of memory. Client machines run Ubuntu 10.04 on top of version 2.6.32-30 of the Linux kernel. They also use a version of httperf [16] that was modified locally to support new features in a workload generation module named wsesslog. These modifications also allow the clients to track additional statistics. All clients are connected to the server via multiple 1 Gbps network links through multiple 24-port switches, helping to further ensure that the network is not a bottleneck.

5 UNDERSTANDING ASAP

In previous work [35, 36], we demonstrated the overall benefits of using the ASAP architecture to modify a web server, and focused on the problem of choosing the most effective prefetch size based on workload characteristics. However, our investigation used a web server that was modified to apply both serialization and aggressive prefetching when accessing the disk. In this section, we perform a low-level analysis to determine the individual benefits of applying serialization or aggressive prefetching techniques on their own, to gain insights into the reasons that ASAP is effective in improving throughput.

We conduct experiments with four different configurations of the userver on FreeBSD. One configuration is unmodified and has no prefetching or serialization (*vanilla*), one serializes read requests but does not do any prefetching (*serialized*), one uses aggressive prefetching without serialization (*prefetching*), and one implements the full ASAP architecture (*ASAP*). We provide details of the implementation of each configuration in the sections that follow.

The results of all experiments conducted using these different configurations are contained in Table 2. For the *vanilla* and *serialized* configurations of the userver, the *size* column specifies the size of network socket buffer used to send requested data. For the *prefetching* and *ASAP* configurations of the userver, the *size* column specifies the prefetch size. The reason for these choices and significance of these results are discussed in the sections that follow.

userver version	Size KB	Request Rate req/s	Disk Tput MB/s	ms/t
vanilla	128	60.0	18.8	4.31
vanilla	2048	70.0	21.6	2.81
serialized	128	60.0	18.8	3.67
serialized	512	120.0	34.4	2.00
serialized	2048	120.0	34.4	2.00
prefetching	2048	120.0	37.4	1.83
prefetching	4096	140.0	47.0	1.53
ASAP	512	120.0	34.5	2.57
ASAP	2048	200.0	55.6	1.40

Table 2. Disk metrics across userver versions for different prefetch/socket buffer sizes (FreeBSD, HP drive)

There are two performance measurements reported, disk throughput (*Disk Tput*) and transaction time (*ms/t*). In FreeBSD, all disk accesses are performed with a series of transactions that can be up to 128 KB in size. If a function specifies a read larger than 128 KB, the request is split into multiple transactions which are issued iteratively. The execution time for a transaction consists of the time necessary to seek to the location on disk, as well as the time to read the data. If two consecutive transactions are contiguous on disk and the transactions are issued in quick succession, the second seek time will be zero; otherwise the time depends on the distance between the transactions. Differences in the average transaction time reflect both the number of contiguous transactions and the distance between non-contiguous transactions.

We compute the transaction time using two values reported by `iostat`: the number of transactions n that were performed in each second of elapsed time and the percentage of the time the disk was busy b. The transaction time is reported in milliseconds per transaction (ms/t) and calculated as: 1000 ms $* b/n$. We do not report transaction sizes because they are approximately the same

for all experiments (128 KB). Since the average transaction size is stable, transaction times can be directly compared between experiments.

To illustrate the reason for the differences between transaction times, we used dtrace to intercept calls to the `remove_bioq` kernel function, and recorded the physical location of each disk transaction, in the order they were scheduled by the kernel. We then created graphs, such as the one shown in Figure 2, that show short representative portions of the traces and provide insights into the reasons that serialization or aggressive prefetching improve throughput. Each of Figures 2 through 5 start from the first transaction at the same specific location on disk, but the locations of subsequent transactions vary widely due to differences in the algorithms used by the different userver configurations, and the non-determinism of concurrent clients.

In the following four sections, we analyze the performance of each configuration of the userver and show that while the serialized and prefetching configurations have lower transaction times and therefore higher disk throughput than the vanilla userver, ASAP is clearly the best alternative. Then in Section 5.5, we compare the spatial locality of disk accesses over the entire workload for each userver variation.

5.1 Vanilla userver

Using the vanilla userver, we attempted to improve transaction times by changing the socket buffer size. The socket buffer size indirectly determines the size of read requests, because we call `sendfile` in response to a notification that the socket buffer could be refilled. The larger the socket buffer, the higher the potential for larger refills. We report the results of experiments using socket sizes of 128 KB and 2048 KB, in Table 2, in the rows labelled *vanilla*. Using the larger socket buffer size of 2048 KB increases disk throughput to 21.6 MB/s, compared to the 18.8 MB/s when using 128 KB sockets. This leads to a corresponding increase in the maximum failure-free rate to 70 req/s, which is the highest request rate we could achieve with the vanilla userver.

Figure 2 shows a short trace of disk accesses by the vanilla userver when using a socket buffer size of 2048 KB at a request rate of 70 req/s. The vanilla userver uses many processes to service clients and access the disk, so there are many instances when disk accesses are interleaved between multiple files located at different locations on disk. This interleaving of disk accesses causes many extra seeks and lower disk throughput, which we can increase using serialization.

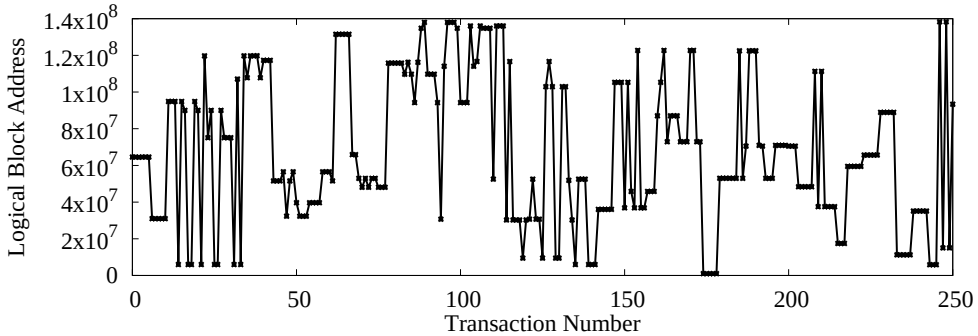

Fig. 2. Disk access pattern of vanilla userver (FreeBSD, HP drive)

5.2 Serialized userver

In order to quantify the effect of the interleaving of disk transactions, we modified the userver to serialize disk reads using a semaphore that is shared among all processes servicing client requests. When `sendfile` (with the `SF_NODISKIO` flag) indicates that data is not in the file system cache, a process waits on the semaphore before calling `sendfile` again without the `SF_NODISKIO` option. When `sendfile` returns, the process signals the semaphore and continues processing. With this modification, disk accesses by concurrent processes are serialized, without changing any of the other implementation details of the userver.

Figure 3 shows the low-level access pattern when using the serialized userver and a socket buffer size of 2048 KB, at a request rate of 110 req/s. Disk access is no longer interleaved as heavily compared to the vanilla userver, so the number of long-distance seeks is greatly reduced.

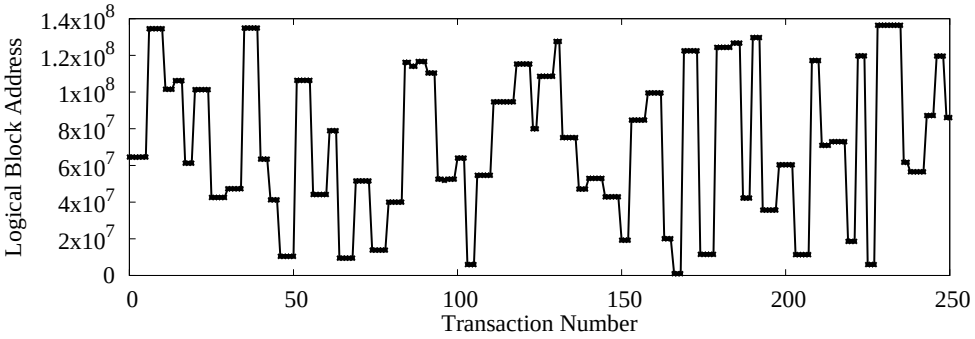

Fig. 3. Disk access pattern of serialized userver (FreeBSD, HP drive)

We repeated this experiment using 3 different socket buffer sizes, and the results are labelled *serialized* in Table 2. The results when using 128 KB socket buffers are similar to the vanilla userver, but there is a large improvement in disk throughput when using a 512 KB or 2048 KB socket buffer (from 18.8 to 34.4 MB/s). The larger socket buffer sizes enable an increase in the maximum failure-free rate from 60 req/s in the worst case to 120 req/s in the best case, which is also an improvement from the best vanilla throughput of 70 req/s.

However, there is no difference in the results when using either 512 KB or 2048 KB socket buffer sizes. This is due to the use of 512 KB client request sizes in our workload. The extra space in the 2048 KB socket buffers beyond 512 KB is never used, so the results will be identical for any socket buffer size of 512 KB or larger with our workload. One could imagine improving this situation by changing the clients and video encoding to implement larger requests. However, this reduces the granularity at which rate adaptation changes can occur [4].

5.3 Prefetching userver

The prefetching configuration of the userver issues larger disk I/O requests to the kernel than the vanilla userver, but does not serialize those larger requests.

For the *prefetching* configuration, we again used the `SF_NODISKIO` flag to detect when disk accesses are necessary. Instead of using `sendfile` to access the disk, we instead issue a `read` system call to prefetch a specified amount of data from disk. We do not make direct use of the data returned by `read`, this function is called solely for the side effect of reading data into the file system cache so it can be accessed by a subsequent call to `sendfile`. The net effect is that amount of data

requested (and prefetched) on a file cache miss is determined by the `read` call rather than the size of the client request that is used for `sendfile`.

Figure 4 shows the access pattern with 2048 KB reads, and a request rate of 120 req/s. Data is usually read in long contiguous blocks, but there are times where disk access is interleaved between multiple files.

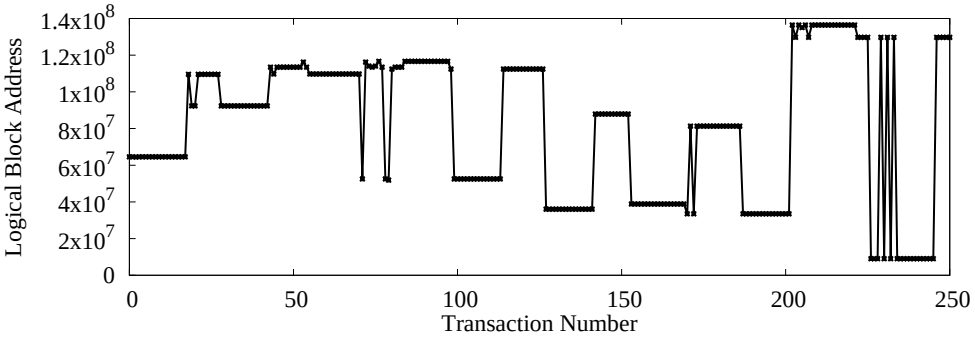

Fig. 4. Disk access pattern with prefetching userver (FreeBSD, HP drive)

Comparing the results between the prefetching and serialized configurations in Table 2 (with a size of 2048 KB) shows that disk throughput is higher for the prefetching configuration, but the request rate of 120 req/s is the same. This occurs because our workload represents typical viewers who rarely watch to the end of a video. For this reason, some prefetched data may not be subsequently requested, which wastes disk throughput. When using a prefetch size of 2048 KB, the benefit from higher disk throughput is completely offset by the cost of reading unnecessary data. But when the prefetch size is increased to 4096 KB, the benefits of higher disk throughput outweighs the costs of unnecessary disk I/O. A slightly higher request rate of 140 req/s can be achieved with the prefetching configuration with disk throughput increasing to 47.0 MB/s, compared to the 120 req/s for the serialized configuration.

5.4 ASAP userver

The serialized and prefetching configurations demonstrate the benefit of serializing disk accesses and larger read sizes, respectively. Both cases increase the number of sequential disk reads when compared with the vanilla userver. The results of applying both serialization and aggressive prefetching using the ASAP configuration are shown in Figure 5. This trace was collected using a prefetch size of 2048 KB and a request rate of 200 req/s, with disk throughput now reaching 55.6 MB/s. In this case significantly more data is read in long sequential blocks than in the previously examined cases, maximizing the benefits of aggressive prefetching by using serialization to prevent the interleaving of disk accesses that can reduce disk performance.

5.5 Sequentiality of Disk Access

In the preceding sections we show short representative excerpts of the disk I/O performed by the different configurations of the userver. In this section, we consider all of the disk accesses that occurred during the experiments to create CDFs that represent the spatial locality of entire experiments. We considered the location of each disk transaction issued during a userver experiment to find sequences of transactions that are adjacent on disk. It requires a disk seek (and rotation delay) to move from the end of one sequence of adjacent requests to the start of another, but disk

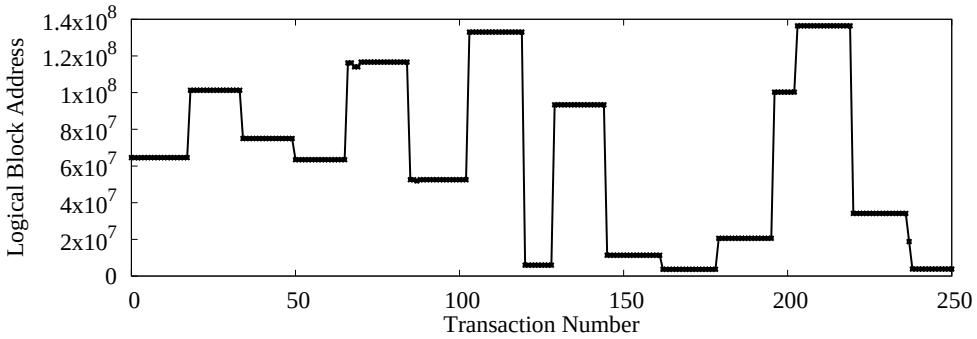

Fig. 5. Disk access pattern with ASAP userver (FreeBSD, HP drive)

transactions that are adjacent on disk do not require a seek. So experiments with longer sequences will have higher disk throughput than experiments with shorter sequences.

Figure 6 shows four different CDFs (Cumulative Distribution Functions) corresponding to the four different userver configurations. The data was obtained from the four experiments that use a 2048 KB prefetch or socket buffer size in Table 2. The CDFs show the percentage of total bytes that are read in sequences that are shorter than a given length. For example, about 10% of the total bytes read using the ASAP configuration are part of sequences that are less than 1792 KB in length. Considering all 4 CDF curves, there are a number of inflection points at 128 KB, 384 KB, 576 KB and 1920 KB. These do not directly correspond to either the client request size of 512 KB or the 2048 KB prefetch size because FreeBSD uses a 128 KB transaction size and also performs prefetching, adding a 64 KB readahead to some requests.

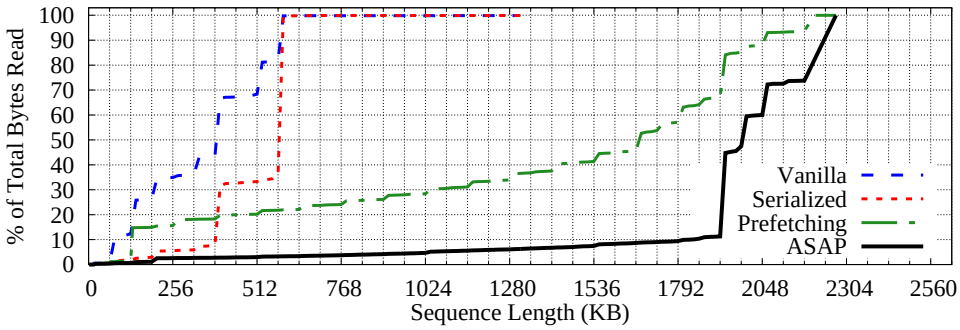

Fig. 6. CDF of percentage of bytes read in sequences during userver experiments (FreeBSD, HP drive)

The *vanilla* userver issues mostly short sequences, with 11% of bytes read in sequences shorter than or equal to 128 KB and more than 99% in sequences 576 KB or shorter. The *serialized* userver configuration prevents the interleaving of disk transactions while servicing client requests, which results in a large reduction in the proportion of short sequences compared to the *vanilla* userver. Fewer than 3% of bytes are read in sequences of 128 KB or less and there are many fewer bytes read in sequences of 384 KB or less, compared to *vanilla*. The higher proportion of longer sequences is the reason that the *serialized* configuration allows higher disk throughput than the *vanilla* userver, but with both of these configurations, there are almost no sequences longer than 576 KB.

The *prefetching* configuration results in much longer sequences. Only 22% of bytes are read in sequences of 576 KB or less and 16% of bytes are read in sequences longer than 1920 KB (i.e. only 84% are shorter). However, there is not much of a net increase in disk throughput for the *prefetching* configuration when compared to the *serialized* version (from 34.4 MB/s to 37.4 MB/s), because nearly 15% of bytes are read in sequences 128 KB or shorter (as opposed to 3% for *serialized*), which offsets the benefits of the longer sequences.

The *ASAP* configuration combines the high proportion of long sequences from the *prefetching* configuration with the low proportion of short sequences from the *serialized* configuration, with fewer than 1% of bytes in sequences shorter than 128 KB and 55% of bytes read in sequences longer than 1920 KB to enable disk throughput to reach 55.6 MB/s.

From these experiments and analysis, we conclude that the spatial locality of disk transactions issued by the ASAP configuration is much higher than the vanilla userver, and results in a nearly three-fold increase in maximum throughput compared to the unmodified vanilla userver. We also conclude that both serialization and aggressive prefetching are required to maximize the failure-free request rate. However, we were required to modify the source code of the userver to implement the ASAP architecture. In the following section we show that it is possible to utilize Libception to achieve the same improvements, without changing the source code or recompiling web servers.

6 LIBCEPTION PERFORMANCE

We now evaluate the maximum failure free throughput (often referred to henceforth as throughput) of the Apache, nginx, and userver web servers while utilizing Libception on FreeBSD and Linux servers. In all cases, we have tuned the web server to the best of our ability so that it provides the greatest maximum failure free throughput. Unless otherwise specified, the prefetch size used by Libception is 2 MB (we examine other sizes in Sections 6.4 and 6.5).

6.1 Evaluation on FreeBSD

In previous work [35, 36] we demonstrated how modifications to the userver web server to perform serialization and aggressive prefetching within the application significantly increased server throughput when servicing streaming video workloads. Unfortunately, these benefits rely on modifying the web server to use the `SF_NODISKIO` option [27] to the `sendfile` system call which is only available on FreeBSD. This flag causes `sendfile` calls that would block on disk I/O to instead return EBUSY.

The basic architecture of the userver using ASAP is to have a separate thread which performs large disk reads (thus implementing asynchronous, serialized, aggressive prefetching). This was relatively straightforward in the userver because it integrates well with its event-driven architecture and we were very familiar with the relatively small code base of the userver.

In this section we are interested in providing similar benefits to the more widely used Apache and nginx web servers without directly modifying either application. Note that nginx running on FreeBSD is of interest because the servers in the Netflix Open Connect Content Delivery Network use nginx on FreeBSD [18]. This is particularly relevant because Netflix currently accounts for a large fraction of peak Internet traffic in the United States [30].

We avoid making code modifications to the web servers because they each have a much larger code base than the userver and because Apache uses a significantly different software architecture (thread-per connection) [23]. Additionally, we want to determine if Libception can improve web server performance without using the non-portable `SF_NODISKIO` option to the `sendfile` system call.

Figure 7 shows the maximum failure free throughput obtained when using each of the Apache (labelled "A"), nginx (labelled "N") and userver (labelled "U") web servers. Throughput is shown without Libception (labelled "Vanilla"), when using Libception (labelled "Libception"), and for the modified version of the userver that uses the `SF_NODISKIO` option (labelled "ASAP"). Additionally, this graph shows both the disk throughput and the web server throughput as observed by all of the client machines.

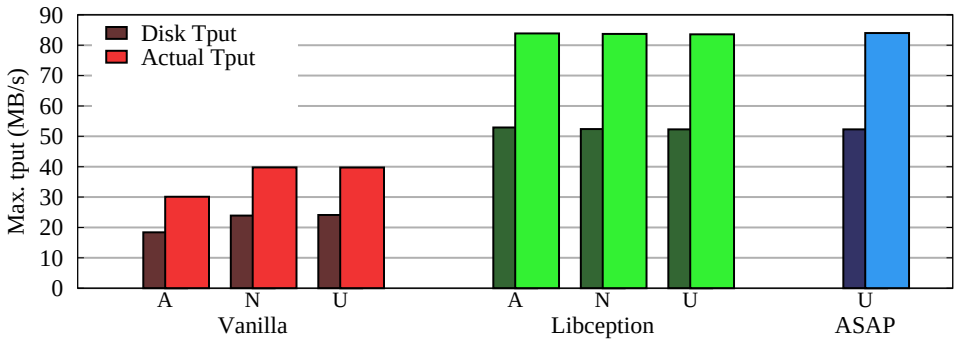

Fig. 7. FreeBSD throughput without and with Libception and with ASAP (HP drive)

These results show that Libception is able to more than double disk throughput and total server throughput for all three web servers. As well, when using Libception, the maximum failure free throughput obtained by each server is equal to that obtained by the modified ASAP userver. This is despite Libception's use of `mincore` prior to each call to `sendfile` (rather than relying on the non-portable `SF_NODISKIO` option) to determine whether or not data needs to be prefetched before calling `sendfile`. Although previous work reports that `mincore` call overhead can be significant [26], these video server workloads are disk-bound and can therefore easily tolerate the increase in CPU overhead.

It is worth pointing out that, in these experiments, not all of the data that is requested needs to be read from disk. For example, two clients requesting the same video content in quick succession will only require the data to be read from disk one time. Therefore, the difference between the total server throughput and the disk throughput is due to file system cache hits.

To the best of our knowledge, the version of FreeBSD used for these experiments does not provide any options to control the block I/O scheduler. There were also relatively few options available to influence kernel prefetching decisions and we were not able to significantly improve throughput using kernel parameters.

6.2 Evaluation on Linux

As noted previously, one of the key goals of Libception is to provide improved throughput for HTTP video web servers using techniques that are portable across different Unix-based operating systems. As a result, we now examine the performance of Apache, nginx and the userver on Linux. Recall that the disks used on the FreeBSD and Linux systems are different so we can not compare performance across the different operating systems.

Figure 8 shows the disk throughput and maximum failure free throughput obtained using each of the different web servers on Linux running with and without Libception. As was the case for FreeBSD, Libception again provides significant improvements in disk and server throughput. On Linux server throughput is increased by a factor of about 2.5 times when using Libception.

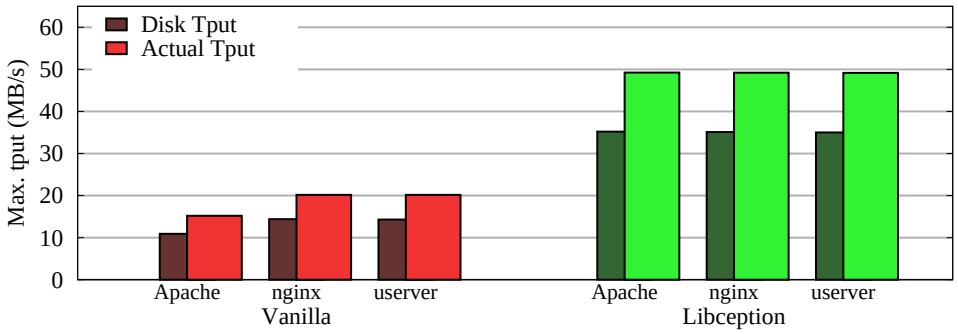

Fig. 8. Linux throughput without and with Libception (WD drive)

To our surprise, despite years of research on prefetching techniques in operating systems, these applications and workloads do not appear to perform very well on either FreeBSD or Linux.

6.3 Evaluating Linux Block I/O Schedulers

The default Linux configuration on our system uses the CFQ block I/O scheduler [2]. We expected that the anticipatory nature of the Linux CFQ scheduler might be well-suited to this workload. Because web servers simultaneously process requests from thousands of clients, we expected that blocks from different requests might provide reordering opportunities that could be exploited by CFQ to improve disk and server throughput. For completeness we now examine the performance of all web servers with each of the three block I/O schedulers available in Linux. Figure 9 shows the throughput obtained without Libception while using the CFQ (labelled "C"), deadline (labelled "D") and NOOP (labelled "N") schedulers. Figure 10 shows the results obtained using the same schedulers but this time while using Libception.

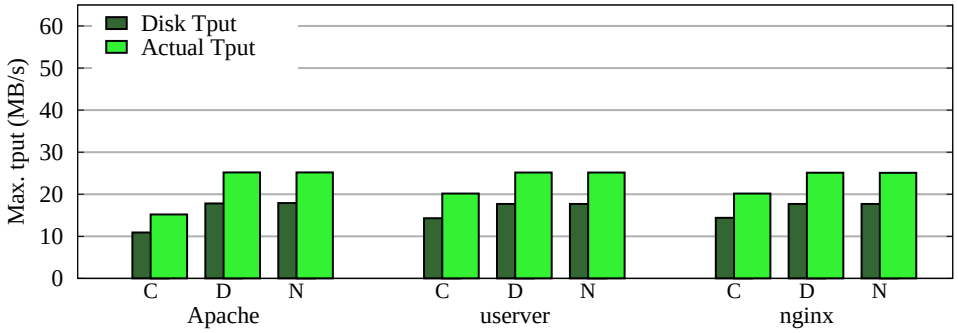

Fig. 9. Servers without Libception using different block I/O schedulers (Linux, WD drive)

Interestingly, Figure 9 shows that without Libception the server throughput is slightly higher with the deadline and NOOP schedulers than with the CFQ scheduler. On the other hand, when using Libception (see Figure 10) all web servers obtain the same maximum failure free throughput of nearly 50 MB/second regardless of the block I/O scheduler used. We believe that this is because Libception is serializing all of the reads that go to disk and as a result the schedulers only ever see a single outstanding request, which leaves no room for scheduling policies to make a difference. Note that we have spent some time attempting to adjust the parameters designed to control the

behaviour of the deadline and CFQ block I/O schedulers. We did not see throughput improvements when compared with the default values.

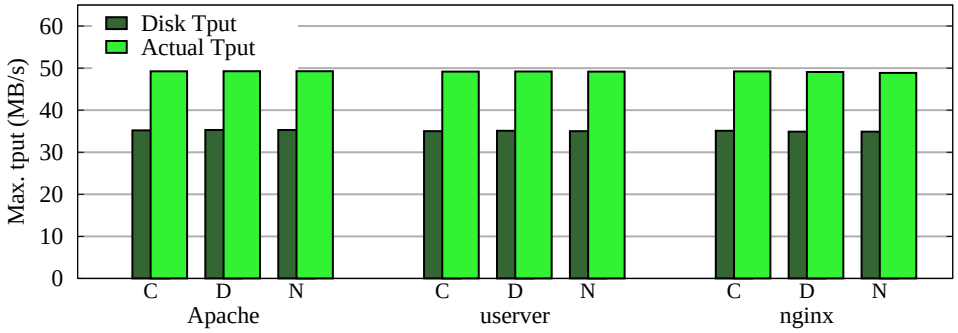

Fig. 10. Servers with Libception using different block I/O schedulers (Linux, WD drive)

In summary, the different schedulers do not significantly improve web server throughput on this workload. More importantly, Libception provides significant increases in throughput that are not possible using the block I/O scheduling algorithms.

6.4 Evaluation Insights

In this section we first conduct a sequence of experiments designed to understand the relative importance of the prefetching and serialization components of Libception. For the remainder of the paper we focus solely on the nginx server and the CFQ block I/O scheduler. We chose nginx because it is used by Netflix for serving HTTP video streaming workloads. We chose CFQ because it is the default block I/O scheduler on our Linux server and because the schedulers did not significantly affect performance when using Libception (see Figure 10).

Figure 11 shows the maximum failure free throughput obtained using nginx without Libception (labelled "Vanilla"), with Libception using only serialization (labelled "Serialized"), with Libception using only prefetching (labelled "Prefetching"), and with Libception using both serialization and prefetching (labelled "Libception"). As can be seen in this figure, serialization alone actually reduces throughput. We believe that this is primarily due to the relatively small size of reads that the application performs. These small reads, in conjunction with serialization, result in small requests being issued one at a time, which causes very poor performance. On the other hand, using prefetching without serialization does significantly increase both disk and server throughput when compared with the "Vanilla" server. Finally, by combining both aggressive prefetching and serialization, a further increase of approximately 25% beyond that of prefetching is alone is achieved. These experiments demonstrate that, while aggressive prefetches are essential, the full potential of Libception is not realized unless the requests to the disk are serialized.

In previous work [35, 36] and in all experiments in this paper up to this point, the prefetch size was set to 2 MB. We now examine a range of prefetch sizes and study the impact on the throughput of nginx while using Libception with prefetching but without serialization (see Figure 12) and with both prefetching and serialization (see Figure 13)

Figure 12 shows that, without serialization, throughput does not improve until a prefetch size of 2 MB or greater is used. It also demonstrates that prefetch sizes of 3 and 4 MB provide slightly better throughput than a prefetch size of 2 MB.

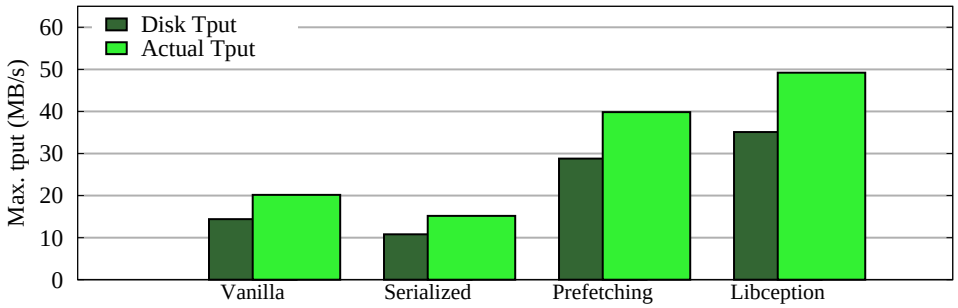

Fig. 11. nginx Vanilla, and nginx with Libception using serialization only, prefetching only, and both (Linux, WD drive)

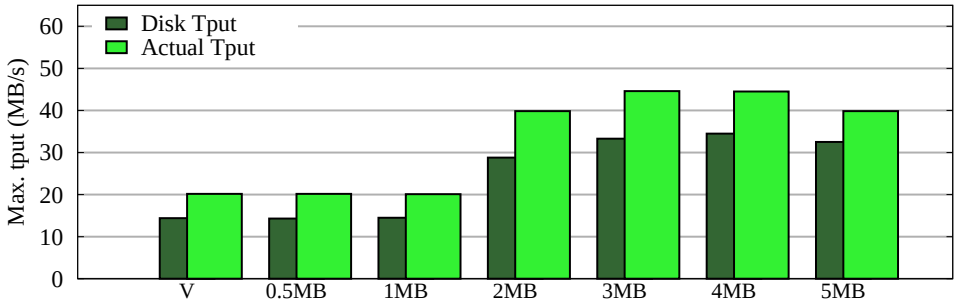

Fig. 12. nginx with Libception using various prefetch sizes without serialization (Linux, WD drive)

Figure 13 shows that when using Libception, with both serialization and prefetching, small prefetch reads actually slightly reduce server throughput. However a prefetch size of 1 MB does significantly improve server throughput, which is not the case when prefetching is used without serialization. When using both serialization and prefetching, throughput peaks with prefetch sizes of 2 – 4 MB, and also shows that serialization provides additional benefits (about 10%) when compared with prefetching alone.

Fig. 13. nginx with Libception using serialization and various prefetch sizes (Linux, WD drive)

6.5 Using Insights to Improve Linux

We now use the insights obtained from the previous section to modify Linux kernel parameters in an attempt to improve the performance of nginx when running on Linux without the use of Libception. The question being examined is: can we find and tune appropriate Linux kernel parameters in order to obtain throughput that equals that obtained using Libception?

It was not too difficult to find a Linux kernel parameter that we were able to adjust to increase the amount of data being read from disk by the kernel. When obtaining data from disk the Linux kernel may (depending on several factors the details of which aren't relevant to this discussion) extend the read beyond what the user has requested. Roughly speaking, a readahead size is tracked per file and under proper conditions grows for each successive read. However, the maximum readahead size for all files on a device is limited by the kernel parameter `read_ahead_kb`, which can be set differently for each device. The default setting for this parameter for the version of Linux used in our experiments is 128 KB. This is quite small compared to the aggressive prefetches we use in Libception (e.g., 2 MB in many cases) in order to obtain significant increases in throughput.

Figure 14 demonstrates how the throughput of nginx changes as the value of the readahead parameter is increased. As an example of how we set the readahead size to 1024 KB on disk drive used to store video files (`/dev/sdb1`), we use the command `blockdev --setra 1024 /dev/sdb1`. As can be seen in this graph, `read_ahead_kb` values of 0.5 MB and 1 MB provide significant improvements over the default value of 128 KB (labelled "V" for Vanilla). Larger values for `read_ahead_kb` do not perform as well as 1 MB. While throughput obtained with 1 MB is about 45 MB/second it is not as high as that obtained using Libception (50 MB/second). Understanding why this is the case involved significantly more work.

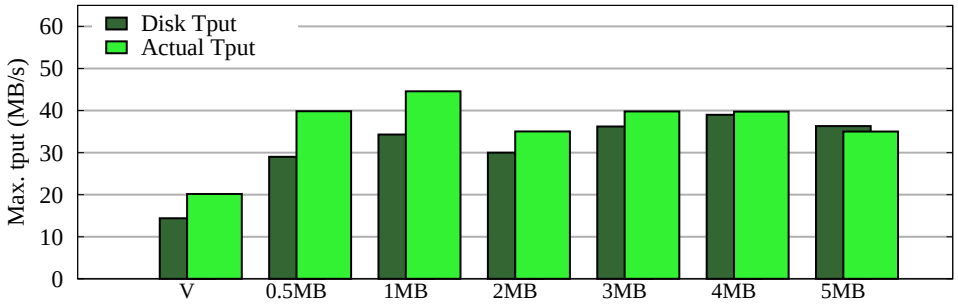

Fig. 14. nginx without Libception using various `read_ahead_kb` sizes (Linux, WD drive)

Although the size of the prefetch for files were actually reaching the limit imposed by `read_ahead_kb`, using the Linux `blktrace` facility we were able to determine that requests to the disk were being limited to 512 KB. We believe that, as a result, some requests for reads to different files were being interleaved (this is similar to the Libception case where prefetching is used but serialization is not).

By examining the Linux kernel source we were eventually able to determine that another kernel parameter was placing further limits on the size of reads. This value `/sys/block/sdb/queue/max_sectors_kb` (for the disk `/dev/sdb`) uses the default size of 512 KB. We expect that the default values for these two limits (`read_ahead_kb` and `max_sectors_kb`) are chosen in order to ensure fairness across different disk requests and to keep latencies low.

To ensure that `max_sectors_kb` does not limit the size of disk reads we set its value to 16 MB. We then adjust `read_ahead_kb` and examine the throughput obtained by nginx. Figure 15 shows the results of these experiments and demonstrates the importance of setting both kernel parameters to proper values in order to obtain good throughput on this workload. In this case a `read_ahead_kb` size of 1 or 2 MB, now results in throughput of about 58 MB/second, which is slightly better than that obtained using Libception.

Fig. 15. nginx without Libception using various `read_ahead_kb` sizes, with `max_sectors_kb` = 16 MB (Linux, WD drive)

Figure 16 now shows results obtained using Libception in addition to using modified Linux kernel parameters. In this case `max_sectors_kb` is set to 16 MB to ensure that it does not limit read sizes and the Libception prefetch size and `read_ahead_kb` are adjusted together using the values shown along the x-axis of the graphs. The column labelled "V" is showing the vanilla case where nginx runs without Libception and the default kernel parameter values are used. Although these results show a slight improvement in maximum failure free throughput when compared with just using Libception, they are still lower than adjusting the kernel parameters alone and not using Libception.

Fig. 16. nginx with Libception using various prefetch and `read_ahead_kb` sizes, with `max_sectors_kb` = 16 MB (Linux, WD drive)

We observe that for some configurations of the experiments conducted in this section, the disk throughput exceeds the total server throughput. In these cases data that is being prefetched is actually being evicted before it is requested by and sent to the client. This means that, for the

amount of memory in the current system, prefetching has become too aggressive and is causing evictions. We expect that in these cases, increasing the amount of memory in the system will likely increase total server throughput because there will be less memory pressure created by aggressive prefetching. It seems that, in HTTP video streaming workloads, it may be more important to have memory acting as a buffer for large aggressive prefetching (in order to obtain high disk throughput) than as a file system cache.

6.6 Evaluating Latencies

When servicing HTTP video server workloads, server throughput is strongly influenced by disk throughput. This is because video services like YouTube and Netflix have large numbers of videos that are viewed infrequently (i.e., the popularity distribution of videos has a long tail). In order to improve disk throughput Libception prefetches relatively large amounts of data and serializes access to each disk. This naturally increases disk throughput while potentially increasing latencies for some requests. The potential for increased latencies should be relatively harmless when servicing video server workloads because clients are designed to be able to tolerate fairly significant latencies by using a play out buffer. The play out buffer is filled before play begins and is used to seamlessly continue playing the video even during periods where the client may experience latencies due to the network or HTTP server. For example, a client with a 10 second play out buffer can tolerate nearly 10 seconds of latency for some requests. As long as the data being requested arrives (and can be decoded before) within 10 seconds of when it is requested, the user will not experience any problems.

Before delving into further empirical analysis, it is important to note that all of our results place an implicit bound on latency. Client timeouts occur when a request has not received a corresponding response after 10 seconds. This latency includes the time to transmit the request, service the request at the server, and transmit the response. Therefore, if a client completes a session without errors, all of the latencies must have been acceptable.

In order to better understand the latencies incurred by using Libception and the aggressive tunings of Linux kernel parameters that support high server throughput, we now examine the latencies experienced by server processes. We use an existing system call tracing facility that exists in the userver to record the time required for every call to `sendfile` in memory and print those times to disk after the server has finished servicing all requests. While client-side latency measurement would give an indication of end-user quality of experience, latency measurement at the server simplifies the collection process. Furthermore, as noted in Section 4, in our workload the network does not serve as a bottleneck. Therefore, server-side latencies should be reflective of client-side latencies, minus network transfer time.

Figure 17 shows the cumulative distribution function of `sendfile` call times without Libception and with default kernel parameters (labelled "Vanilla"), with Libception, and with the aggressive kernel parameter tunings. The "Vanilla" and "Libception" lines are created using data from one run with the userver using the configuration that obtains the highest error free rate (i.e., the two userver configurations used in Figure 8). The "Aggressive Kernel Params" line was created by setting `read_ahead_kb` to 2 MB and `max_sectors_kb` to 16 MB and using the same request rate as used for the configuration of nginx shown in Figure 15.

Figure 17 shows that Libception has the smallest density of requests that are serviced in the lowest range of latencies (i.e., below 0.1 ms). This is due to the extra overhead incurred from the `mincore` system call, which is used to determine if the data being requested is in memory or if it should be prefetched from disk. In cases where the data is already in memory, the lowest latencies are achieved by calling `sendfile` directly. For requests where the desired data is not in memory

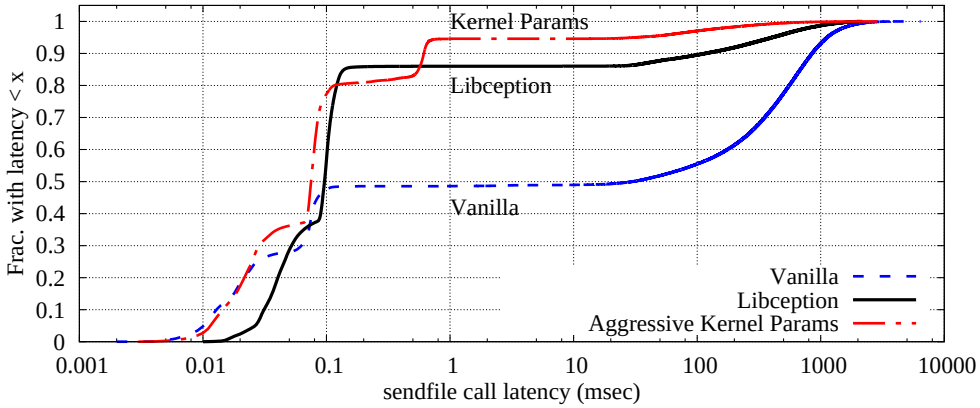

Fig. 17. userver sendfile latency CDF (Linux, WD drive)

at the time of the `sendfile` call, the "Vanilla" configuration results in a blocking call to `sendfile` because the data being requested needs to be read from disk. When a disk read occurs using either Libception or aggressively tuned kernel parameters, the application-initiated read is serialized and increased in size (to 2 MB in this case) to implement prefetching. Although this incurs some overhead for that individual request, subsequent reads will often be served directly from the file system cache with comparatively low latencies. The net result is a significant reduction in latencies for a large number of `sendfile` calls. While about 49% of `sendfile` calls take less than 1 ms under the "Vanilla" configuration, about 85% of `sendfile` calls take less than 1 ms for the Libception configuration, and about 95% of `sendfile` calls take less than 1 ms for the aggressive kernel parameters configuration.

The key observation from this experiment is that, although one might expect latencies to increase because of large serialized prefetches, for this workload they actually decrease for a large number of data requests.

6.7 Evaluation Summary

Table 3 summarizes the main results from our experiments. Entries marked with an asterisk are not possible to obtain (e.g., the ASAP userver requires an option to `sendfile` that is only available on FreeBSD so it can't be run on Linux). Entries marked with "–" are not included due to space and/or because they are unlikely to provide new insights. On FreeBSD, the use of Libception more than doubled the peak server throughput for Apache, nginx and userver. We were not able to improve the poor performance achieved by these web servers without Libception by tuning FreeBSD kernel parameters. Comparing the results for the userver modified to directly incorporate aggressive prefetching and serialization shows that the overhead of implementing these techniques in a shim library, rather than directly, is negligible for our HTTP video streaming workload (on FreeBSD the throughput obtained with Libception for all servers matches that obtained with ASAP).

These FreeBSD results raise the question of whether they might reflect some deficiency in the block I/O scheduler in that system. This prompted us to take a careful look at performance on Linux, which supports three different block I/O schedulers. We found that use of Libception approximately doubled the peak server throughput on Linux, for all three web servers, regardless of the choice of block I/O scheduler. However, using our insights from Libception as a guide and in some cases examining the Linux kernel source code, we were able to discover kernel parameters

	Default	Libception	Kernel Tuning	Libception + Kernel Tuning
FreeBSD				
nginx	39.75	83.71	*	*
Apache	30.10	83.88	*	*
userver	39.74	83.56	*	*
ASAP	83.99	–	*	*
Linux				
nginx	25.11	49.06	58.29	53.85
Apache	25.18	49.26	–	–
userver	25.16	49.18	–	–
ASAP	*	*	*	*

Table 3. Summary of Results: Throughput in MB/sec.

that we could tune to obtain slightly better performance than that with Libception. In contrast to Libception which is highly portable, we could not run our modified version of the userver on Linux since it makes use of a system call option that is not available on that system. Finally, a potential concern might be that these throughput improvements have significant cost with respect to latency, but as shown in Section 6.6 this does not appear to be the case.

7 RESULTS WITH TWO DISKS

We have conducted some experiments using FreeBSD and the HP disk drives where the HTTP streaming video workloads are serviced using two disks. In this case a file set was created on one disk using the same methodology as was used for the single-disk experiments. A copy of the file set was made on the second disk, and videos were then randomly selected from only the first or second disk (but not both). This static division prevented multiple copies of the same file from being cached in memory. *Note that this means that the workload differs from the workload used in previous experiments in this paper because the same number of clients are used but requests are now spread across two disks. As a result, one cannot directly compare the results from previous experiments using only one disk with these experiments (i.e., the throughput from the two disk experiments may not necessarily be expected to be double the throughput of the one disk experiments).* To drive an appropriate amount of load across two disks request rates were increased dramatically for these experiments.

When servicing this workload without Libception the average throughput observed on the two disks using `iostat` was 20 and 21 MB/sec and the actual server throughput obtained was 65 MB/sec. When using Libception with a prefetch size of 2 MB, the average observed disk thoughput improved to 46 and 45 MB/sec and actual server throughput increased to 134 MB/sec. This provides some evidence for our claim that Libception can be used with multiple disks and that serialization should be done on a per-disk drive basis.

8 POTENTIAL FOR IMPROVING PERFORMANCE OF OTHER WORKLOADS

Other disk I/O-bound workloads with concurrently-issued requests could also benefit from Libception. To demonstrate this, we use a simple `diff` microbenchmark that compares the corresponding files in two versions of the Linux kernel (versions 3.7.1 and 3.8-rc2), with output sent to

`/dev/null`. We measure the completion time for a workload consisting of just a single instance of this microbenchmark, as well as that for a workload with two instances running in parallel (using two copies of each kernel), with and without the use of Libception. Experiments are carried out on Linux with default kernel parameter settings using each of the three available block I/O schedulers. The results are shown in Figure 18.

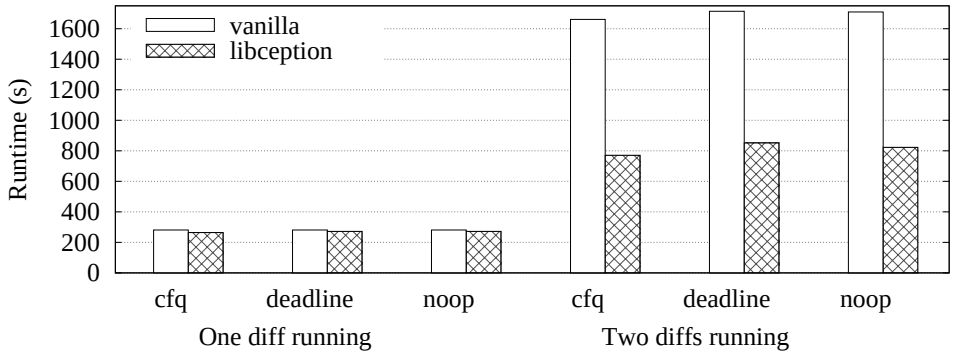

Fig. 18. `diff` runtime (Linux, WD drive)

As shown by the three pairs of bars on the left-hand side of Figure 18, when running a single instance of our `diff` microbenchmark, Libception does not significantly impact completion time. However, with two instances running in parallel generating multiple concurrent disk I/Os, Libception reduces the workload completion time by over 50% compared to results obtained without Libception (shown on the right-hand side of Figure 18). As expected, the choice of block I/O scheduler has little impact on these results.

Our understanding from examining the Linux kernel source code is that, when the first read from a file occurs, an initial readahead size is calculated based on the requested amount of data. In the case of `diff`, the requested read size is 4 KB and Linux determines that up to 4 KB will be read synchronously to satisfy the read request, which unblocks the process. Another 12 KB is also read asynchronously in anticipation of future sequential reads.

For files larger than 16 KB, additional reads will be needed. Figure 19 shows the CDF of file sizes and total number of bytes for files of different sizes for version 3.7.1 of the kernel (plots for version 3.8-rc2 are indistinguishable and are not included). The curve for total bytes shows that files larger than 16 KB account for about 50% of the total bytes read, even though they constitute less than 10% of the files. Note that even though the files are substantially smaller than in our HTTP streaming video workload, multiple reads for the same file are still common.

When two instances of our `diff` microbenchmark are run in parallel on vanilla Linux using the CFQ scheduler, disk I/O becomes much less efficient owing to the interleaving of disk I/Os from the two instances. This reduction in disk I/O efficiency is quantified in Table 4 by the average disk I/O service and wait times, as obtained from `iostat`. Comparing the average disk I/O service and wait times in Table 4 for the two `diff` instance workload on vanilla Linux, to that when using Libception, it can be seen that Libception greatly improves disk I/O efficiency for this workload. Libception also reduces the total I/O count, as can be seen from the products of the average reads per second values in Table 4 and the respective completion times shown in Figure 18. The net result is a large reduction in total disk busy time, and therefore workload completion time. These results illustrate that other applications besides HTTP streaming video servers may also benefit from using Libception.

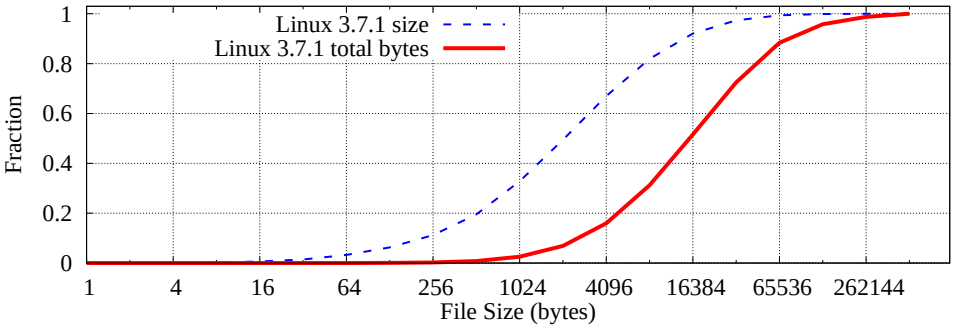

Fig. 19. CDF of file sizes and total bytes in version 3.7.1 of the Linux kernel

System and Workload	Utilization (%)	Avg Service Time (ms)	Avg Tput (MB/s)	Avg reads/s	Avg Wait (ms)
vanilla one diff	96.6	3.2	3.9	387.2	19.6
vanilla two diffs	99.7	8.2	1.3	130.2	45.0
Libception one diff	94.3	3.5	4.2	339.7	19.4
Libception two diffs	98.6	5.0	2.9	231.3	17.3

Table 4. Disk performance data from iostat

9 DISCUSSION

Perhaps surprisingly, our results show that, without Libception, none of the web servers we investigated yield good performance for HTTP video streaming workloads, on either FreeBSD, or Linux with default kernel parameter settings. This finding suggests that web server implementations are still optimized for more traditional web workloads, despite the rapid growth of HTTP video streaming. Also, web server developers might assume that, after many years of research and development, prefetching (or readahead) in operating systems, to efficiently use a magnetic disk is a "solved problem", but evidently this is not the case. Although with Linux, it was eventually possible to achieve good performance after kernel parameter tuning, it is noteworthy that such manual tuning was required. For services like Netflix and YouTube, due to the large amounts of video available, the wide variety of bit rates at which each video is encoded, and because of the large number of videos that are viewed infrequently, it is not economically viable to store it all on SSDs. As a result, obtaining good performance when servicing video from disks is still important in these settings. We have found Libception to be a relatively simple and portable platform for implementing and evaluating techniques for improving disk I/O efficiency. In particular, using Libception we were able to readily evaluate the benefits of aggressive prefetching and I/O serialization for HTTP video streaming workloads, using multiple web servers and operating systems.

As demonstrated in Section 6.4 and Section 6.5, selecting a prefetch size which is too small results in low disk throughput. Likewise, selecting a prefetch size which is too large leads to a drop in system throughput. As a result obtaining peak server throughput requires choosing the most appropriate prefetch size. In previous work we have explored how the best prefetch size and the benefits obtained from prefetching are sensitive to workload and system properties [34]. We show how the best prefetch size can be affected by the amount of available system memory, the distribution of the popularity of videos requested, hard drive characteristics, and the bit rates

of files being served. For example, we demonstrate that increasing the prefetch size can increase disk throughput if sufficient system memory is available. However, if there is insufficient system memory available then prefetched data may be evicted from the file cache before it can be used, reducing overall system throughput. Likewise, we demonstrate that the physical properties of the disk, including transfer times and seek times, significantly impact the benefits conferred by prefetching.

In order to avoid having to exhaustively and repeatedly determine the best prefetch size when workload or system characteristics change, we have designed an algorithm for dynamically and automatically adjusting the prefetch size with the goal of obtaining peak server throughput [34]. We demonstrate that a gradient descent algorithm, which minimizes a score based on a combination of disk transfer times and file system cache miss ratio, is effective at selecting prefetching sizes that result in high server throughput. This adaptive algorithm results in throughputs that rival those obtained by exhaustive manual tuning across a variety of different workload and system characteristics. We believe that such a strategy could be added to Libception, allowing it to continue providing high throughput while also eliminating the need to manually set a prefetch size.

In more recent work [32, 33] we have analyzed log files of HTTP requests to characterize the workloads of two different types of production Netflix servers. We use simulation to show that workload-specific adjustments can be used to increase server throughput. However, some of these improvements require knowledge of request streams and other information about the workload that may be difficult to infer in Libception. An interesting question is how much information can be provided to Libception, without requiring changes to web server code and whether or not it such changes can compete with the performance that can be obtained by directly modifying the web server. We intend to explore such questions in future work.

We additionally intend to explore Libception's utility for improving other workloads, for example a workload combining multiple applications with different I/O needs sharing the same disks. When all involved applications request large amounts of data, Libception can continue to provide aggressive prefetching and request serialization. However, applications that perform small, random reads will not benefit from these techniques. For these applications, large prefetching is wasteful as data cached during readahead may never be accessed or it may not be accessed before it is evicted from memory. Rather, prefetching should be minimized for these applications ensuring that only useful data is read. Similarly, serialization would not help to improve read performance for such applications since their reads are small enough that they can be handled with one disk request (and the operating system will not interleave requests from other processes with these requests). Therefore, in order to satisfy both types of applications, Libception would need to balance the trade-off between the performance of the throughput-dependent applications making large sequential reads and the latency needs of applications performing small random reads, since small reads may be delayed by the large prefetching reads.

Our work has focused on improving the throughput of streaming video content that is serviced from hard disk drives (HDDs). With large video services like Netflix where the popularity distribution of content has a long tail (i.e., there are large numbers of files that are not accessed often), storing infrequently accessed files on relatively expensive solid state devices (SSDs) rather than HDDs is not cost effective. However, SSDs are being used to store hot content in Netflix servers to increase overall throughput. Examining techniques designed for improving the throughput of systems using SSDs is a topic for future research.

10 CONCLUSIONS

HTTP video streaming has become an important class of web server workloads. Such workloads have significantly different characteristics than the web server workloads that have been the focus of most prior work on web server performance. In particular, most requests to HTTP video streaming servers are for large chunks of data stored on disk, so these servers are frequently disk-bound.

In this work, we have explored aggressive prefetching and serialization as techniques for improving the performance of HTTP streaming video web servers. Using our findings, we have designed, implemented and evaluated Libception, an application-level shim library that incorporates these techniques to improve disk access efficiency. HTTP video streaming servers can achieve the benefits of these techniques simply by using LD_PRELOAD to dynamically link with Libception at runtime, preventing the need for any source code modification. Experiments with three web servers (Apache, nginx and the userver) and two operating systems (FreeBSD and Linux) show that with the aggressive prefetching and disk I/O serialization techniques currently implemented in Libception, peak server throughput can be doubled. Only after studying the Linux kernel source code and adjusting the appropriate tunable kernel parameters was it possible to achieve performance competitive with Libception. We believe that Libception could be applied to investigate other techniques for improving HTTP video streaming performance, and possibly for improving the performance of other disk-intensive applications.

ACKNOWLEDGMENTS

We 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. Cassell, Summers and Szepesi were partially supported by NSERC graduate scholarships and Cassell and Summers were partially supported by a University of Waterloo Cheriton Scholarship. Cassell was partially supported by an OGS scholarship. The authors would also like to thank the anonymous reviewers for their comments.

REFERENCES

[1] Adhikari, V. K., Guo, Y., Hao, F., Varvello, M., Hilt, V., Steiner, M., and Zhang, Z.-L. Unreeling Netflix: Understanding and improving multi-CDN movie delivery. In *Proc. IEEE International Conference on Computer Communications (INFOCOM)* (2012), pp. 1620–1628.

[2] Axboe, J. Linux block IO – present and future. In *Proc. Ottawa Linux Symposium (OLS)* (2004), pp. 51–61.

[3] Axboe, J. Linux kernel Git commit. http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=492af6350a5ccf087e4964104a276ed358811458, 2009.

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

[5] Butt, A. R., Gniady, C., and Hu, Y. C. The performance impact of kernel prefetching on buffer cache replacement algorithms. In *Proc. ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (2005), pp. 157–168.

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

[7] Dhage, S. N., Patil, S. K., and Meshram, B. B. Survey on: Interactive video-on-demand (VoD) systems. In *Proc. IEEE International Conference on Circuits, Systems Communication and Information Technology Applications (CSCITA)* (2014), pp. 435–440.

[8] Finamore, A., Mellia, M., Munafò, M. M., Torres, R., and Rao, S. G. YouTube everywhere: Impact of device and infrastructure synergies on user experience. In *Proc. ACM SIGCOMM Internet Measurement Conference (IMC)* (2011), pp. 345–360.

[9] Ghose, D., and Kim, H. J. Scheduling video streams in video-on-demand systems: A survey. *Springer Multimedia Tools and Applications 11*, 2 (2000), 167–195.

[10] Gill, P., Arlitt, M., Li, Z., and Mahanti, A. Youtube traffic characterization: a view from the edge. In *Proc. ACM SIGCOMM Internet Measurement Conference (IMC)* (2007), pp. 15–28.
[11] Iyer, S., and Druschel, P. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *ACM Symposium on Operating Systems Principles (SOSP)* (2001), pp. 117–130.
[12] Jiang, S., Ding, X., Xu, Y., and Davis, K. A prefetching scheme exploiting both data layout and access history on disk. *ACM Transactions on Storage (TOS) 9*, 3 (2013), 10:1–10:23.
[13] Kasbekar, M. On efficient delivery of web content. GreenMetrics Keynote Talk, 2010.
[14] Li, C., Shen, K., and Papathanasiou, A. E. Competitive prefetching for concurrent sequential I/O. In *Proc. ACM European Conference on Computer Systems (EuroSys)* (2007), pp. 189–202.
[15] Microsoft. Detours. https://www.microsoft.com/en-us/research/project/detours/.
[16] Mosberger, D., and Jin, T. httperf – a tool for measuring web server performance. *ACM SIGMETRICS Performance Evaluation Review (PER) 26*, 3 (1998), 31–37.
[17] Netflix. Appliance hardware. https://openconnect.netflix.com/en_gb/hardware, 2017.
[18] Netflix. Appliance software. https://openconnect.netflix.com/en_gb/software, 2017.
[19] Netflix. Requirements for deploying embedded appliances. https://openconnect.netflix.com/en_gb/requirements-for-deploying, 2017.
[20] Pai, V., Druschel, P., and Zwaenepoel, W. Flash: An efficient and portable Web server. In *Proc. USENIX* (1999).
[21] Panagiotakis, G., Flouris, M. D., and Bilas, A. Reducing disk I/O performance sensitivity for large numbers of sequential streams. In *Proc. IEEE International Conference on Distributed Computing Systems (ICDCS)* (2009), pp. 22–31.
[22] Papathanasiou, A. E., and Scott, M. L. Aggressive prefetching: An idea whose time has come. In *Proc. USENIX Workshop on Hot Topics in Operating Systems (HotOS)* (2005).
[23] Pariag, D., Brecht, T., Harji, A., Buhr, P., Shukla, A., and Cheriton, D. R. Comparing the performance of web server architectures. In *Proc. ACM European Conference on Computer Systems (EuroSys)* (2007), pp. 231–243.
[24] Patterson, R. H., Gibson, G. A., Ginting, E., Stodolsky, D., and Zelenka, J. Informed prefetching and caching. In *Proc. ACM Symposium on Operating Systems Principles (SOSP)* (1995), pp. 79–95.
[25] Rizzo, L. Dummynet: a simple approach to the evaluation of network protocols. *ACM SIGCOMM Computer Communication Review (CCR) 27*, 1 (1997), 31–41.
[26] Ruan, Y., and Pai, V. S. Making the "Box" transparent: System call performance as a first-class result. In *Proc. USENIX Annual Technical Conference (ATC)* (2004).
[27] Ruan, Y., and Pai, V. S. Understanding and addressing blocking-induced network server latency. In *Proc. USENIX Annual Technical Conference (ATC)* (2006).
[28] Ruemmler, C., , and Wilkes, J. An introduction to disk drive modeling. *IEEE Computer 27*, 3 (1994), 17–28.
[29] Ruemmler, C., and Wilkes, J. UNIX disk access patterns. In *Proc. USENIX Winter Conference* (1993), pp. 405–420.
[30] Sandvine. Global Internet phenomena report, 2012.
[31] Steinmetz, R. Multimedia file systems survey: Approaches for continuous media disk scheduling. *Elsevier Computer Communications 18*, 3 (1995), 133–144.
[32] Summers, J. *Understanding and Efficiently Servicing HTTP Streaming Video Workloads*. PhD thesis, University of Waterloo, 2016. https://uwspace.uwaterloo.ca/handle/10012/10956.
[33] Summers, J., Brecht, T., Eager, D., and Gutarin, A. Characterizing the workload of a Netflix streaming video server. In *Proc. IEEE International Symposium on Workload Characterization (IISWC)* (2016).
[34] Summers, J., Brecht, T., Eager, D., Szepesi, T., Cassell, B., and Wong, B. Automated control of aggressive prefetching for HTTP streaming video servers. In *Proc. ACM International Conference on Systems and Storage (SYSTOR)* (2014), pp. 5:1–5:11.
[35] Summers, J., Brecht, T., Eager, D., and Wong, B. Methodologies for generating HTTP streaming video workloads to evaluate web server performance. In *Proc. ACM International Systems and Storage Conference (SYSTOR)* (2012), pp. 2:1–2:12.
[36] Summers, J., Brecht, T., Eager, D., and Wong, B. To chunk or not to chunk: Implications for HTTP streaming video server performance. In *Proc. ACM International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)* (2012), pp. 15–20.
[37] Szepesi, T., Cassell, B., Brecht, T., Eager, D., Summers, J., and Wong, B. Using Libception to understand and improve HTTP streaming video server throughput. In *Proc. International Conference on Performance Engineering (ICPE)* (2017), pp. 51–62.
[38] VanDeBogart, S., Frost, C., and Kohler, E. Reducing seek overhead with application-directed prefetching. In *Proc. USENIX Annual Technical Conference (ATC)* (2009).
[39] Varki, E., Hubbe, A., and Merchant, A. Improve prefetch performance by splitting the cache replacement queue.

In *Proc. IEEE International Conference on Advanced Infocomm Technology (ICAIT)* (2012), pp. 98–108.

[40] Wachs, M., Xu, L., Kanevsky, A., and Ganger, G. R. Exertion-based billing for cloud storage access. In *Proc. USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)* (2011).