Circus: Opportunistic Block Reordering for Scalable Content Servers

Stergios V. Anastasiadis Rajiv G. Wickremesinghe Jeffrey S. Chase
Department of Computer Science, Duke University
Durham, NC 27708, USA
{stergios,rajiv,chase}@cs.duke.edu

Abstract

Whole-file transfer is a basic primitive for Internet content dissemination. Content servers are increasingly limited by disk arm movement given the rapid growth in disk density, disk transfer rates, server network bandwidth, and content size. Individual file transfers are sequential, but the block access sequence on a content server is effectively random when many slow clients access large files concurrently. Although larger blocks can help improve disk throughput, buffering requirements increase linearly with block size.

This paper explores a novel block reordering technique that can reduce server disk traffic significantly when large content files are shared. The idea is to transfer blocks to each client in any order that is convenient for the server. The server sends blocks to each client opportunistically in order to maximize the advantage from the disk reads it issues to serve other clients accessing the same file. We first illustrate the motivation and potential impact of opportunistic block reordering using a simple analytical model. Then we describe a file transfer system using a simple block reordering algorithm, called Circus. Experimental results with the Circus prototype show that it can improve server throughput by a factor of two or more in workloads with strong file access locality.

1 Introduction

Many Internet services are based on whole-file transfers or *file downloads*. For our purposes, the file download primitive has three defining characteristics: (i) each client initiates transfer of an entire file, rather than a block or fragment, (ii) the client postpones interpretation of the data until the transfer completes, and (iii) the transfer may occur as rapidly as permitted by the server, the client, and the network resources. Whole-file transfer is a building block of peer-to-peer (P2P) file sharing, the Web, media services, and data grids [2].

Conventional file download protocols such as FTP or HTTP transfer each file as an ordered stream of bytes, and use the underlying transport protocol (e.g., TCP) to deliver data in sequence. Nevertheless, the semantics of file download permit the server to deliver data to each client in any order, as long as the client eventually receives the entire file. Reordering relies on the client to reassemble the content, similar to P2P “content swarming” systems that disperse file blocks across multiple servers (such as BitTorrent [14]).

This paper investigates *opportunistic* block reordering as a technique to improve the cache effectiveness of a content server. More effective caching can increase the server throughput for a given disk subsystem, and decrease the disk and memory cost needed to fill the server network link. Block reordering complements well-known techniques to improve cache effectiveness with improved replacement algorithms (e.g., [20]) or integrated caching and prefetching [10]. Most obviously, it changes the block access sequence to suit the needs of the storage system, instead of just managing the storage cache to suit a specific block access sequence. In essence, block reordering extends the server memory by using the client memory as a reassembly buffer.

Block reordering is helpful primarily when multiple clients request large shared files concurrently. Workload studies have shown this case to be quite common and important for overall performance. Typically a modest percentage of popular files receive most of the requests in a content server, and larger files account for a disproportionally large share of the data traffic [3, 5, 7, 11, 27]. A recent workload characterization of a popular P2P system concluded that 42% of the data requested from a typical academic site involved transfers of a few hundred large objects with an average size of several hundred megabytes each [27]. In that study, large object caching on its own could yield a byte hit ratio as high as 38%. Block reordering is especially promising in content networks that employ content-based request routing to concentrate the requests for each file on a small set of servers; recent studies show large improvements in server cache locality from this technique [16, 22, 30].

When files are small it is sufficient to cache them in their entirety to capture most of the benefit from sharing. But caching of large files is less effective because they consume more cache space and therefore they are more vulnerable to eviction before the next request can generate a cache hit. As file size increases relative to the server memory, the hit ratios degrade and the disk access transaction rate limits server throughput [6, 27]. Unfortunately, ongoing advances in disk bandwidth—due to in-

creasing areal density and faster rotation speeds—do not help appreciably. In fact, seek overheads dominate because concurrent requests from a large number of clients tend to destroy the inherent sequentiality of block accesses to each file, producing a block access sequence that is effectively random. Since seek times improve more slowly than disk bandwidth, faster disks actually make the problem worse because these seek overheads consume a larger share of the arm time.

Several other techniques are directed at serving large shared content files, primarily for continuous media. One solution is to abandon caching and use large disk transfers to reduce seeking, a technique that is well-studied for streaming video servers (e.g., [4]). But buffer demands increase linearly with transfer size and the number of clients. A second alternative is to encode the shared files using forward error correcting codes (FEC) [9, 26], e.g., the Tornado codes used by Digital Fountain. FEC yields optimal caching effectiveness, since every receiver can benefit from each block fetched from the disk. Unfortunately, FEC increases the volume of data needed to store and transmit a stream by as much as an order of magnitude, depending on the skew of client transfer rates. Other techniques include stream merging methods [17, 18]. Section 6 discusses related work in more detail.

The rest of this paper is organized as follows. Section 2 uses a simple model to explore the performance behavior of file download servers and motivate opportunistic block reordering. Section 3 proposes a simple parameterized block selection algorithm for reordering. Section 4 describes its implementation in the Circus content server, and Section 5 presents experimental results. Section 6 sets opportunistic block reordering in context with previous work, and Section 7 summarizes our conclusions.

2 Overview and Motivation

We first define the *file download problem* more formally, and then outline the goals and motivation for our work. Consider a content server with a network link of bandwidth R_n bytes/s and a population of N clients concurrently downloading shared files. We use the term *file* to mean a unique ordered set of data blocks; thus it could refer to any segment of a larger data set. Suppose that each client i can receive data at a sustained rate $R_c^i \leq R_n$, with an average client rate R_c bytes/s. The goal is to schedule block reads from disks and block transfers to clients in order to maximize $X(N)$, the system throughput for N concurrent requests, or the number of download requests completed per time unit. Unless it is saturated, the server should be fair and it should complete each request in the minimum time that the network allows: a request from client i to download a file of length L bytes should complete after approximate delay L/R_c^i.

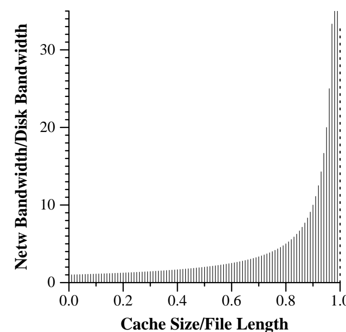

Figure 1: The maximum ratio of network bandwidth to disk bandwidth, as a function of the portion of the data set size that fits in the cache. The system is heavily disk-limited unless the bulk of the data set fits in memory.

Since this paper focuses on the design of the server and its storage system, we will claim success when the server is network-limited, i.e., it consumes all of its available network bandwidth. Suppose the average file length is L bytes. For low client loads, the server may be limited by the aggregate network bandwidth to its clients: $X(N) \leq R_c N/L$. If the client population is large, or if the clients have high-speed links, then the server may be limited by its own network link: $X(N) \leq R_n/L$ (with unicast). We assume without loss of generality that the server's CPU and memory system are able to sustain the peak data rate R_n when serving large files.

Our objective, then, is to minimize the memory and storage resources needed to feed content to the network at the peak rate R_n for a sufficiently large client population: $N \geq R_n/R_c$. Suppose the content server has a memory of size M and D disks delivering a peak bandwidth of R_d byte/s each. Our purpose is to explore how block reordering can reduce the number of disks D needed for a given M, and/or reduce the M needed for a given D and R_d. Equivalently, we may view the objective as maximizing the network speed R_n or client population size N that a given configuration (M, D) can support. The problem is uninteresting for small R_n, e.g., in serverless P2P systems that distribute server functions across many slow clients. The problem is most interesting for servers in data centers with high-capacity network links, serving large client populations with slow transfer rates R_c. N may be quite large for commercial content servers with a large bandwidth disparity relative to their clients: IP network bandwidth prices are dropping [13], while broadband deployments for the "last mile" to clients continue to lag behind. Although our approach does not require multicast support, it is compatible with multicast and N may be even larger when it exists.

2.1 Caching

The server's memory of size M consists of a common pool of shared buffers. A server uses these buffers for some combination of disk buffering, network buffering, and data caching. For example, suppose the server fetches data from storage in blocks of size B bytes. A typical server would buffer each fetched block until all client transmits of the block to clients have completed; the surplus memory acts as a cache over blocks recently fetched for some client i that are deemed likely to be needed soon for another client j requesting the same file. Caching is effective for small files that can reside in the cache in their entirety until the next request [6]. However, if j's request arrives t time units after i's, then the server has already fetched up to tR_c^i bytes of the file into memory and delivered them to i. The memory needed to cache the segment until j is ready to receive it grows with the inter-arrival time t. If $R_c^i > R_c^j$ then the required cache space continues to grow as the transfers progress—when the system is constrained to deliver the data in order to both clients, as for conventional file servers.

Since each file request accesses each block of the file exactly once, block accesses are uniformly distributed over the entire file length. As the number of clients increases, and as client rates and arrival times vary, the block access request stream shows less temporal locality and becomes effectively random. Of course, there is spatial locality when the server delivers each block in sequence; the system may exploit this by using a larger block size B, as discussed below (or, equivalently, by prefetching more deeply). Suppose without loss of generality that the content consists of a single file of length $L >> M$, or any number of equally popular files with aggregate size L. Then the probability of any block access being a cache hit is $P_{hit} = M/L$, and $P_{miss} = 1 - M/L$ is the probability that the access requires a disk fetch. Then the server's storage system must be capable of sustaining block fetches at rate $(R_n/B)(1 - M/L)$. Figure 1 shows the role of cache size in determining the number of disks required to sustain this rate.

2.2 Storage Throughput

The areal density of magnetic storage has been doubling every year, and disks today spin several times faster than ten years ago [13]. While these trends increase sequential disk bandwidth, throughput for random block accesses (IOPS) has not kept pace. Seek costs tend to dominate random accesses, and these costs are limited by mechanical movement and are not improving as rapidly. Unfortunately, the block miss stream for a content server degrades to random access as the client population N increases, for the same reasons outlined above.

One solution is to increase the block size B. This reduces the rate of disk operations required to sustain a given effective bandwidth, which is inversely proportional to B. This technique can significantly reduce the number of disks required. Suppose each disk has an average head positioning time per access of T_{pos} seconds (seek plus half-rotation). Every disk access moves a block of size B bytes, and takes time $T_{pos} + B/R_d$. Thus, each disk supports $\frac{1}{T_{pos}+B/R_d}$ random accesses per time unit, and the aggregate peak disk bandwidth for D disks becomes $X_d = \frac{B \times D}{T_{pos}+B/R_d}$ bytes/s. The server needs $D = \frac{R_n}{B/(T_{pos}+B/R_d)}$ disks to fully pipeline the network. For example, Figure 2 illustrates the disk random access throughput when $T_{pos} = 0.005$ s, a typical value. With block size $B = 256$ KB, and disks of $R_d = 50$ MB/s, the disk throughput X_d becomes roughly 25 MB/s. We need about $D = 5$ such disks to feed a server network link of $R_n = 1$ Gb/s.

However, Figure 3 shows that this approach consumes large amounts of memory with large client populations. Depending on the buffering scheme the minimum memory size M for N active clients is $NB/2 \leq M \leq 2NB$. The 1 Gb/s server can support $N = 647$ T1 (1.544 Mbps = 193 KB/s) client connections, consuming a minimum $M = 81$ MB just for device buffering with $B = 256$ KB. If we increase the block size to $B = 1$ MB then disk throughput becomes $X_d = 40$ MB/s, the number of disks drops to $D = 4$, and the minimum memory grows to $M = 324$ MB. If instead, we assume slower clients with $R_c = 56$ Kbps, the server needs a minimum $M = 2.18$ GB and $M = 8.7$ GB for block size $B = 256$ KB and $B = 1$ MB, respectively. In media servers using large block sizes it is common to eliminate the cache and partition memory into separate buffer regions for each client; each region is sufficient to hold any blocks in transit between the disk and the network for that client [4].

2.3 Summary

In this section we explained why accesses at the block level appear increasingly random as the client population N grows. This minimizes the benefit of conventional caching and weakens the locality of the disk accesses from the cache miss stream. The combination of these effects increases server cost, since more disks and/or more memory are needed to fill any given network link. For example, the experiments in this paper use disks with average sequential throughput of $R_d = 33$ MByte/s, but a conventional FTP server under even modest load delivers a per-disk throughput of roughly 10 MB/s, including cache hits. Thus we need about a dozen such disks to feed a network link of 1 Gb/s.

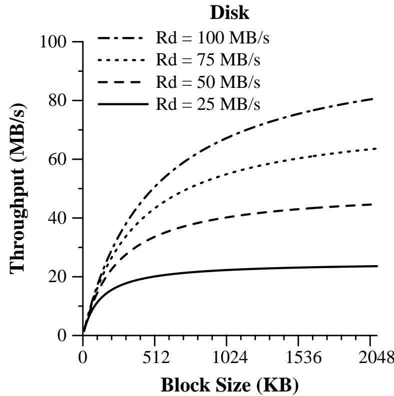

Figure 2: Disk throughput with random block accesses in a disk with positioning time $T_{pos} = 0.005$s across different transfer block sizes, and sequential disk transfer rates R_d. We show that as R_d increases, block size must exceed one megabyte to achieve peak disk throughput.

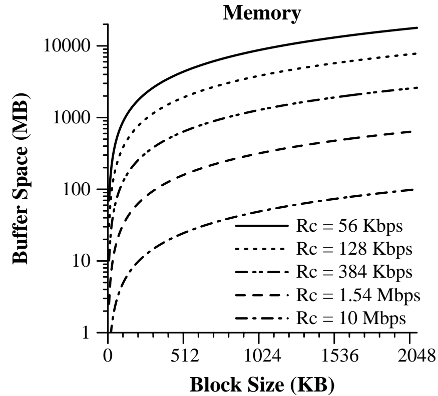

Figure 3: Minimum buffer space required to support clients at different rates in a server with a 1 Gbps network link. The minimum buffer space can reach several gigabytes for large populations of clients with bandwidth less than 1 Mbps.

3 Opportunistic Block Reordering

To support large client populations inexpensively, we need to minimize the number of disk fetches and maximize the number of clients served with each fetch. Ideally, when multiple clients request the same file, the server retrieves each block from disk only once and sends it to every client. Then, using the notation introduced previously, serving a file to N clients at network throughput NR_c requires disk throughput R_c and buffer space $B/2 \leq M \leq 2B$, all of which are independent of N.

A file download server must transfer all the blocks of a file to satisfy each file request; thus an arriving file request from client i reveals information about block accesses on behalf of client i for at least the next L/R_c^i time units. The system can use these revealed block sequences to improve performance and/or reduce cost. In particular, it can improve cache performance by opportunistically reordering the accesses, e.g., to send each fetched block opportunistically to all clients known to need the block in the future, before replacing it from memory. The server can send blocks out-of-order by attaching an application-layer header [12] to each transmitted block that specifies its offset; clients resequence the data according to these block headers. This section presents the block reordering algorithm in the *Circus* content server.

The *Circus* server maintains a list of files with downloads currently in progress (*active files*), and for each active file a list of clients currently downloading that file (*active clients*). For each active client the server creates a FIFO queue of references to the file blocks selected to send next to that client. The server transmits blocks from the head of the queue when the client's network transport send window opens. When a queue drains below a threshold, the server selects another block to transmit to that client, schedules a disk fetch if necessary, and adds the block reference to the queue tail. *Circus* strives to guarantee forward progress and fairness by serving all active clients at their maximum network transfer rates.

The block selection policy is the key to the *Circus* algorithm. For each active client, *Circus* also maintains a set data structure (bitmap) to record the set of blocks already sent to the client. Thus, for each block of an active file, it knows the set of active clients that have yet to receive the block. The selection policy could select blocks to greedily maximize the number of clients that benefit from each block, or minimize the number of block fetches needed to fill up all block queues at any given time. However, these policies are computationally complex, and their cost scales with the file size.

We propose a simple block selection policy that is fast and has modest memory requirements. Figure 5 outlines the algorithm. Its complexity scales linearly with the number of active clients for each file. To preserve locality of reference among the active clients, *Circus* designates an (*active region*) for each file as preferred for block selection and caching. The active region is a moving contiguous region within a circular block space. The *file cursor* specifies the front edge of the active region of a file. The length of a file's active region (the *active length*, a tunable configuration parameter) controls the amount of memory devoted to caching the file.

Each active client has its own *client cursor* that keeps track of the file offset of the block that was last queued to send to the client. Based on the client cursor positions, we call the leading active client for a file (often the highest-rate client) the *frontrunner* of the file; the trailing clients active on the same file are the *followers*. The frontrunner advances the file cursor according to its current client cursor. This operation refreshes the system cache with new data blocks that are likely also needed by the followers. When the cursor of a follower lags behind the

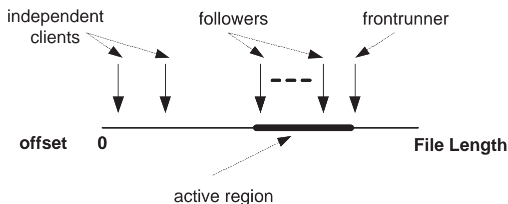

Figure 4: We call the *active region* of a file the part of the file that is resident in server memory. The client who downloads the file with highest rate (*frontrunner*) advances the active region, while clients with lower rates (*followers*) are kept within the active region. Some clients (*independent*) are allowed to move outside the active region.

```
1.  proc circus_algorithm
2.  while (true) do
3.   file := next entry of active_file_circular_list
4.   while (true) do
5.     client := next entry of file.active_client_list
6.     if (client = nil) break
7.     if (client.fifo_queue = full) continue
8.     if (file.cursor-client.cursor > active_length AND
9.       client.sent_blocks_fraction < client.threshold)
10.        client.cursor = file.cursor - active_length/2
11.    client.cursor := next block not transmitted
12.    if (client.cursor = invalid) continue
13.    insert client.cursor location to client.fifo_queue
14.    if (client.cursor > file.cursor AND
15.      client.cursor-file.cursor < leapfront_distance)
16.        file.cursor = client.cursor
17.   done
18. done
```

Figure 5: Pseudocode of the *Circus* block selection algorithm.

frontrunner at a distance that exceeds the active length, the follower's cursor advances to the middle of the active region. Moving the cursor to the middle rather than the front of the active region prevents the follower from immediately becoming a frontrunner. We can summarize these operations with the following two rules:

- Advance the file cursor to the current cursor location of the fastest client (*frontrunner*).
- When the cursor of a trailing client (*follower*) falls behind the *active region*, advance the cursor of the *follower* to the midpoint of the active region.

To avoid stalling clients that are missing only a small number of blocks, *Circus* tracks the fraction of file blocks each client has already received (the *client threshold*). When this fraction exceeds a configured maximum value, the client becomes *independent*; the algorithm selects blocks for independent clients by scanning their block maps for needed blocks, even if they are outside of the active region. A client also becomes independent if it is the frontrunner and its next missing block is more than a configured *leapfront distance* ahead of the current file cursor. This avoids damage to the reference locality of the other active clients. In our experiments, we found a leapfront distance equal to the active length of the file to achieve stable operation in the system. We investigate the sensitivity of the system to several configuration parameters in Section 5.5.

4 Evaluating Circus

This section gives an overview of our methodology for evaluating *Circus*, including the performance metrics of our experiments.

4.1 Prototype Implementation

We incorporated the *Circus* algorithm into a version of the FTP daemon of FreeBSD Release 4.5 (Figure 6). We modified both the FTP client and server to support block reordering using a simple block transfer protocol with sequencing headers.

The current version of the *Circus* server is a fully functional implementation that required no kernel modifications. For each active file the server constructs a header structure (*file header*) containing information about the file. This includes a linked list of headers (*client headers*) for the active clients of each file, and a block FIFO queue of limited length for each active client. The *block size* is a configurable parameter that serves as logical unit of disk and network transfers. When the socket buffer for a client is ready to send, a server process removes a block descriptor from the queue, and submits a request to transfer the block to the network socket. We used the *sendfile()* system call—available in FreeBSD and other operating system kernels—for zero-copy transfers from the disk to the network. A background process runs the block selection algorithm to refill the FIFO queues with descriptors of blocks to transmit to each client. The headers and the queues are maintained in shared memory accessible by all server processes.

The modified FTP client reassembles downloaded files and optionally writes them to the local file system on the client. We disabled the writes in our experiments so that multiple clients (up to a few hundred) could run on each test machine.

4.2 Metrics and Workload

We define *disk throughput* as the total disk bandwidth (byte/s) used by the server, and *network throughput* as the total network bandwidth (byte/s) used to send data over the network to clients (with unicast). We can approximate the *miss ratio* from the ratio of disk throughput

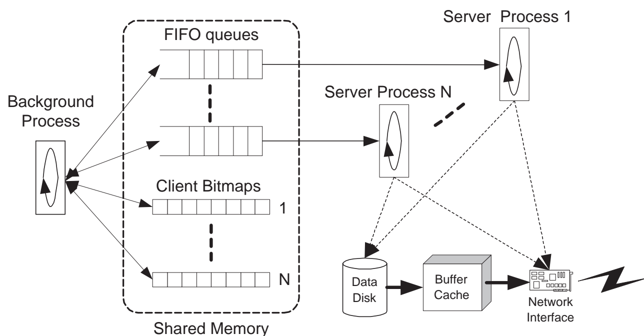

Figure 6: Simplified diagram outlining the prototype implementation of the *Circus* download server. Solid arrows show data transfers, while dotted arrows illustrate transfer of control.

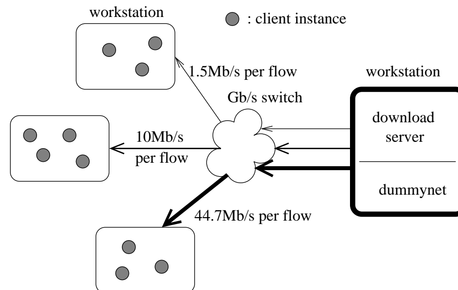

Figure 7: Multiple client instances corresponding to different download requests are partitioned across several workstations according to their network link capacity. The server is connected to the client workstations through a gigabit Ethernet switch. *Dummynet* is used to control the per flow rate from the server to each client workstation.

to the network throughput, ignoring locality effects. A low ratio indicates that disk block fetches are contributing to multiple outstanding transfers for a shared file. The reduced disk activity can improve *download time* and server throughput (the download completion rate).

We assume Poisson arrivals for the download requests because they closely match real workloads studied in previous work [3]. For the definition of the system load ρ during steady-state operation, we take the delivered network throughput as the aggregate transfer rate requested by the users; ideally, each download request should be served at the client's network link rate. Based on the analysis of Section 2, we consider the server's network bandwidth to be the limiting resource: maximum system load $\rho = 1$ occurs when the arrival rate generates network throughput that saturates the server's network link. Depending on the file sizes, this load definition leads to different request arrival rates and different interarrival gaps. We derive the mean request interarrival time corresponding to the maximum system load from the ratio of the average file size over the outgoing server link capacity. For lower loads, the interarrival gap is the ratio of the peak load interarrival time to the load level ρ.

4.3 Measurement Environment

All experiments use Intel PIII-based systems with 256 MB main memory running FreeBSD 4.5 at 733 MHz or 866 MHz. A group of client workstations run multiple client instances to generate request loads to a server. On the server node, we use *Dummynet* [25] to specify the per flow transfer rate from the server node to each client (Figure 7). We store the file data in a 18GB Seagate Cheetah 10K RPM disk with sequential transfer rate 26-40MByte/s. The systems are equipped with both 100 Mbit/s and 1 Gbit/s Ethernet interfaces connected via two separate switches. File network transfers take place over the gigabit switch using jumbo Ethernet packets (9000 bytes) to reduce network protocol overhead.

We focus on handling download requests of files with total storage footprint that exceeds the memory of the server. Most of our experiments use files of size 512MB. Our results also apply when files fit in server memory but the aggregate footprint exceeds server memory. Due to reported correlations between the transferred file size and the client link capacity [31], and to reduce experimentation time, we conservatively consider clients with broadband transfer rates. For each different client link capacity that we support, we dedicate a separate client node and configure its connection speeds to the server through *Dummynet*.

From recent studies in peer-to-peer network systems it has been found that 20-30% of the users have downstream network links less than 1Mbit/s, about 80-90% have downstream links less than 10Mbit/s, and the remaining 10-20% have links that exceed 10Mbit/s [28]. In accordance with the above results and the fact that broadband user population tends to increase over time, we specified three groups of clients with 1.544 Mbit/s (T1), 10 Mbit/s (we call it 10T), and 44.736 Mbit/s (T3). We experiment with each of these groups separately, and also with a mixed workload (Mx) where 20% of the users are of type T1, 60% are of type 10T, and the remaining 20% are type T3. In several cases, we focus on 50%-50% mixes of two client groups. We allow each experiment to run for between $\frac{1}{2}$ hour and $1\frac{1}{2}$ hour, depending on the network link capacity of the clients. Measurements start after one or more initial download requests complete. Each experiment is repeated until the half-length of the 95% confidence interval of the measured network throughput lies within 5% of the estimated mean value across different trials.

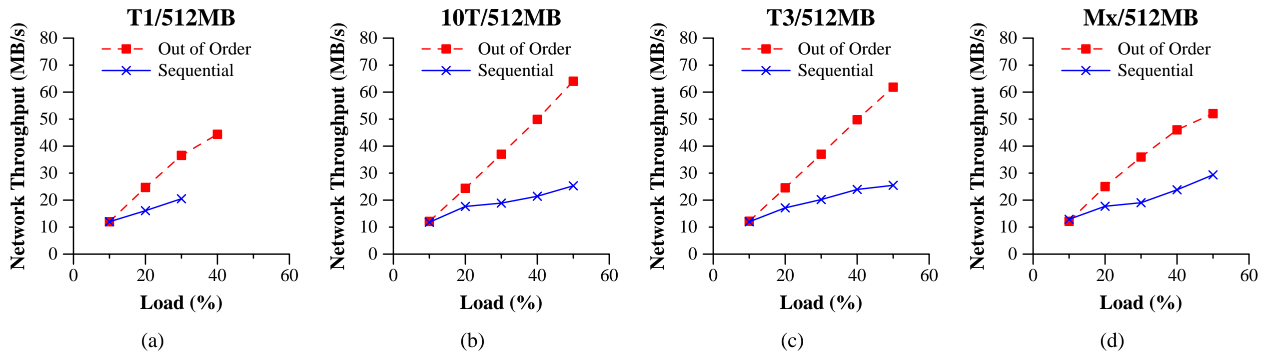

Figure 8: We compare the network throughput of the unmodified (sequential) and *Circus* (out-of-order) download server implementations at increasing system loads using 512MB file requests. We consider the client link capacity to be equal to (a) 1.5Mbit/s, (b) 10Mbit/s, (c) 44.7Mbit/s, and (d) a mix of 20% 1.5Mbit/s, 60% 10Mbit/s and 20% 44.7Mbit/s. The out-of-order approach more than doubles the network throughput at higher loads. The higher the network throughput, the better the system throughput as well.

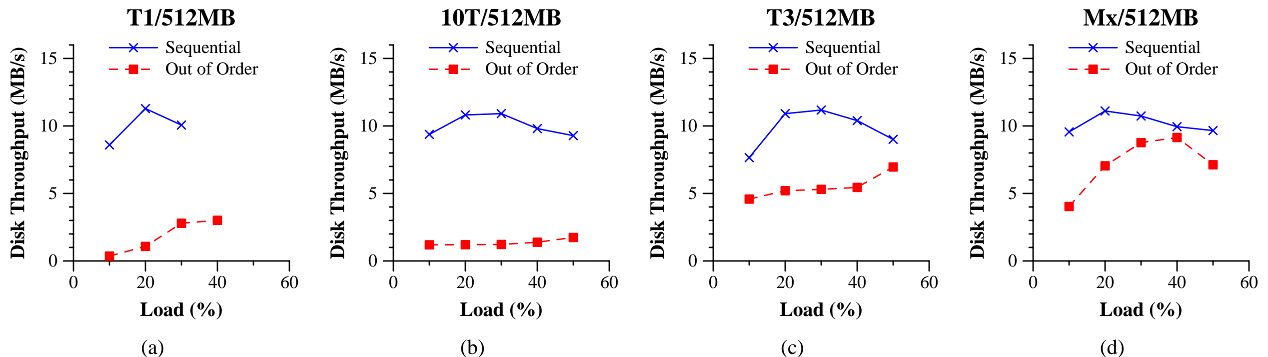

Figure 9: At low and moderate loads, the disk throughput with out-of-order transfers remains roughly equal to the transfer rate of the most demanding individual client across different client link rates (a-d). With sequential transfers, the disk is highly utilized and becomes a bottleneck as shown in Figure 8 (lower disk throughput is better for a given network throughput).

5 Experimental Results

We compare the performance of the *Circus* prototype with an unmodified FreeBSD 4.5 ftpd implementation. The experiments investigate alternative client features, network conditions, file characteristics, and server configuration parameters. We find *Circus* to improve the server throughput and file download time when files are shared by multiple clients.

5.1 Client Link Capacity

A key challenge in the design of a download server is to adapt automatically to different client rates without manual tuning. The closer the transfer rates of two clients match, the easier it becomes to exploit the data sharing among them. As the difference increases, it becomes more difficult to share cached data effectively.

Figure 8 depicts the network throughput of an unmodified (sequential) and a *Circus* (out-of-order) server as clients with different rates download a single file of size 512MB. In typical ftpd implementations (including the one that we use here), each active download request spawns an extra server process with resident memory space of about 1MB. Consequently, we show only T1 measurements for up to 30-40% load, roughly corresponding to about 200 concurrent clients. Beyond this point memory paging interferes with the measurements.

In all the three cases of a single client link rate (a-c), the out-of-order network throughput increases proportionally with the system load. In particular, at 40% load, we expect to receive 51.2MByte/s throughput, which is roughly what we observe in cases (b) and (c). The measured throughput is somewhat lower (in case d) with clients of different rates on the same server, but still reaches 50MByte/s at 50% load. Quite remarkably, the sequential system only matches the out-of-order performance at 10% load in the four cases, and never exceeds 30MByte/s (on average) as the load increases.

Figure 9 shows disk throughput for the same experiment. With sequential transfers, the disk is highly utilized even at low loads, regardless of the client rates. In contrast, with out-of-order transfers (a-c) the disk throughput drops to the transfer rate of a single client. For example the disk throughput is about 1MByte/s with 10T transfers (b), an order of magnitude lower than the se-

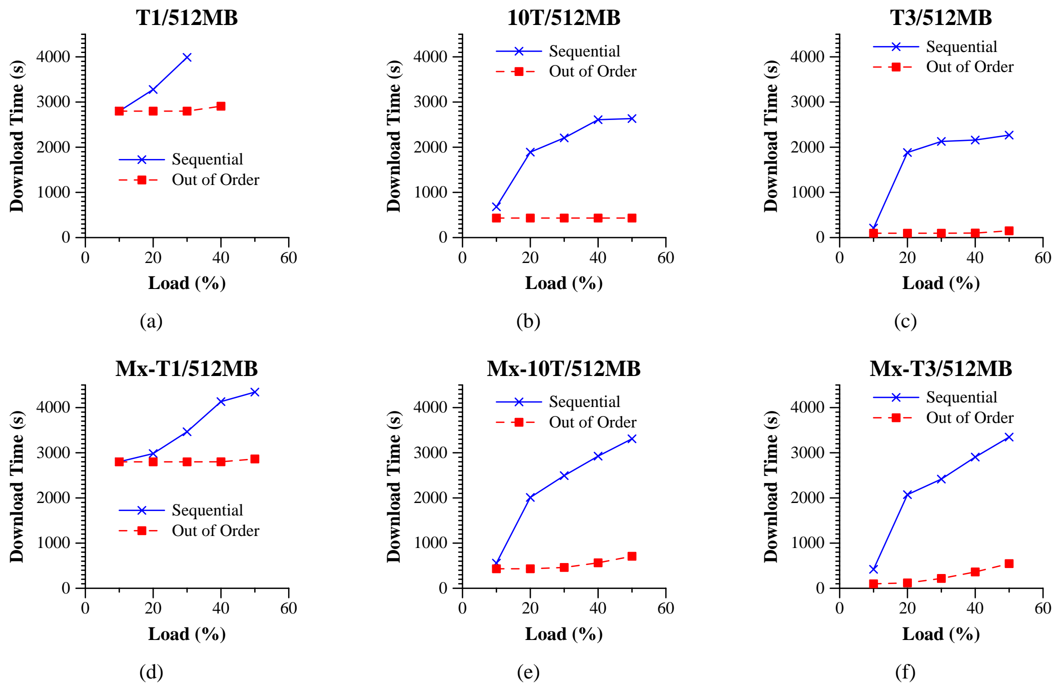

Figure 10: With out-of-order transfers, download requests take almost constant time to complete as the load increases. When file data transfers are served sequentially, however, the download duration increases significantly as a function of the system load. We consider the case where all clients have the same network link capacity (a-c), and when clients of different link capacities are served by a single server (d-f).

quential case. When we mix clients of different capacities (d), this behavior holds at low loads with the disk throughput about 5.6 MByte/s. At higher loads, the proportion of non-sharing (independent) clients increases, raising the disk throughput accordingly. Figure 10 further verifies these observations. With out-of-order transfers, the download latencies remain roughly constant at different system loads, according to the client rates. But when sequential transfers are used, the download latency increases rapidly with the system load.

5.2 Transferred File Size

This section investigates how the file size affects the system performance. Figure 11(a) shows the server network throughput in a file size range between 256MB and 1GB. We observe that, with out-of-order transfers, the network throughput remains above 50MByte/s, consistent with the 40% offered load. Sequential transfers cause the network throughput to drop below 20MByte/s, approaching the disk throughput. As a result, download latency (not shown) increases dramatically for sequential transfers to several tens of minutes. For the out-of-order case all downloads complete within a few minutes at all the file sizes that we examined.

5.3 Multiple Files

Even though it is likely that only a few files will be in heavy demand by the clients, we investigate how the performance of the system is affected when the number of popular files increases. We consider 1 to 16 different files of 512 MB each, all stored on a single server disk, and requested with equal probability. The clients receive data over 10Mbit/s links, and the system is at 40% load. In Figure 12(a), we illustrate the network throughput of the server with sequential and out-of-order transfers respectively. In the out-of-order case, the measured throughput remains roughly 50 MByte/s with up to 8 files, and drops slightly to 48MB/s with 16 files. From Figure 12(b), the average disk throughput increases linearly with the number of files up to eight, and reaches 10MB/s at 16 files. This behavior is expected because the number of disk access streams increases with more active files, and the disk throughput begins to limit the system as it approaches 10MByte/s. With sequential transfers, the disk throughput always limits the system and performance only worsens as the number of files increases.

5.4 Round-trip Delay and Packet Loss

Packet loss rate and propagation delay can vary significantly in a wide-area network depending on the physical span and the operating conditions of the network. We in-

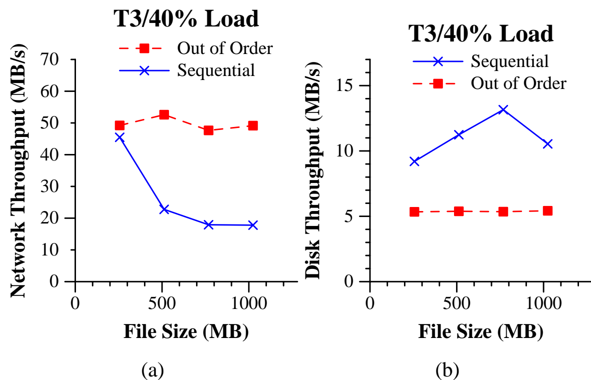

Figure 11: Server network throughput and disk throughput for transfers along T3 links of a file with size between 256MB and 1GB. The system load is set equal to 40%. With out-of-order transfers the the network throughput is always higher and the disk throughput is constant regardless of the size of the file.

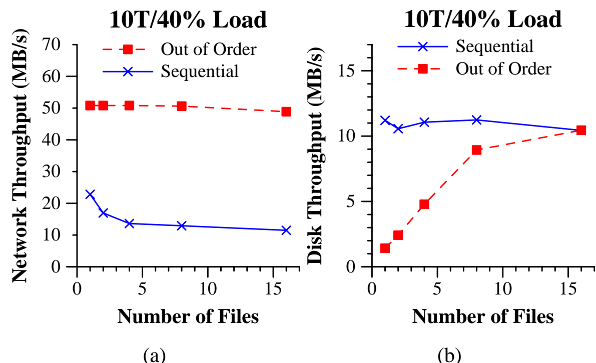

Figure 12: Server network throughput and disk throughput when the number of concurrently requested files varies from 1 to 16. The system load is 40% and the file size is always 512MB. All files are requested with equal probability.

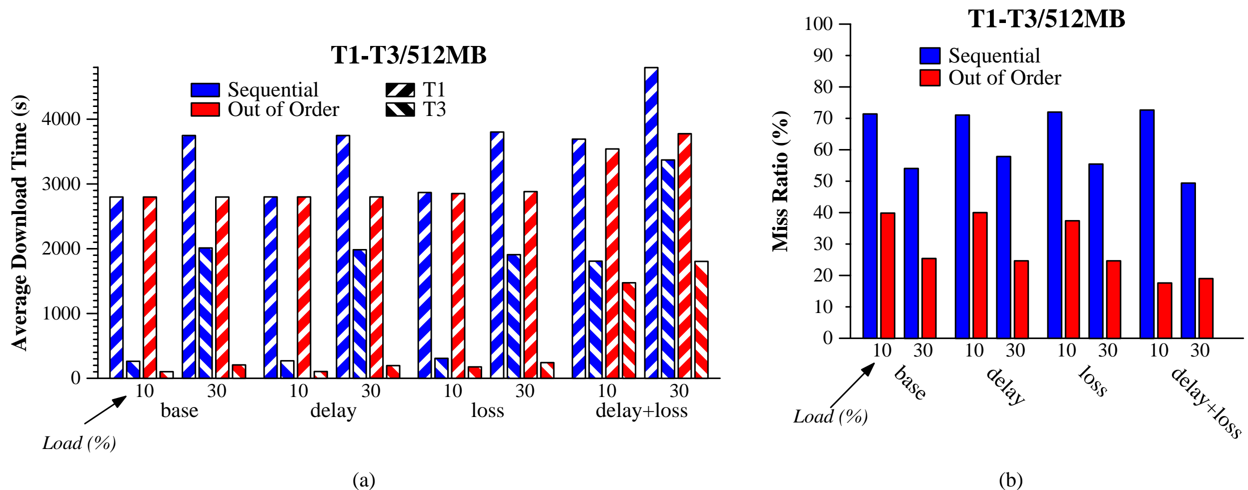

Figure 13: In the base case we assume round-trip delay less than 1ms and packet loss close to 0%. In the delay case we increase the round-trip delay to 75ms and in the loss case we increase the loss rate to 10%, correspondingly, with respect to the base case. Both the download time and the miss ratio of sequential and out-of-order transfers can be affected when combining round-trip delay of 75ms with packet loss rate of 10% (delay+loss). T1 and T3 links are used with equal probability to connect a single server to multiple clients.

vestigated the impact of such factors to file transfers by experimenting with round-trip times of about 1 and 75 ms, and with packet loss rates about 0% and 10%, respectively, using Dummynet. In Figure 13, we measure the download time and server miss ratio when transferring a 512MB file over T1 and T3 links from the same server. When packet loss of 10% and delay of 75ms are combined in out-of-order transfers, download time over T3 links increases by an order of magnitude approaching the level of sequential transfers. This ten-fold increase from the base case can be attributed to the mechanism used by the congestion avoidance algorithm to recover the congestion window at the sender.

Longer round-trip delays increase the recovery time and the wasted network bandwidth. This can be explained by the TCP operation: packet losses lead to triple duplicate acknowledgments (rather than timeouts), and the congestion window increases by at most one data segment every round-trip time [21]. Individual sequential transfers have low throughput due to the disk bottleneck, and are not affected further at low load. However, raising the system load from 10% to 30% doubles the time of T3 sequential transfers, while leaving the out-of-order transfer time almost unchanged. When combining delay and loss with out-of-order transfers, disk throughput drops because data retransmissions hit in the buffer cache. We don't observe similar effects for sequential transfers, which provides additional evidence about the poor disk access locality of this policy.

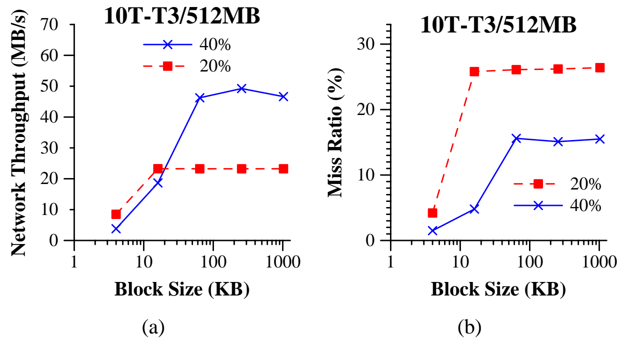

Figure 14: We examine the sensitivity of the system to the block size parameter, when mixing equiprobable requests from clients with 10T and T3 links requesting a file of 512MB at system load 20% and 40%. Both the network throughput and the miss ratio are adversely affected by block sizes smaller than 64KB, but remain insensitive to larger values.

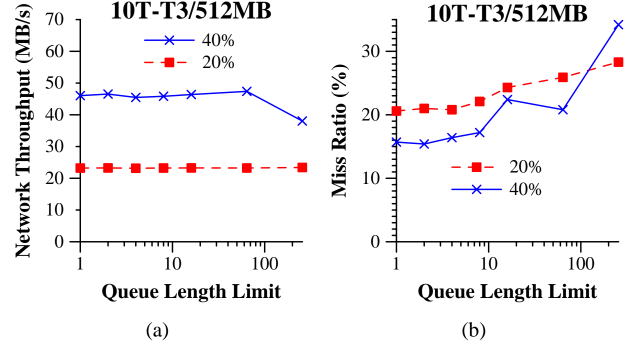

Figure 15: We investigate the effect of the queue length limit, when mixing equiprobable requests from clients with 10T and T3 links for a 512MB file at different loads. As the queue length limit increases, the disk throughput also grows leading to higher miss ratio. Eventually, the disk bandwidth becomes bottleneck which makes the server network throughput to drop.

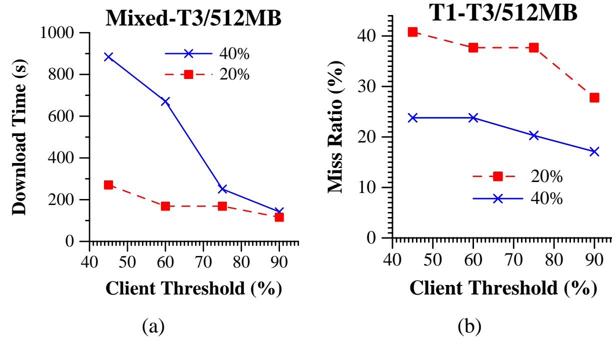

Figure 16: We examine the sensitivity of the system performance to the client threshold when mixing download requests over T1 and T3 network links. We found the client thresholds equal to 75% or higher to keep both the download time over T3 links and the miss ratio low.

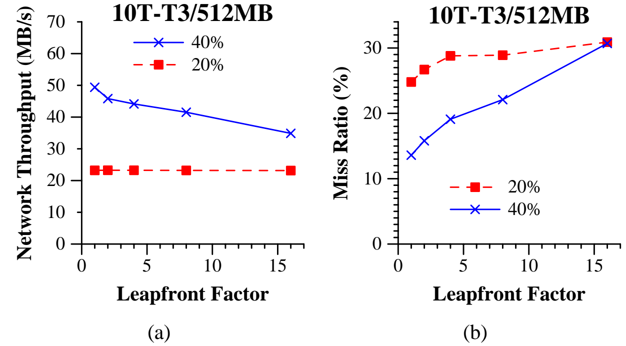

Figure 17: We show the effect of the leapfront factor using equiprobable download requests over 10T and T3 lines at different system loads. As the leapfront factor increases, network throughput drops and miss ratio surges, especially at high system load.

5.5 Sensitivity to System Parameters

This section examines how sensitive the system behavior is to important configuration parameters. We did extensive experimentats to ensure that the system remains robust across a wide range of workloads, but we include only a few representative measurements here. Overall, the system behavior is affected by the configuration parameters below, but remains stable when the parameters remain within the ranges that we suggest.

Block Size. The *block size* is a configurable parameter that specifies the unit of disk access and network transfer requests in the server. Its value affects the utilization of the devices, the overhead involved in the operation of the system, and the overall server throughput. In Figure 14, we illustrate the network throughput and miss ratio across different system loads for block sizes ranging between 4KB and 1MB. We observe that both the measured metrics remain constant with block size larger than 16KB and 64KB at low and high load, respectively. Low loads show higher miss ratios because there is less sharing. Smaller block sizes increase the disk access overhead and block selection overhead. In general, we found the block size equal to 64KB to perform well, and we used it in all the other experiments.

Queue Length Bound. Figure 15 shows the effect of varying the bounds on the block queue FIFOs for active clients. Shorter queues make the system more adaptive to the variability of the client behavior, because the blocks sent to each client are chosen based on recent system conditions. On the other hand, large queue lengths can increase the throughput of the system by keeping each client's network path fully pipelined. We examine the impact of the *queue length limit* on the performance of the system using 512MB download requests over equiprobable 10T and T3 links. With longer queues the miss ratio increases, the disk bandwidth becomes a bottleneck, and the server network throughput drops. This is expected because longer queue lengths can lead to stale requests for blocks that have been evicted from the cache

and incur extra disk activity. In all the other experiments, the queue length limit is set equal to 5.

Client Threshold. The *client threshold* controls the creation of independent clients according to the percentage of file blocks the client has received. From our experiments we found the system to perform well with client threshold around 0.75. Lower client thresholds reduce data sharing, increase disk access activity and lead to longer download duration (Figure 16), while higher client thresholds make the system operation less stable especially with large number of clients.

Leapfront Distance. The *leapfront distance* determines when a client is allowed to play the role of a frontrunner depending on how far ahead from the file cursor the client cursor has moved. For convenience, we introduce the *leapfront factor* as the ratio of the *leapfront distance* over the *active length*. In Figure 17, we notice that as the leapfront factor grows larger than 1, the network throughput drops and the miss ratio increases. Setting the leapfront distance equal to the *active length* gives good performance by allowing the active region to move smoothly forward; larger leapfront distances tend to reduce spatial locality among different clients and lead to lower throughput. The active length was set equal to 16MB throughout our study.

6 Related Work

File Transfer and File Sharing. FTP and HTTP transfer objects sequentially, relying on the TCP transport to preserve byte ordering. With marker blocks [23] it is possible to restart a transmission after a failure. Raman et al. improve the interactive transfer of images over the Internet by delivering data to the client as they arrive, weakening the in-order abstraction of TCP [24]. Diot and Gagnon examine benefits of out-of-sequence packet processing [15], but do not consider large file delivery or interactions with storage devices.

Many wide-area storage systems allow a client to download different parts of a file from multiple servers (e.g., BitTorrent [14]); these clients resequence the data to tolerate out-of-order delivery. Acharya et al. propose a server architecture for repetitive transmission of data over a broadcast channel [1]. The frequency of transmitted data is determined by data popularity across the served client population.

Forward Error Correction. Digital Fountain [9] encodes content with forward error correcting (FEC) codes (e.g., Tornado codes) for distribution over a multicast network. FEC allows a client to reconstruct a file once it has received a minimum number of distinct blocks. This approach eliminates the need for acknowledgments in a multicast setting. The system can be extended to transport large files to a client from multiple collaborating sources in overlay networks [8].

In a unicast network, FEC encoding can be applied to improve caching efficiency at the server [26]. Since the client can reconstruct the data from any sufficiently large subset of the encoded blocks, a block fetched from disk may be useful to multiple clients with different request arrival times and different rates. If a block is lost, another may be sent in its place, avoiding the need for the server to buffer data for retransmission. However, duplicate blocks waste client bandwidth; in a typical heterogeneous environment, where client receiving rates can differ by several orders of magnitude, the encoded version of the transmitted file is much larger than the original to limit the probability that any arbitrary block is a duplicate for some active client [9, 26]. Recent theoretical work begins to address this problem [19]; if a satisfactory solution is found, then FEC could meet our objectives for downloading large files with a high degree of sharing with low network impact, e.g., when multicasting is available.

Circus demonstrates a technique with similar goals for content distribution: to maximize the advantage of data sharing across concurrent requests, while allowing clients at different rates to reassemble the requested file quickly and efficiently. However, *Circus* does not use FEC codes, and it is effective for unicast, although it could also benefit from multicast.

Stream Merging Methods. A class of *merging methods* for multicast delivery of streaming media allows a client to receive data transmitted concurrently to other clients [17, 18]. These file segmentation schemes balance the server network throughput, the client network throughput, and the playback initiation latency. Those schemes are significantly different from *Circus* because they have been specifically designed to support real-time delivery guarantees over reliable multicast-enabled networks assuming a fixed receiving rate for each client. In contrast, *Circus* supports efficient file (or file segment) download transfers over unicast networks for clients with different rates, and exploits the complementary technique of block reordering.

The insights underlying our approach are related to Steere's work with asynchronous set iterators [29], although our approach does not affect the order in which data is delivered to a client application.

7 Conclusions

This paper explores opportunistic block reordering to exploit the data sharing among concurrent file transfers. We introduce the *Circus* algorithm for scheduling disk access

and reordered network transfers, and evaluate an implementation in a modified FTP file server and client under synthetic file access workloads. We conclude that block reordering can significantly improve server cache performance for large, shared files. The average file download time with *Circus* remains close to minimum across the workloads, and is significantly lower than with conventional sequential file download. Additionally, *Circus* more than doubles the server network throughput when there is significant sharing, and reduces the required disk bandwidth by an order of magnitude in some cases.

8 Acknowledgments

We thank Balachander Krishnamurthy for an early discussion, and Amin Vahdat for helpful comments on a draft. Darrell Anderson, Wei Jin, and Ken Yocum also provided useful feedback. Miriam O' Mahony and David Becker offered valuable technical support.

References

[1] Acharya, S., Franklin, M., and Zdonik, S. Balancing Push and Pull for Data Broadcast. In *ACM SIGMOD* (Tuscon, AZ, May 1997), pp. 183–194.

[2] Allcock, B., Bester, J., Bresnahan, J., Chervenak, A. L., Foster, I., Kesselman, C., Meder, S., Nefedova, V., Quesnal, D., and Tuecke, S. Data Management and Transfer in High Performance Computational Grid Environments. *Parallel Computing Journal* **28**, 5 (May 2002), 749–771.

[3] Almeida, J. M., Krueger, J., Eager, D. L., and Vernon, M. K. Analysis of Educational Media Server Workloads. In *Intl Workshop on Network and Operating System Support for Digital Audio and Video* (Port Jefferson, NY, June 2001), pp. 21–30.

[4] Anastasiadis, S. V., Sevcik, K. C., and Stumm, M. Modular and Efficient Resource Management in the Exedra Media Server. In *USENIX Symposium on Internet Technologies and Systems* (San Francisco, CA, Mar. 2001), pp. 25–36.

[5] Arlitt, M. F., and Williamson, C. L. Web server workload characterization: the search for invariants. In *ACM SIGMETRICS* (Philadelphia, PA, May 1996), pp. 126–137.

[6] Baker, M. G., Hartman, J. H., Kupfer, M. D., Shirriff, K. W., and Ousterhout, J. K. Measurements of a distributed file system. In *ACM Symposium on Operating Systems Principles* (Oct. 1991), pp. 198–212.

[7] Barford, P., and Crovella, M. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *ACM SIGMETRICS* (Madison, WI, July 1998), pp. 151–160.

[8] Byers, J., Considine, J., Mitzenmacher, M., and Rost, S. Informed Content Delivery Across Adaptive Overlay Networks. In *ACM SIGCOMM* (Pittsburgh, PA, Aug. 2002), pp. 47–60.

[9] Byers, J. W., Luby, M., Mitzenmacher, M., and Rege, A. A Digital Fountain Approach to Reliable Distribution of Bulk Data. In *ACM SIGCOMM* (Vancouver, BC, Sept. 1998), pp. 57–67.

[10] Cao, P., Felten, E. W., Karlin, A., and Li, K. A Study of Integrated Prefetching and Caching Strategies. In *SIGMETRICS/Peformance '95* (May 1995).

[11] Chesire, M., Wolman, A., Voelker, G. M., and Levy, H. M. Measurement and Analysis of a Streaming-Media Workload. In *USENIX Symposium on Internet Technologies and Systems* (San Francisco, CA, Mar. 2001), pp. 1–12.

[12] Clark, D. D., and Tennenhouse, D. L. Architectural Considerations for a New Generation of Protocols. In *ACM SIGCOMM* (Philadelphia, PA, Sept. 1990), pp. 200–208.

[13] Coffman, K., and Odlyzko, A. M. Internet growth: Is there a "Moore's Law" for data traffic? In *Handbook of Massive Data Sets*. Kluwer Academic, 2002, pp. 47–93.

[14] Cohen, B. Incentives Build Robustness in Bittorrent, May 2003. bitconjurer.org.

[15] Diot, C., and Gagnon, F. Impact of out-of-sequence processing on the performance of data transmission. Computer Networks, 31 (1999), 475–492.

[16] Doyle, R. P., Chase, J. S., Gadde, S., and Vahdat, A. M. The Trickle-Down Effect: Web Caching and Server Request Distribut ion. In *Intl Workshop on Web Caching and Content Delivery* (June 2001).

[17] Eager, D., Vernon, M., and Zahorjan, J. Minimizing Bandwidth Requirements for On-Demand Data Delivery. *IEEE Transactions on Knowledge and Data Engineering* **13**, 5 (September/October 2001), 742–757.

[18] Jin, S., and Bestavros, A. Scalability of Multicast Delivery for Non-sequential Streaming Access. In *ACM SIGMETRICS* (Marina Del Rey, CA, June 2002), pp. 97–107.

[19] Luby, M. LT Codes. In *IEEE Symposium on Foundations of Computer Science* (Vancouver, BC, Nov. 2002), pp. 271–282.

[20] Megiddo, N., and Modha, D. S. ARC: A Self-tuning, Low Overhead Replacement Cache. In *Proc. of the 2nd USENIX Conference on File and Storage Technologies (FAST 03)* (March 2003).

[21] Padhye, J., Firoiu, V., Towsley, D. F., and Kurose, J. F. Modeling TCP Reno Performance: A Simple Model and Its Empirical Validation. *IEEE/ACM Transactions on Networking* **8**, 2 (Apr. 2000), 133–145.

[22] Pai, V. S., Aron, M., Banga, G., Svendsen, M., Druschel, P., Zwaenepoel, W., and Nahum, E. Locality-aware Request Distribution in Cluster-based Network Servers. In *ACM ASPLOS* (San Jose, CA, Oct. 1998), pp. 205–216.

[23] Postel, J., and Reynolds, J. File Transfer Protocol (FTP), Oct. 1985. USC/ISI, Network Working Group RFC 959.

[24] Raman, S., Balakrishnan, H., and Srinivasan, M. An Image Transport Protocol for the Internet. In *Intl Conf. on Network Protocols* (Osaka, Japan, Nov. 2000), pp. 209–219.

[25] Rizzo, L. Dummynet: A simple approach to the evaluation of network protocol. *ACM Communication Review* **47**, 1 (Jan. 1997), 31–41.

[26] Rost, S., Byers, J., and Bestavros, A. The Cyclone Server Architecture: Streamlining Delivery of Popular Content. In *Intl Workshop on Web Caching and Content Distribution* (Boston, MA, June 2001).

[27] Saroiu, S., Gummadi, P. K., Dunn, R. J., Gribble, S. D., and Levy, H. M. An Analysis of Internet Content Delivery Systems. In *USENIX Symposium on Operating Systems Design and Implementation* (Boston, MA, Dec. 2002), pp. 315–328.

[28] Saroiu, S., Gummadi, P. K., and Gribble, S. D. A measurement study of peer-to-peer file sharing systems. In *SPIE/ACM Multimedia Computing and Networking Conference* (Jan. 2002).

[29] Steere, D. C. Exploiting the Non-Determinism and Asynchrony of Set Iterators to Reduce Aggregate File I/O Latency. In *ACM Symposium on Operating Systems Principles* (Oct. 1997), pp. 252–263.

[30] Wang, L., Pai, V. S., and Peterson, L. L. The Effectiveness of Request Redirection on CDN Robustness. In *USENIX Symposium on Operating Systems Design and Implementation* (Boston, MA, Dec. 2002), pp. 345–360.

[31] Zhang, Y., Breslau, L., Paxson, V., and Shenker, S. On the Characteristics and Origins of Internet Flow Rates. In *ACM SIGCOMM* (Pittsburgh, PA, Aug. 2002).