# Efficient File Distribution in a Flexible, Wide-area File System

by

Irene Y. Zhang

S. B., Computer Science and Engineering
Massachusetts Institute of Technology, 2008

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2009



Author .....
Department of Electrical Engineering and Computer Science
May 22, 2009

Certified by .....
Jeremy Stribling
Ph.D. Candidate
Thesis Supervisor

Certified by .....
M. Frans Kaashoek
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

# Efficient File Distribution in a Flexible, Wide-area File System

by

Irene Y. Zhang

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 2009, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

## Abstract

WheelFS is a wide-area distributed file system designed to help applications cope with the challenges of sharing data over the wide-area network. A wide range of applications can use WheelFS as a storage layer because applications can control various trade-offs in WheelFS, such as consistency versus availability, using *semantic cues*. One key feature that many applications require from any storage system is efficient file distribution. The storage system needs to be able to serve files quickly, even large or popular ones, and allow users and applications to quickly browse files. Wide-area links with high latency and low throughput make achieving these goals difficult for most distributed storage systems.

This thesis explores using prefetching, a traditional file system optimization technique, in wide-area file systems for more efficient file distribution. This thesis focuses on *Tread*, a prefetcher for WheelFS. Tread includes several types of prefetching to improve the performance of reading files and directories in WheelFS: read-ahead prefetching, whole file prefetching, directory prefetching and a prefetching optimization for WheelFS's built-in cooperative caching. To make the best use of scarce wide-area resources, Tread adaptively rate-limits prefetching and gives applications control over what and how prefetching is done using WheelFS's semantic cues.

Experiments show that Tread can reduce the time to read a 10MB file in WheelFS by 40% and the time to list a directory with 100 entries by more than 80%. In addition, experiments on Planetlab show that using prefetching with cooperative caching to distribute a 10MB file to 270 clients reduces the average latency for each client to read the file by almost 45%.

Thesis Supervisor: Jeremy Stribling
Title: Ph.D. Candidate

Thesis Supervisor: M. Frans Kaashoek
Title: Professor of Electrical Engineering and Computer Science

# Acknowledgments

I would like to thank my advisor, Frans Kaashoek, for dispensing sage advice when I needed it the most, and most importantly, for helping me become a better researcher.

This project would not have been possible without the insight and inspiration of Jeremy Stribling and the efforts of the WheelFS team. I would like to thank Jeremy especially for his continued support and guidance throughout this project and his dedication to reading every word of this thesis.

I would like to thank the PDOS group for providing a stimulating and fun work environment for the last year and a half.

I would like to thank my parents, who have continued to provide encouragement and advice throughout my 5 years at MIT. Finally, I would like to thank Dan Ports for his love and unwavering belief in me as a researcher.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

The focus of this M.Eng. thesis is providing efficient file distribution in WheelFS, a configurable wide-area file system. The goal is to ensure that WheelFS offers low latency file transfers, even when files are very large or popular, and fast browsing of directories over the wide-area network. This chapter first gives background on WheelFS and motivation for the problem, then discusses the basic approach of our solution and the contributions made in this thesis.

## 1.1 Background

Sharing data between a set of wide-area sites poses a challenge for many wide-area distributed applications. There are several factors that make distributing data difficult in wide-area systems:

- The clients and servers are often connected by a wide-area link with high latency and low throughput. Any operation that requires contacting a large number of servers or transferring a large amount of data can become a performance bottleneck for the application.

- The limited throughput of the link between a server and a client limits the scalability of the system. The number of clients that one server can handle simultaneously is limited by the bandwidth of the server's bottleneck link.

- With nodes at several geographically distant locations, it becomes likely that some number of nodes in the system will be unable to communicate with each other at any point in time, so the application must give up either consistency or availability [7].

All of these challenges force distributed applications to make trade-offs to operate over the wide-area network. Different applications make these trade-offs in different ways, so many wide-area applications require a complex, custom storage layer for sharing data between nodes.

WheelFS [18] offers a storage solution for wide-area applications. WheelFS simplifies the construction of wide-area applications by providing a basic storage layer in the form of a distributed file system. WheelFS is general enough to be used for different wide-area applications because it gives applications control over wide-area trade-offs. Using *semantic cues*, an application can adjust the behavior of WheelFS to suit its requirements. Applications express semantic cues as keywords embedded in the pathname. For example, WheelFS allows applications to use cooperative caching when the application knows that many nodes will simultaneously access the file. In addition to cooperative caching, there are WheelFS cues for file placement, durability and consistency.

## 1.2 Problem Statement

One feature that many applications require from a storage system like WheelFS is efficient file distribution. The storage system must be able to serve files, even large or popular ones, quickly and support efficient browsing of directories. For example, a user maybe storing large executables for a distributed experiment in WheelFS. At the start of the experiment, the executable needs to be efficiently distributed to a large number of nodes in a short period of time. Another example might be a user running `ls -l` on a large directory. If each file in the directory is on a different server, a large number of wide-area servers may have to be contacted to list the directory.

A naive design cannot easily cope with the problem of efficient file distribution

over the wide-area network. For example, the basic design of most cluster file systems does not work efficiently in the wide-area environment [22]. A cluster file system would not offer good performance in either of the two scenarios above. In the first scenario, the servers' bottleneck links would quickly become saturated and the latency would increase linearly as the number of clients increases. In the second scenario, the user would have to wait while the NFS client sends one request to a server over the wide-area network for each item in the directory.

## 1.3 Proposed Solution

We provide efficient file distribution in WheelFS by adapting prefetching, an optimization technique from traditional file systems, to a wide-area file system. Prefetching is a predictive technique used by many traditional file systems to improve performance when reading files from disk. The file system preemptively fetches data not yet requested by the application and caches it, so the data will be immediately available if the application requests it later. Prefetching works well for traditional on-disk file systems because the disk seek for fetching a piece of data dominates the cost of reading [17]. Prefetching data located consecutively after the requested data on disk adds little additional cost to the original disk seek made for the request and studies have show that most file reads are sequential [2]. This technique of prefetching data located after the requested data is known as read-ahead.

Prefetching can provide many performance benefits for a wide-area file system. Prefetching can hide the latency of wide-area file transfers by fetching data asynchronously from servers while the application is busy. Because the latency of the link between the client and the server is high, synchronously sending small requests does not use all of the available bandwidth. With prefetching, WheelFS can make better use of limited wide-area bandwidth by allowing several requests to be in-flight between clients and servers. Prefetching can improve the performance of cooperative caching by allowing WheelFS clients to fetch blocks in a more randomized order to avoid synchronization with other clients.

This thesis presents Tread, a prefetcher for WheelFS. Tread uses prefetching in several ways to provide efficient file distribution in WheelFS. Like traditional file systems, Tread uses read-ahead to predictively prefetch file data. Using WheelFS's semantic cues, Tread gives applications control over whole file prefetching, where Tread starts a prefetch of the entire file when it is first read. To improve the performance of listing directories, Tread prefetches file and directory attributes, so requests to many servers can be made in parallel. Applications can also control directory prefetching with a semantic cue. WheelFS uses cooperative caching to serve popular files efficiently. As an optimization for when the application uses cooperative caching and prefetching together, Tread randomizes the order in which it prefetches the pieces of the file to avoid synchronization between clients and maximize the amount of data fetched from other clients rather than from the servers.

Traditional file system prefetching must be adapted for a wide-area file system because the data in a wide-area file system resides on a server connected by a wide-area link rather than on a disk connected by a bus. Aggressive prefetching in a wide-area file system can place additional load on the servers, preventing it from serving actual read requests quickly. More importantly for wide-area file systems, predictive prefetching can waste valuable wide-area bandwidth if the application does not end up using the prefetched data.

We tackle these wide-area challenges by rate-limiting prefetch requests and using semantic cues to give applications control over more aggressive prefetching. In order to ensure that WheelFS servers are not overloaded by prefetching requests, all prefetching is controlled by an adaptive rate-limiting algorithm that controls the rate at which prefetch requests are issued. To minimize wasted bandwidth, WheelFS uses conservative predictive prefetching by default and allows applications to request more aggressive prefetching, such as whole file prefetching, with semantic cues.

## 1.4 Contributions

The primary contribution of this thesis is Tread, the prefetcher in WheelFS. Tread includes a mechanism for prefetching data and several prefetching policies. More specifically, the contributions are:

- **Adaptive prefetch rate-limiting.** Tread adjusts the rate at which prefetch requests are issued to make good use of the bandwidth between clients and servers without overloading the server's resources with prefetch requests. The rate-limiting algorithm is adaptive and is not controlled by the application.

- **Read-ahead prefetching.** Tread can do read-ahead prefetching similarly to traditional disk-based file systems. WheelFS always uses read-ahead prefetching; the application does need to enable read-ahead with semantic cues. In the case of read-ahead prefetching, Tread must balance the amount of prefetching with bandwidth usage because it is not clear ahead of time if the prefetched data will be useful to the application. To evaluate the effectiveness of read-ahead, we compared the latency of file transfers with and without read-ahead prefetching.

- **Whole file prefetching.** With whole file prefetching, Tread starts prefetching the entire file when the application first reads the file. Whole file prefetching is more expensive than read-ahead prefetching, so WheelFS provides a semantic cue to give applications control over whole file prefetching. When an application knows that it will be using the entire file, the application can request whole file prefetching by accessing the file with the **.WholeFile** cue. The **.WholeFile** cue informs Tread that prefetching will be valuable to the application, so Tread strives to minimize latency rather than balancing prefetching and bandwidth usage. We evaluate the effectiveness of **.WholeFile** cue in comparison to no prefetching at all and WheelFS's default read-ahead prefetching.

- **Directory prefetching.** Tread prefetches file and directory attributes for improving the performance of listing directories. The application can request directory prefetching by using the **.WholeFile** cue to read the directory. We

compare the performance of directory prefetching with no prefetching to evaluate the benefit of directory prefetching.

- **Cooperative caching with prefetching.** WheelFS can use cooperative caching on a per-request basis when the file is accessed with the **.Hotspot** cue. When the **.Hotspot** cue is used with prefetching, Tread prefetches the file from other clients and reads the file in random order to avoid synchronization with other clients. We performed a performance evaluation with the several hundred clients simultaneously reading a file using cooperative caching, with and without prefetching.

## 1.5 Outline

The remainder of this thesis discusses the design, implementation and evaluation of Tread. Chapter 2 gives an overview of the design of WheelFS as background to the discussion in the following chapters. Chapter 3 discusses the design of the prefetching mechanism and the various prefetching policies used in Tread. Chapter 4 describes the implementation of the prefetching mechanism and policies. Chapter 5 gives a performance evaluation for each of the optimizations. Chapter 6 reviews the related work and Chapter 7 concludes.

# Chapter 2

# WheelFS

This chapter describes the design of WheelFS to provide context for the rest of this thesis. We first give an overview of WheelFS's system model and default behavior, then explain the basic parts of WheelFS that support the default behavior and finally, review some of WheelFS's semantic cues. The WheelFS paper [18] provides the full details about the design of WheelFS.

## 2.1 WheelFS Overview

WheelFS is designed to run on nodes at geographically diverse sites. A WheelFS deployment consists of WheelFS servers that are responsible for storing data and WheelFS clients that access stored data on behalf of applications. Applications running on one node can share a WheelFS client or run separate WheelFS clients. The servers and clients may be separate sets of nodes or a single host can run both a WheelFS client and a WheelFS server. In addition, a small set of nodes at multiple sites run a separate configuration service that is responsible for tracking WheelFS servers and which files each server stores. Figure 2-1 shows what an example deployment might look like. We expect the nodes to be managed by a single administrative entity or several cooperating entities. WheelFS handles the occasional failure of nodes, but we do not expect Byzantine failures.

WheelFS clients contact WheelFS servers to access and modify files and directories.

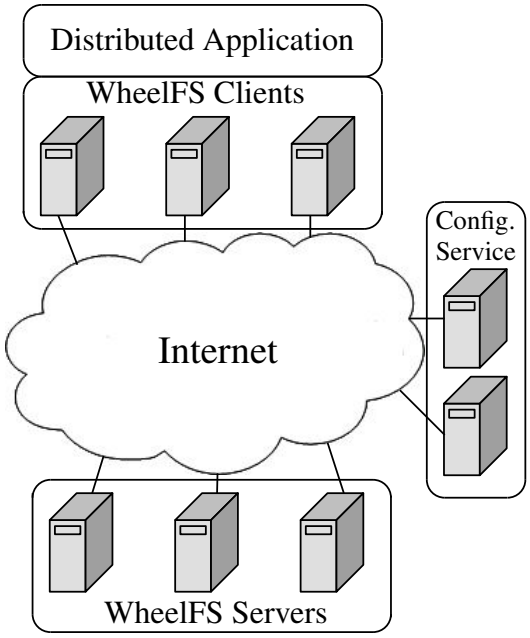



**Figure 2-1: WheelFS system model.** A WheelFS deployment consists of servers and clients scattered over the wide-area. In addition, nodes at several sites run the configuration service. Each node in the distributed application runs a WheelFS client to access data stored on the WheelFS servers.

Each WheelFS client keeps a local cache of directory and file contents. WheelFS clients can serve data to other clients out of their caches, enabling cooperative caching. Each file and directory has a primary server that stores the latest version of the file. WheelFS clients use the configuration service to find the primary server for a file or directory. A client may have to contact multiple WheelFS servers to access a file or directory because each directory in the file's pathname may have a different primary server. Likewise, a client may have to contact several servers to list a directory because each entry in the directory may have a different primary server.

### 2.1.1 Default file system semantics

WheelFS presents a POSIX file system interface with a hierarchical organization of directories and files. All clients in a deployment see the same files and directories in the mounted WheelFS file system. By default, WheelFS provides close-to-open consistency. If one client writes to a file and then closes the file, any client that opens

the file afterwards will see the changes. WheelFS also provides UNIX-like access controls for files and directories. We assume that no nodes in the system misbehave, so clients and servers are trusted to enforce these user access controls.

## 2.2 WheelFS Design

This section discusses some of the basic implementation details, such as how files are stored and accessed, that are relevant to the rest of this thesis.

### 2.2.1 Data placement

WheelFS servers store file and directory objects. Each object is named by a unique object ID. File objects consist of meta-data and opaque file data. Directories consist of meta-data and name-to-object-ID mappings that make up the directory. The ID space is divided into slices. Each slice is assigned to one primary server, which is responsible for storing the latest version of all of the files and directories in the slice.

WheelFS has a write-local data placement policy, meaning WheelFS will try to use a local server as the primary server of a new file by default. If each node runs both a WheelFS client and WheelFS server, then the local host can be used as the primary. Otherwise, WheelFS will choose a nearby WheelFS server as the primary.

### 2.2.2 File access

WheelFS uses FUSE to present a mountable file system to the application. FUSE translates file system operations from the operating system into user-level calls in the WheelFS client. When an application calls `read()` on a file, the operating system breaks the read request into one or more read operations on the file system. FUSE turns those operations into read operations in WheelFS.

The WheelFS client handles the read operation for the application by fetching file and directories from the WheelFS servers. The client fetches file data from the servers in blocks and one primary server is responsible for all of the blocks of a file. The

WheelFS client always makes requests on behalf of a user, generally the user running the application. The WheelFS client and WheelFS server use the user ID associated with the file request to enforce access controls. Every WheelFS file is versioned; the version number incremented with each update to the file.

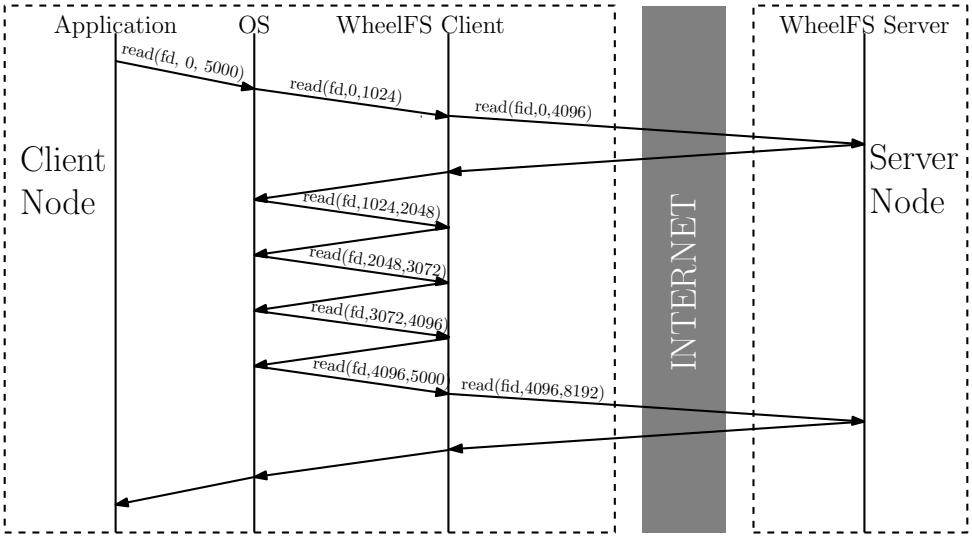



**Figure 2-2: Sequence of function calls that occur for a file read in WheelFS.** In this example, the operating system is reading in 1KB blocks and WheelFS uses 4KB blocks (in reality, Linux generally uses 4KB blocks and WheelFS uses 64KB blocks). The application reads 5KB from the file, which the operating system turns into five 1KB reads on the file system. The WheelFS client only needs to request two 4KB blocks from the server, but the WheelFS client cannot make these two requests to the server in parallel because the operating system synchronously requests each 1 KB chunk of the file.

Figure 2-2 gives an example of what a file read looks like in WheelFS without any prefetching. Notice that even though the application is performing one large read, WheelFS actually sees many small read calls. Likewise, when an application calls `ls` or `readdir()` on a directory, the WheelFS client receives several requests from the operating system. Figure 2-3 shows the sequence of function calls that occur when a user lists a directory.

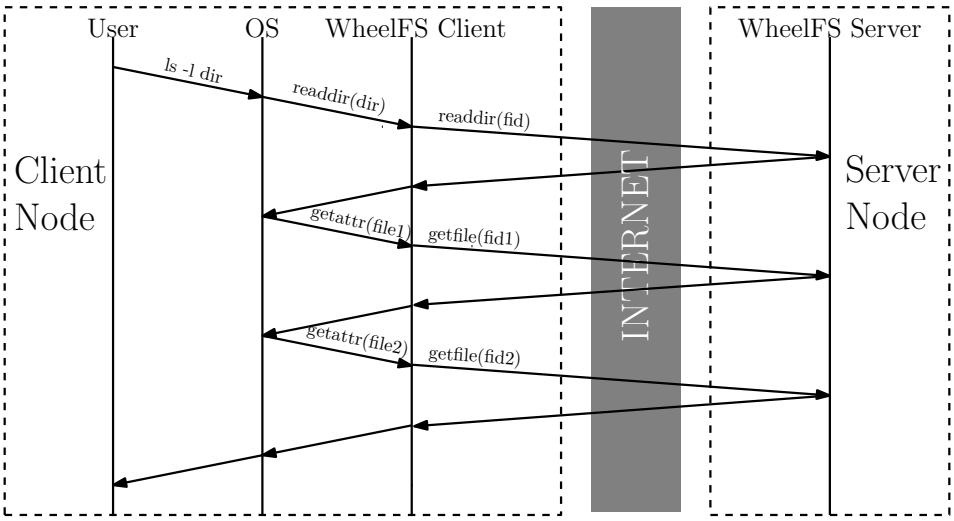



**Figure 2-3: Sequence of calls made when a user performs a `ls -l`.** In this example, a user is listing a directory with two files. The operating system makes one readdir request to get the list of files in the directory and then requests the meta-data for each file in the directory. For each request from the operating system, the WheelFS client must send one request to the server over the wide-area network. The operating system makes each request synchronously, so the WheelFS client also must make each request to the server synchronously.

### 2.2.3 Cooperative caching

WheelFS clients have the ability to serve file data from their caches, enabling cooperative caching. To read a file using cooperative caching, the WheelFS client first contacts the primary WheelFS server to get a list of other clients that have the file. Then the WheelFS client chooses a few nearby peers from the list and requests blocks of the file from the selected peers. Cooperative caching increases the latency for fetching files because the WheelFS client must make a request to the WheelFS server and then make requests for the block to several other clients. But cooperative caching can reduce the overall latency for all clients, especially if many clients are making simultaneous requests for the file to one WheelFS server. Because cooperative caching presents a trade-off between lower latency for one client and lower average latency for all clients, cooperative caching is controlled by a semantic cue, which we will discuss in the following section.

## 2.3 Semantic Cues

Applications can change many of WheelFS's default behaviors using semantic cues. In order to preserve the POSIX file system interface for compatibility, applications specify cues in the pathname. Cues can be added to the pathname for any file system operation to change file system behavior. Cues could also be implemented in other various ways, such as a configuration file, but embedding cues in the pathname allows the most reuse of existing applications.

Cues are one of two types: persistent or transient. Persistent cues must be used in the creation of the file and apply to the file permanently. Transient cues are used when accessing the file and only apply to the current operation. Multiple cues can be used together; the cues are separated by slashes in the pathname like directories.

WheelFS clients parse cues using two rules:

1. A cue applies recursively to all files and directories that follow the cue in the pathname.

2. Cues placed later in the pathname override previous cues in the pathname.

Table 2.1 lists some of the cues currently available in WheelFS.

**Table 2.1:** List of semantic cues currently available in WheelFS

| Cue | Behavior |
|---|---|
| **.EventualConsistency** | Relax open-close consistency |
| **.MaxTime=**$T$ | Upper-bound time spent looking for a file to $T$ ms |
| **.Site=**$X$ | Store data with a node at $X$ |
| .KeepTogether | Store all files in a directory on as few nodes as possible |
| **.RepSites=**$N_{RS}$ | Store replicas across $N_{RS}$ different sites |
| **.RepLevel=**$N_{RL}$ | Keep $N_{RL}$ replicas of the data object |
| **.SyncLevel=**$N_{SL}$ | Return from update as soon as $N_{SL}$ servers have accepted the update |
| **.Hotspot** | Use cooperative caching |

As an example of how WheelFS's cues can be used, we look at how to build a simple distributed web cache, similar to content distribution networks, using Apache caching proxies and WheelFS. With WheelFS cues, it is easy to turn an application

designed to run on a single host into a distributed application. The only modification needed is changing the Apache caching proxy's configuration file to store cached files in the folder show below:

```
/wfs/.EventualConsistency/.MaxTime=1000/.Hotspot/cache/
```

`/wfs` is where the WheelFS file system is usually mounted. By storing cached data in WheelFS, the Apache proxies running at different sites can share data and become a distributed web cache. The **.EventualConsistency** cue relaxes the requirement that all accesses must go through the primary, so that clients can use stale cached data or data from backup servers. We use the **.EventualConsistency** cue because files are mostly static and performance is more important for a cache than consistency. The **.MaxTime** cue bounds the amount of time that WheelFS should spend looking for the file before returning an error and allowing the Apache proxy to go to the origin server. The web cache needs the **.MaxTime** cue to bound the amount of time spent looking for a cached copy because the cache can always go to the origin server. Finally, we use the **.Hotspot** cue for cooperative caching.

# Chapter 3

# Prefetching Design

This chapter discusses the design of Tread. Tread uses several types of prefetching (*i.e.*, read-ahead prefetching, whole file prefetching, etc.) to achieve different performance goals. Tread uses one common prefetching mechanism for actually fetching data from the servers, which supports all of the different types of prefetching. Therefore, we divide the design into two parts: (1) the common prefetch mechanism that performs the actual prefetching and (2) the various prefetching policies that regulate how prefetching happens.

First, Section 3.1 gives an overview of the performance goals for Tread and the ways in which Tread uses prefetching to achieve those goals. Section 3.2 explains the design of the prefetching mechanism and Section 3.3 discusses the different prefetching policies.

## 3.1 Overview

Tread offers several performance benefits for WheelFS. In particular, there are five performance goals we would like to accomplish with Tread:

1. **Reduce the latency for reading a file from a server.** We would like to reduce the latency for reading a file both when the application knows what it will be reading ahead of time and when the application does not know.

2. **Reduce the latency for listing a directory from the servers.** We would like to reduce the time it takes to list a directory, especially when directories are large.

3. **Use available bandwidth for prefetching.** We want to make full use of the available bandwidth on the bottleneck link between the client and the server, while ensuring that prefetching does not take so much bandwidth that it slows down application requests.

4. **Do not overload the servers with prefetch requests.** We want to ensure that server does not get overwhelmed with more prefetch requests than it can handle. The WheelFS servers serve all requests in order, so prefetch requests can delay requests from applications.

5. **Improve the performance of cooperative caching.** We would like to lower the average latency of fetching a file for all clients by increasing the sharing between WheelFS clients when the application uses cooperative caching with prefetching.

Tread uses several different prefetching policies to realize these performance goals. Tread has a single prefetch mechanism to prefetch data, but uses polices to decide what data to prefetch and how to prefetch the data. By default, Tread uses two conservative prefetching policies:

- **Rate-limited prefetching.** Tread uses an adaptive algorithm to limit the number of requests in-flight between the client and the server at a time. The adaptive algorithm minimizes the number of prefetch requests pending at the server, while ensuring there are enough in-flight requests to use all available bandwidth.

- **Read-ahead prefetching.** After an application reads part of a file, Tread predictively prefetches some data located after the requested portion of the file in hopes that the application will continue linearly reading the file. Read-ahead

prefetching reduces the latency of file reads when the application is reading linearly.

Tread also has several prefetching policies that are controlled by the application with semantic cues:

- **Whole file prefetching.** When an application first reads some part of a file with the **.WholeFile** cue in the pathname, Tread will prefetch the rest of the file. Whole file prefetching reduces the latency of file reads when the application knows that it will be using the whole file.

- **Directory prefetching.** When an application reads a directory with the **.WholeFile** cue in the pathname, Tread will prefetch meta-data for all of the entries in the directory. Directory prefetching reduces the latency of listing a directory.

- **Cooperative caching with prefetching.** When an application uses cooperative caching with prefetching, Tread will randomize the order in which blocks are prefetched. Randomizing the prefetch ordering maximizes the amount of sharing between clients because each client will have different blocks cached.

Each prefetching policy dictates some aspect of the behavior of the prefetching mechanism. Read-ahead, whole file and directory prefetching all dictate what the prefetching mechanism requests. The rate-limiting algorithm controls the rate at which the prefetching mechanism sends requests, while the cooperative caching policy controls the order of prefetch requests. Section 3.3 talks more about the design of these prefetch policies.

## 3.2 Prefetching Mechanism

This section explains Tread's prefetching mechanism that supports the different types of prefetching and performs the actual prefetching of data. Sections 3.3 describes how Tread uses the prefetching mechanism for the different types of prefetching.

### 3.2.1 Design overview

Tread uses the same mechanism to prefetch both file contents and directory contents. The prefetching mechanism prefetches file data in blocks like normal read operations. The prefetching mechanism has a pool of *prefetch threads* to asynchronously prefetch data. Requests for file data and file meta-data are placed in separate file and directory *prefetch queues*. A prefetch thread processes each request in first-in, first-out order with directory requests served first. For each request, the prefetch thread will send a request for the block or meta-data to the WheelFS servers and then store the result in the client's local cache for later use. The number of prefetch requests in-flight between the client and the server is limited by a *prefetch request window*, so Tread can control the rate of prefetch requests. Figure 3-1 gives an overview of the different parts of the prefetching mechanism and how they interact.

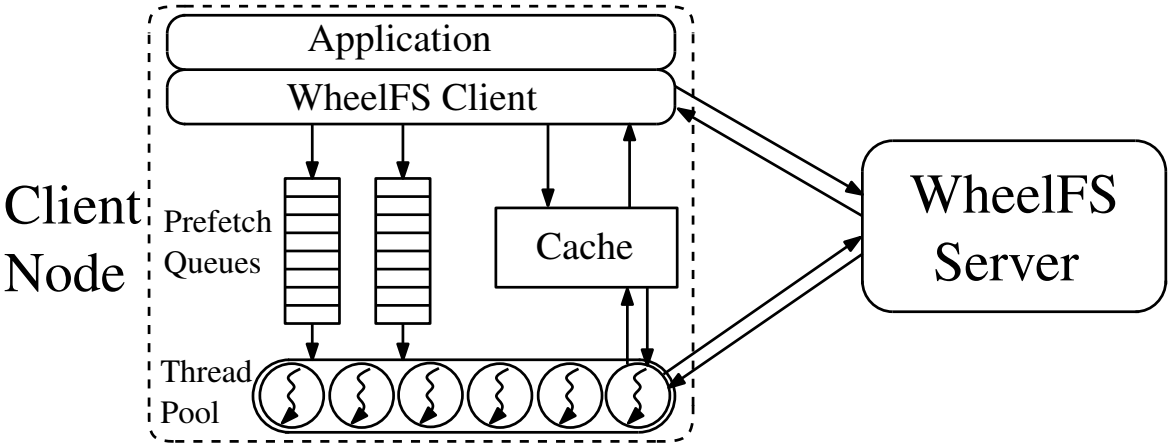



**Figure 3-1: Overview of the Tread prefetching mechanism**. The prefetching mechanism prefetches both file data and file meta-data. Tread requests prefetching by placing a request in a prefetch queue. The prefetching mechanism keeps two queues: one for data and another for meta-data. The prefetching mechanism performs prefetching asynchronously in the background using prefetch threads. The prefetching mechanism keeps a pool of prefetch threads to handle prefetch requests. For each request, the prefetch thread sends a request to the WheelFS server and stores the returned value in the cache for the WheelFS client to use later.

### 3.2.2 Handling prefetch requests

The prefetching mechanism stores information for each file data and meta-data request in a *request table*. File data requests are stored on a per-block basis and meta-data requests are stored on a per-file basis. Each request is identified by a unique *request tag* that includes the file ID, a flag indicating whether it is a data or meta-data request, and if it is a data request, the block number. This tag is used to look up request information in the request table and inserted into the file and directory request queues.

When the request is picked up by a prefetch thread, the prefetch thread uses the request tag to find the *request information* in the request table. The request information includes everything the prefetch thread needs to make a block or meta-data request to the WheelFS server. The information includes the WheelFS file ID, the user ID of the application requesting the prefetching and the WheelFS file version number. For a data request, the prefetch thread synchronously sends a read request for the block to the WheelFS server and places the returned block into the local cache. For a meta-data request, the prefetch thread sends a read request for the meta-data and stores returned meta-data in the local cache.

### 3.2.3 Avoiding duplicate requests

If the WheelFS client gets a request from the application for a file's meta-data or a file block that is in the prefetch queue or currently being prefetched, we would like to avoid making a duplicate request to the WheelFS server. In order to avoid duplicate requests, Tread uses the request table to track pending requests. In addition, Tread keeps information on which request each thread is working on, so the WheelFS client will know if it needs to wait for a thread to finish processing a prefetch request.

Before the WheelFS client makes a request to the server on behalf of an application, the client first checks with Tread's prefetch mechanism to see if the block or meta-data has been requested for prefetching. If not, the WheelFS client fetches the block or meta-data from the WheelFS server. If a prefetch request for the block or meta-data is currently pending, Tread removes the request from the request queue and allows

the WheelFS client to perform the request. If a prefetch thread is working on the request, the WheelFS client waits for the thread to finish.

### 3.2.4 Limiting the request rate

Tread modulates the prefetch rate to ensure that requests are not being sent faster than the servers can handle, but all of the available bandwidth between the client and servers is being used. In order to allow Tread to modulate the request rate, the prefetch mechanism limits the number of in-flight requests to the prefetch request window. Inherently, the rate for prefetch requests is limited to the size of the thread pool because requests to the server are always synchronous, so each prefetch thread can only send one request at a time. The prefetch mechanism further limits the number of in-flight requests to the prefetch request window, so Tread can adaptively adjust the prefetch rate, as discussed in the next section.

## 3.3 Prefetching Policies

This section discusses the design of the different prefetching policies in WheelFS.

### 3.3.1 Rate-limited prefetching

Rate-limited prefetching only applies to file prefetching. Tread does not rate-limit directory prefetching because directory requests are small. Tread starts with a base request window size of 1 and adjusts the window size adaptively. The goal is to send prefetch requests fast enough to make use of available bandwidth, but not so fast that prefetch requests overwhelm the server or slow down other traffic on the link. Tread rate-limits all file prefetching by default; applications cannot turn off or control the rate-limiting with a semantic cue.

The high-level goal of the adaptive algorithm is to maximize the throughput of prefetching while keeping the latency of each block request within some acceptable bounds. The rate-limiting algorithm is similar to the window scaling used by TCP [9].

But unlike in TCP, the prefetch mechanism does not see lost packets, so instead, the prefetching mechanism uses latency for determining when to scale back the window size.

Tread times the latency for each block request sent to the server. Tread tracks the performance of each prefetch window size by keeping the average latency of a block request for each window size. Tread does not track average latency on a per server basis, but overall for all servers. Using one average works well if the client is talking to one server at a time or if the bottleneck link is between the client and all servers. We chose this approach because WheelFS clients are usually used by one application, so in general will be prefetching files from only one server at a time.

Using average latency, Tread calculates the average throughput for each window size by dividing the window size by the average latency. Measuring throughput as well as latency is important because often a rise in average latency per block request does not correspond with a drop in throughput if the prefetch window size has increased. Therefore, Tread must measure the average throughput per window size in order to maximize throughput overall.

Starting with a window size of 1, Tread increases the window size by 1 each time a request returns until the latency of the current window size is more than 20% more than average latency of the base window size of 1. If the latency increases, but the throughput increased as well, Tread decreases the window size by one. This case indicates that the server has started queuing pending requests and can no longer serve prefetch requests as quickly as they come, but the client is still getting good throughput for prefetch requests. If the latency increases and the throughput decreases, Tread halves the prefetch window size. This case indicates that the amount of available bandwidth has changed or the server has become so severely overloaded that congestion collapse is happening, so Tread backs off aggressively. Algorithm 1 gives the psuedocode for the adaptive rate-limiting algorithm.

Since the adaptive algorithm is based solely on latency, the algorithm cannot perfectly predict what the optimal prefetch window size should be. The algorithm is constantly adjusting the window size, but the window size generally averages close

to the optimal value. Because the prefetch algorithm cannot keep the window size constantly optimal, Tread does not always provide optimal performance for prefetching. Section 5.2 gives an evaluation of the performance of the adaptive rate-limiting in Tread.

**Algorithm 1 Psuedocode for adaptive-rate limiting algorithm** The adaptive algorithm chooses the next window size based on the throughput of the current window size and latency. The algorithm compares the latency for the current window size to the base latency, which is the average latency for a window size of 1, and compares the throughput against the average throughput for next smallest window size.

```
if latency < 1.2 · baselatency then
  if winsize/latency > avgthroughput[winsize − 1] then
    winsize ← winsize + 1
  else
    winsize ← winsize − 1
  end if
else
  if winsize/latency > avgthroughput[winsize − 1] then
    winsize ← winsize − 1
  else
    winsize ← winsize/2
  end if
end if
```

### 3.3.2 Read-ahead prefetching

Like other file systems, WheelFS uses sequential file read-ahead. When the application reads some number of blocks, Tread will prefetch the same number of consecutive blocks from the file. For example, if an application reads the first two blocks from a file, Tread will prefetch the third and fourth blocks.

There is already some read-ahead inherent in the design of WheelFS because file data is always fetched in 64KB blocks. So even if the application only reads the first byte of a block, WheelFS will fetch the entire block. But read-ahead prefetching adjusts the amount of prefetched data based on the amount of data already requested by the application, rather than only in 64KB chunks.

Doubling the number of blocks was chosen as a simple metric for predicting how much data to prefetch. More complicated predictive algorithms have been explored

for prefetching [11, 8], but doubling the number of blocks achieves good performance for linear file access, as we will show in the evaluation section. A doubling policy is simple, but still uses how much the application has read to choose how much to prefetch and is conservative enough to use by default for WheelFS.

Read-ahead is simple to implement with the prefetching mechanism described in the previous section. For each read operation, the WheelFS client will fetch some number of blocks to satisfy the application's read request. After the WheelFS client fetches the requested blocks, Tread places the same number of consecutive blocks into the prefetch queue. The prefetch threads will handle the prefetching request from there; fetching the blocks and placing the blocks into the cache. Figure 3-2 gives an example of what happens in WheelFS during a normal read with read-ahead prefetching.

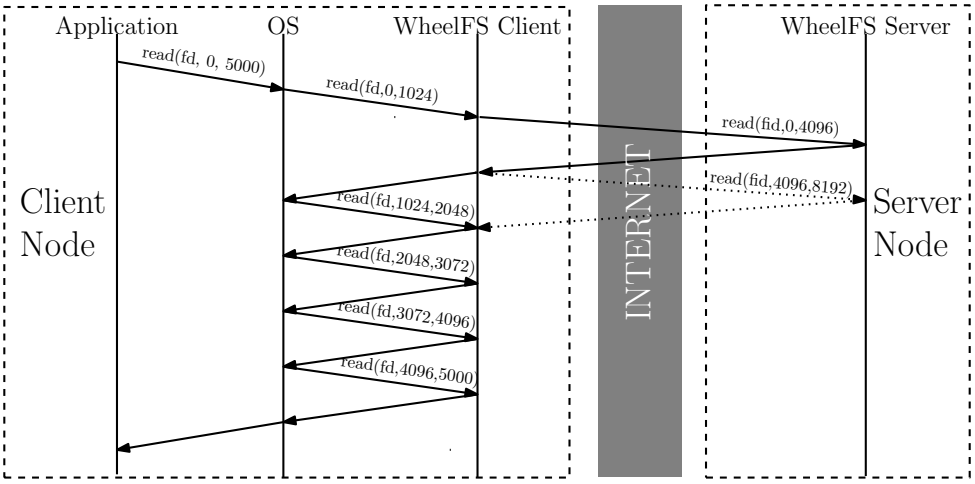



**Figure 3-2: Example of read-ahead file prefetching.** The dotted lines represent requests to the server made by the prefetch threads. The read proceeds just as without prefetching, but after the first block is requested, the second block of the file is prefetched. Now when the application needs the second block, the WheelFS client can serve the block out of the cache without making the application wait while it makes a request to the server.

### 3.3.3 Whole file prefetching

Applications can use the **.WholeFile** cue to tell WheelFS that it will be using an entire file or directory, causing Tread to start whole file prefetching when the file is first read. With whole file prefetching, the entire file is queued for prefetching immediately because WheelFS knows that the application will be using the prefetched data. Unlike read-ahead prefetching, whole file prefetching is not predictive, but is controlled by the application.

When an application calls `read()` on a file with the **.WholeFile** cue in the pathname, the WheelFS client first handles the read request normally by fetching blocks from the WheelFS server. After the client has finished the application's read request, it tells Tread to puts all other blocks in the file into the prefetch queue and the prefetch mechanism starts working on the requests. The WheelFS client sends the application's requests to the server first to ensure that the WheelFS server handles those requests before prefetching requests.

### 3.3.4 Directory prefetching

When an application calls `readdir()` on a directory with the **.WholeFile** cue in the pathname, Tread places a directory prefetch request. The client first synchronously performs the readdir operation in the normal way. After the WheelFS client gets the list of directory entries from the readdir request to the server, the client gives the list to Tread and Tread places a meta-data request for each entry into the prefetching queue. The prefetching mechanism handles the actual prefetching as described in Section 3.2. Figure 3-3 gives an example detailing the function calls that occur when an application reads a directory and prefetches meta-data.

### 3.3.5 Cooperative caching with prefetching

When the application combines cooperative caching with prefetching, Tread randomizes the order in which blocks are prefetched. This randomization minimizes the chance that clients become synchronized as they linearly read the file and simultaneously

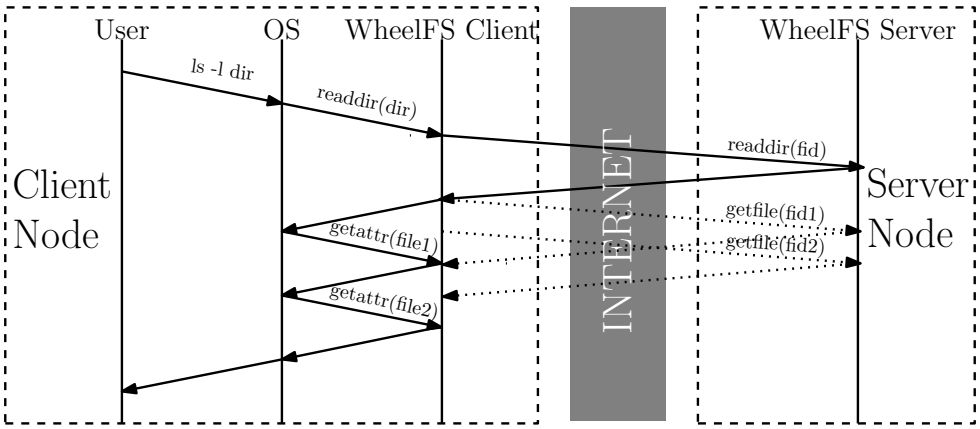



**Figure 3-3: Example of directory prefetching.** As without prefetching, the WheelFS client makes a `readdir` request to the server to fetch the contents of the directory. As soon as the `readdir` returns from the server, Tread starts prefetching meta-data for the files in the directory. Now, each `getattr` request from the operating system can be served from cache and does not require a request to the WheelFS server.

request the same block at the same time. In this way, randomization further reduces the load on the WheelFS server. Randomizing block requests would not be possible without prefetching because the operating system always requests blocks in a linear order, so WheelFS must fetch blocks in a linear order as well.

Randomizing the order in which blocks are prefetched is simple using the prefetching mechanism. When Tread places blocks into the prefetch queue, Tread first checks to see if the application is using the **.Hotspot** cue. If so, Tread randomizes the order of the blocks that need to be fetched before placing them into the queue. Since the prefetch threads serve requests from the queue in a first-in, first-out order, the prefetch threads will always request the blocks in the order in which they were placed into the queue.

# Chapter 4

# Implementation

This chapter describes the implementation details of the prefetching mechanism and various prefetching policies in Tread. WheelFS and all of Tread is implemented in C++. Overall, the prefetching mechanism consist of around 600 lines of code and the prefetching polices are implemented in around 400 lines. Like the rest of WheelFS, Tread uses the pthread library for threading and the C++ Standard Template Library for various data structures.

## 4.1 Prefetching Mechanism

The prefetching mechanism in Tread is responsible for fetching requested data. The implementation must also track the state of each prefetch request in case the WheelFS client receives a request from the application for that block or meta-data. The prefetching mechanism uses several data structures to track prefetch requests. Each request consists of request information and a request tag that identifies the request as described in Section 3.2. The request table implemented as a hash table that keeps the mapping between request tags and request information. The requests in the request table represent every prefetching request that is either waiting to be served or is currently being served. Each thread has a *thread ID*. The prefetch mechanism tracks the thread ID of the prefetch thread, if any, that is working on the request. The number of prefetch threads allowed to serve requests at once is kept at the prefetch

window size by a semaphore.

If the WheelFS client receives a request from the application that requires some block or meta-data that has been scheduled for prefetching, it is important that Tread can easily distinguish the state of the prefetch request. If a prefetch thread is working on the request, Tread needs to be able to tell the WheelFS client which thread to wait on. If the request is still queued, Tread needs to dequeue the request and let the WheelFS client make the request.

Prefetch requests go through 3 states: pending, working and complete. Tread can distinguish which state the request is in by checking the request table, request information and the WheelFS client cache. When the request is pending, the request is in the prefetch table and there is no thread ID associated with the request because no prefetch threads has claimed it yet. When a prefetch thread is working on the request, the request is still in the request table, and the request information contains the thread ID of the thread that is processing the request. When request has been completed, the request will no longer be in the request table, but the block or meta-data will be in the client cache.

Once Tread dequeues a prefetch request, the request table no longer tracks the request. Although Tread tracks the request that each prefetch thread is working on, we decided not to track what requests the WheelFS client threads are working on. Therefore, there is one case where two duplicate requests could be sent to the WheelFS server. If Tread dequeues a prefetch request for one WheelFS client thread, another client thread could make a new prefetch request for the same block or meta-data. If the new prefetch request is immediately picked up by a prefetching thread before the first WheelFS client thread finishes fetching the block or meta-data, the prefetching thread will make a duplicate request to the WheelFS server. This case could be handled by tracking requests made by the WheelFS client threads as well, but we chose not to because it does not result in incorrect behavior, only a slight performance loss.

## 4.2 Prefetching Policies

The implementations of the prefetch policies are relatively simple. Other than the adaptive rate-limiting algorithm, the prefetch policies deal with which blocks to place in the prefetch queue and the order in which the blocks are placed. Therefore, the implementations of read-ahead, whole file and directory prefetching simply consist of calculating the block or meta-data requests to place in the prefetch queue. For read-ahead prefetching, the number of blocks placed in the queue is the same as the number of blocks that have already been fetched in the file. For whole file prefetching, we find the size of the file and place all of the blocks into the prefetch queue. For directory prefetching, Tread uses the list of the contents of the directory that it got from the WheelFS client.

When prefetching is used with cooperative caching, Tread places the prefetch requests for each block into the queue in a random order. First, Tread makes a list of all of the blocks that need to be prefetched. For example, with whole file prefetching, this list would include all of the blocks in the file that are not already in the client cache. Next, Tread randomly shuffles the list of blocks using the Knuth shuffle [4]. Finally, Tread places the blocks into the prefetch queue in the shuffled order. The queue maintains the shuffled order because the prefetch threads handle requests from the queue in first-in, first-out order.

The adaptive rate-limiting policy is different from the others because it controls the rate at which prefetch requests are sent rather than deciding what to prefetch. The adaptive algorithm is based on the average latency of prefetch requests to the WheelFS server, so the implementation mainly consists of collecting the latency measurement. The window size will generally change while the request is in-flight, so Tread estimates the window size by taking an average of the window size when the request starts and finishes. The prefetch thread measures each request and adds the measurement to a weighted average for the estimated window size. The adaptive rate-limiting algorithm uses the weighted average and latency measurement to decide whether to increase or decrease the prefetch window size.

# Chapter 5

# Evaluation

We use several experiments to evaluate the performance of Tread. We use two platforms for these tests: Emulab [21] and Planetlab [3]. Emulab is a network emulation testbed that emulates different network topologies. Using Emulab, we can emulate a wide-area deployment of WheelFS by using links with low bandwidth and high latency. We also emulate various different bandwidths to test our adaptive rate-control mechanism. Planetlab is a wide-area testbed with almost 1000 nodes running around the world. Nodes in Planetlab are connected by the wide-area Internet, so the Planetlab testbed is very realistic for testing WheelFS. We use Emulab primarily for benchmarking various aspects of Tread and used the large number of nodes in Planetlab to test Tread's cooperative caching optimization.

Our experiments are designed to test whether Tread's prefetching policies achieve the goals listed in Section 3.1. Briefly, the goals are:

1. Reduce the latency for reading a file from a server.

2. Reduce the latency for listing a directory from the servers.

3. Use available bandwidth for prefetching.

4. Do not overload the servers with prefetch requests.

5. Improve the performance of cooperative caching.

Section 5.1.1 covers latency and throughput micro-benchmarks, comparing read-ahead prefetching, whole file prefetching and no prefetching. The latency micro-benchmarks show that prefetching does reduce the latency of reading files from the WheelFS servers. In addition, the latency micro-benchmarks evaluate the effectiveness of the sequential read-ahead algorithm by comparing read-ahead prefetching with whole file prefetching. The throughput micro-benchmark shows that WheelFS makes better use of available bandwidth with prefetching. Section 5.2.2 evaluates the effectiveness of Tread's adaptive rate-limiting mechanism for prefetching. We measure the average window size used by the adaptive algorithm with different bandwidths to show that the algorithm adjusts well. Section 5.3 gives a performance evaluation of cooperative caching and prefetching, showing that cooperative caching performs better when it is used with prefetching.

## 5.1 Prefetching Micro-benchmarks

We use several micro-benchmarks to test the effectiveness of prefetching. We use Emulab to emulate WheelFS clients and WheelFS servers connected by a wide-area link. The round-trip time between the server and the clients is 100 ms and the bandwidth of the link is 6 Mbps. For testing purposes, we do not use the rate-limiting algorithm or read-ahead prefetching by default. The number of in-flight requests is indicated for each test and read-ahead prefetching and whole file prefetching are only used where noted.

### 5.1.1 Latency experiments

The primary goal of prefetching is to improve the performance of file distribution by reducing the latency of reading a file from WheelFS. To test the improvement offered by prefetching, we measure the time to read files of differing sizes with and without prefetching. We tested both read-ahead and whole file prefetching. We used one node to write and serve files and four other nodes as clients to read the file. Each client reads the entire file in turn, so the server does not have to serve simultaneous requests.

For prefetching, we limit the number of in-flight requests to 10. We chose 10 requests because it is large enough to ensure that there are enough requests in-flight to make use of all of the bandwidth. A rough rule of thumb for network connections is to use a window that is twice the size of the bandwidth delay product [19], which in the case of this experiment is 75KB. Each block request in WheelFS is at least 64KB. So, as long as the clients keep more than 1.2 requests in-flight, they will be sending a sufficient number of requests to use available bandwidth. Since there is only one client reading at a time, 10 requests in-flight will not overwhelm the network or the server.

We expect the time needed to read a file to be less with prefetching than without. We know that without prefetching the time to fetch a file will grow linearly as the size of the file increases because the WheelFS client makes a synchronous request to the server for each block in the file. We expect the time to read a file with prefetching will grow linearly as well, but remain lower than without prefetching. In this experiment, clients are reading the file linearly, so we expect read-ahead prefetching to perform similarly to whole file prefetching. Figure 5-1 gives the average read times for each file.

From our end-to-end latency measurement, it is clear that prefetching offers better performance. The time to read a file with prefetching remains lower than the time to read a file without prefetching as the size of the file grows. The time to read a file with prefetching still grows linearly, but at a slower rate. This linear growth is expected even in the optimal case because the speed of file transfers is limited by the bandwidth of the link between the server and the client. Both read-ahead and whole file prefetching average between 4 and 4.5Mbps of throughput and reading without prefetching averages about 2Mbps of throughput. We explore throughput more throughly in the next section.

This experiment also shows that read-ahead prefetching and whole file prefetching offer similar performance when the file is being linearly read. This result shows that, although Tread's read-ahead algorithm is fairly simple, it performs well for linear file reads.

We also measure the time to read a directory using directory prefetching and

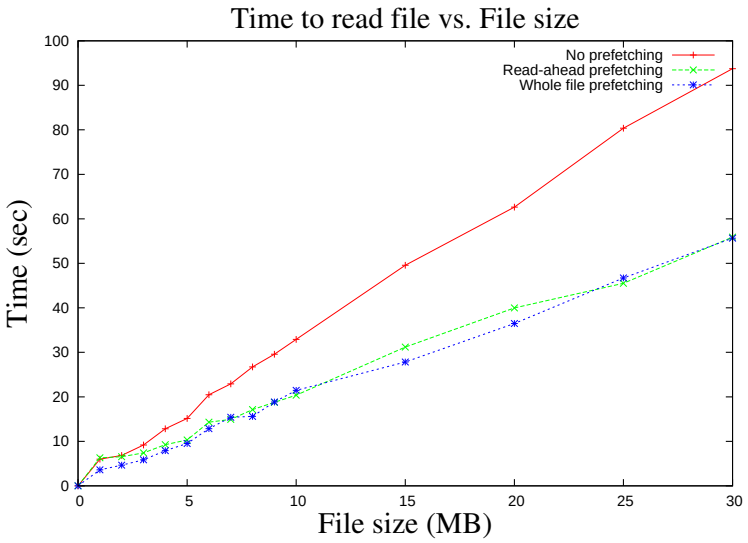



**Figure 5-1: Time to read a file without prefetching, with read-ahead prefetching and with whole file prefetching.** The time to read a file without prefetching increases linearly as the size of the file increases. The timed needed to read a file with prefetching rises linearly as well, but at a slower rate. In this experiment, read-ahead prefetching and whole file prefetching perform similarly because the entire file is being read linearly.

compare that against the time to read a directory without prefetching. One node creates the directory and the files in the directory and acts as the server for the experiment. Another 4 nodes act as clients and each client performs a `ls -l` on the directory in turn. The number of in-flight requests to the server was not limited.

Again, our goal is to show that prefetching directories offers better performance. To list a directory, the operating system makes one request for each file in the directory, so we expect the time to read the directory to grow linearly with the number of files in the directory. Without prefetching, the WheelFS client must make a synchronous request to the WheelFS server for each request from the operating system, so the latency is expected to increase quickly with the size of the directory. With prefetching, the performance should be significantly better. Figure 5-2 gives the average time for reading each directory.

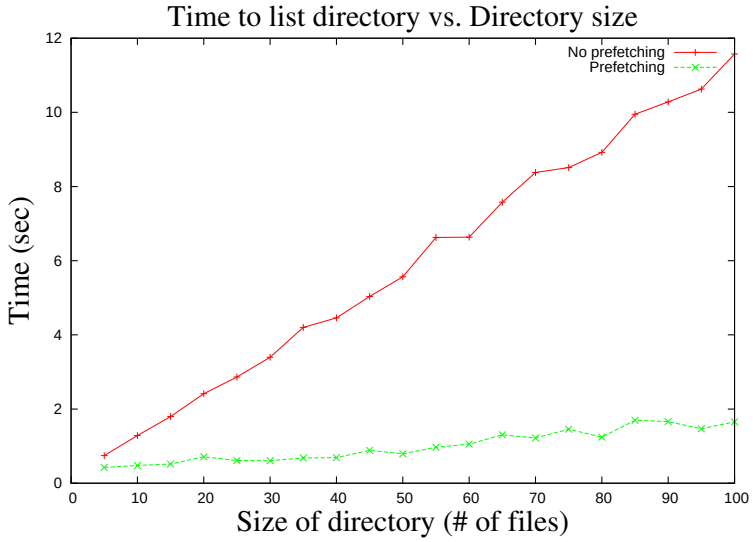



**Figure 5-2: Time to read a directory with and without prefetching.** The time needed to list a directory rises sharply as the size of the directory increases. Listing a directory with prefetching performs better, especially as the directories get larger.

Our experiments clearly show the performance gain from prefetching directories. The latency for listing a directory without prefetching increases quickly as the size of

the directory increases. With prefetching, the WheelFS client can make many small requests in parallel, so the time to list a directory grows very slowly as the number of files increase.

### 5.1.2 Throughput experiments

We test the throughput of WheelFS for reading files with and without prefetching to show that WheelFS does not make good use of available bandwidth without prefetching. We measure the throughput of WheelFS when prefetching with different prefetch window sizes to show that WheelFS makes better use of bandwidth with particular window sizes. For comparison, we also test the throughput of the TCP connection using `ttcp`.

The setup for this experiment is the same as the latency experiments, with one node acting as the server and another 4 nodes as clients. We measure the throughput for reading a 10MB file without prefetching and with whole file prefetching. Read-ahead prefetching performs similarly to whole file prefetching, as we showed in the last experiment. For prefetching, we adjust the number of requests in-flight to the server between 1 and 50. For `ttcp` test, we transmit a 10MB file from the client node to the server node.

The goal of these experiments is to show that prefetching makes better use of available bandwidth than reading a file without prefetching. Without prefetching, the WheelFS client must send one block request at a time to the WheelFS server. We expect WheelFS to use about 2.8Mbps when sending requests synchronously. The total time for a block request to make it to the server and back is 100 milliseconds and the time to actually transmit 64KB of data on a 6Mbps link is 85 milliseconds. Therefore, the WheelFS client receives 64KB of data every 185 milliseconds when sending requests synchronously, giving the client an average throughput of 2.8Mbps.

We hope to show that prefetching can make better use of the available bandwidth by making several requests for file blocks in parallel. Figure 5-3 shows the average throughput for each prefetch window size.

Our experiment shows that, without prefetching, WheelFS uses about 2Mbps.

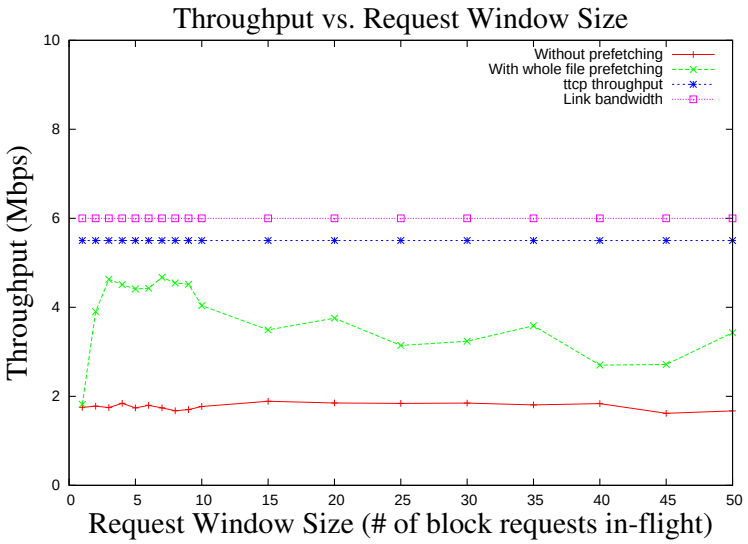



**Figure 5-3: Throughput with read-ahead and whole file prefetching based on the prefetch window size.** Without prefetching, the throughput achieved for file reads depends on the size of each request since only one request is made at a time. WheelFS can take advantage of prefetching by making many requests in parallel to use all of the bandwidth available. The optimal window size appears to be around 3 requests. There is a slight drop in performance after 10 requests because the WheelFS server has a hard-coded limit of 10 concurrent requests, so the server begins to drop requests.

This result indicates that there is some overhead to sending requests in WheelFS. With prefetching, we get a higher throughput, but do not match the bandwidth of the link. Once the number of requests in-flight increases to 2 or 3 requests at a time, the throughput increases to about 5Mbps, using about 80% of the available bandwidth. But the average throughput of the TCP connection measured by `ttcp` is 5.5Mbps. So although Tread uses at most 80% of the total bandwidth, it can use up to 90% of the TCP connection.

The throughput drops slightly after the window size become greater than 10 because the WheelFS server can only handle 10 requests at a time, so when the window size is greater than 10, the WheelFS server begins to drop requests from the client. This limit is a hard-coded to keep the WheelFS server from running too many concurrent threads. When requests start getting dropped, performance suffers because the client must retransmit.

## 5.2 Request Rate Limiting

We evaluate two aspects of our adaptive rate control mechanism for prefetching. First, we show that our rate control mechanism does not hurt prefetching performance. Next, we show that the average prefetch window size used by our adaptive mechanism adjusts well to different bandwidths.

### 5.2.1 Latency experiments

We measure the latency of whole file prefetching with and without rate control for files of different sizes to test how Tread's rate-limiting algorithm affects performance. The setup is exactly like the latency experiments presented in Section 5.1.1. One server writes the file and four clients read the file in turn. The round-trip time between the clients and the server is set at 100ms and the bandwidth between the clients and the server is 6Mbps. For the test without the rate control mechanism, the prefetch window size is set to 3 in-flight requests at a time. From the experiment in Section 5.1.2, we know that having 3 requests in-flight between the server and the client make good use

of the available bandwidth.

We expect the performance of rate-limited prefetching to be slightly worse because the adaptive algorithm causes the window size fluctuate somewhere around the ideal window size, whereas we set the window size to the ideal size for the other test based on our throughput experiments. Figure 5-4 shows the difference in latency between rate-limited prefetching and unlimited prefetching. The experiment shows that rate-

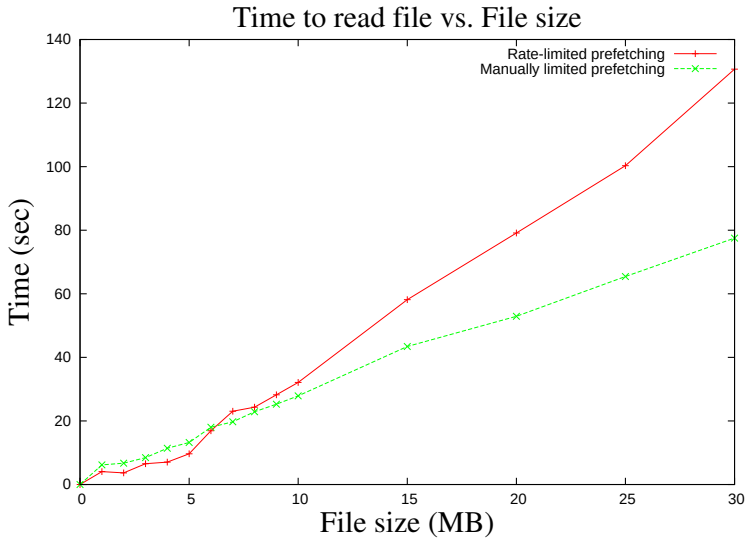



**Figure 5-4: Time to read a file with rate-limited prefetching and prefetching with an ideal window size.** Prefetching with the ideal window size peforms better than adaptive rate-limited prefetching because the adaptive algorithm is conservative in choosing the prefetch window size. The rate-limiting algorithm averaged 3 requests in-flight, which uses the available bandwidth most of the time, but does not give the best performance.

limited prefetching does not perform as well as prefetching with an ideal window size, but can achieve some percentage of the performance. The adaptive rate-limiting algorithm averages a window size of about 3 over the course of the experiment. While this average is the same as the window size used in the non-adaptive test, the window size used by the adaptive algorithm fluctuates, so its performance will not match prefetching with an ideal window size.

### 5.2.2 Bandwidth experiments

Using Emulab, we measure the average prefetch window size chosen by our adaptive algorithm for different bandwidths. We use one WheelFS server and one WheelFS client. We adjust the bandwidth of the link connecting the server and the client and measure the average window size used while the client reads a 10MB file from the server with whole file prefetching. The round-trip time remains at 100ms and the bandwidth varies from 10Mbps to 100Mbps.

We expect the average window size to grow linearly as the bandwidth increases. The window size should adjust to the increase in bandwidth by linearly increasing. Figure 5-5 shows the change in the average window size as the bandwidth increases.

From the experiment, we see that the average window size does increase as the

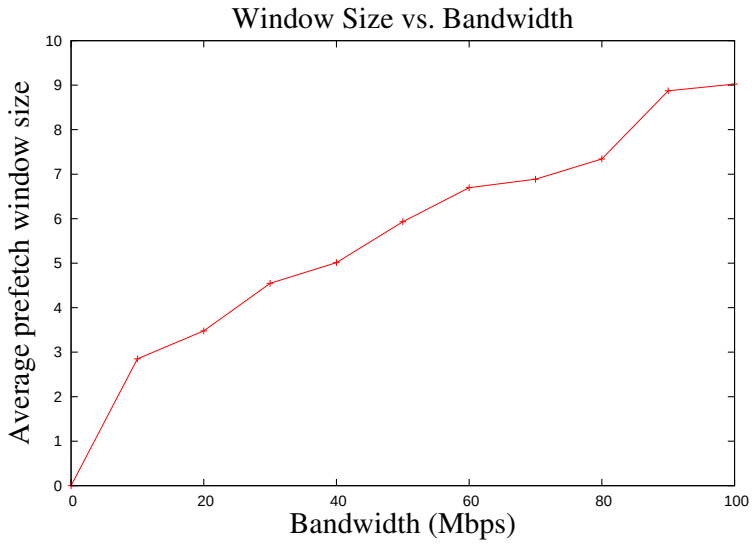



**Figure 5-5: Average prefetch window size chosen by the adaptive rate-limiting algorithm for varying bandwidths.** The average window size chosen by the adaptive algorithm increases linearly as the available bandwidth increases, showing that the adaptive algorithm adjusts the prefetch window size accordingly to make use of available bandwidth.

bandwidth increases but at a slow rate of about 1 request per 30Mbps of bandwidth. This slow increase is because Tread's adaptive algorithm is conservative in increasing

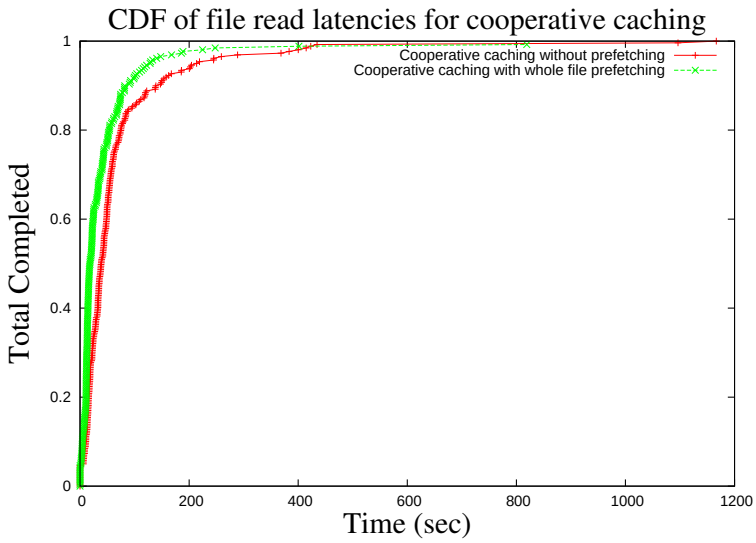
the window size. Tread will only increase the window size if it does not lead to a significant increase in the latency of block requests to the server. The goal is not to maximize throughput, but maximize throughput while keeping the latency of block requests low. We chose this policy because if prefetching begins to increases the latency of block requests at the servers, prefetching will start to hurt the performance of application requests. So while Tread may be able to get better throughput with a bigger window size, Tread always chooses to keep the latency of block requests low, rather than increase throughput, to ensure that prefetching is not affecting applications.

## 5.3 Cooperative Caching

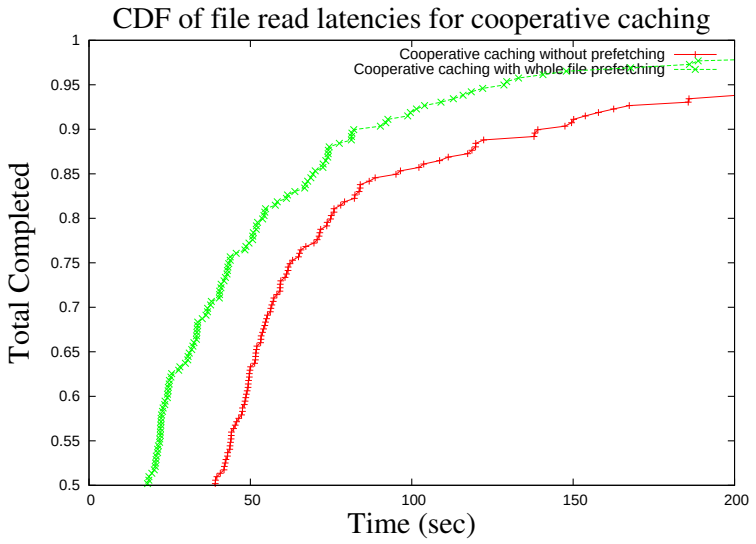
To test the benefit of prefetching with cooperative caching, we run a wide-area experiment with a large number of Planetlab nodes simultaneously reading a file. We use one server located at MIT and about 270 Planetlab nodes as clients. We measure the latency for each client to read a 10MB file with no prefetching and with whole file prefetching. All clients use cooperative caching and call `read()` on the entire file at once. We chose to test no prefetching and whole file prefetching because whole file prefetching is easy to control with the semantic cue and, as shown in Section 5.1.1, read-ahead prefetching performs similarly to whole file prefetching when clients are reading whole files. For applications that sequentially read entire files, the default performance of WheelFS with cooperative caching turned on should be close to the performance of cooperative caching with whole file prefetching.

We hope to find that clients finish faster on average using prefetching with cooperative caching. With prefetching, clients prefetch the file in random order rather than reading the file linearly, which should increase the opportunities for sharing and reduce the number of blocks that nodes have to wait to fetch from the server. In addition, using prefetching allows clients to download from multiple peers at once. Figure 5-6 gives a CDF of latencies for cooperative caching with and without prefetching.

The cooperative caching experiment shows that clients finish fetching files faster




(a) Full view of CDF




(b) Zoomed view of CDF.

**Figure 5-6: Full and zoomed view of CDF of latencies for nodes simultaneously reading a file using cooperative caching with and without prefetching.** The experiment shows that nodes are finished reading the file faster when using prefetching with cooperative caching. The average latency for reading the file 68 seconds without prefetching and 38 seconds with prefetching. This result shows that there is an advantage to randomizing the prefetching order and making requests to several clients at once.

with whole file prefetching. The average latency for cooperative caching without prefetching is 68 seconds, while the average latency with prefetching is 38 seconds. Without prefetching, half of the nodes finished in 39 seconds, but with prefetching, half of the nodes were done in 18 seconds. There are always stragglers with cooperative caching, but the stragglers finish faster when prefetching is used. Without prefetching, 99% of reads finish within 435 seconds and 100% of reads finish in 1167 seconds. With prefetching, 99% of reads finish within 248 seconds and 100% of reads finish in 819 seconds. The results of this experiment show that prefetching with cooperative caching reduces the average time for each node to read the file as well as the overall time needed to distribute the file to all of the nodes.

For a comparison of WheelFS and BitTorrent, refer to the WheelFS paper [18].

# Chapter 6

# Related Work

The optimizations used in Tread to improve wide-area file system performance take inspiration from many previous systems. This chapter covers the related work for each of Tread's different prefetching policies.

## 6.1 Prefetching in File Systems

Prefetching is used widely in traditional on-disk file systems. Shriver, et al. [17] sum up the benefits of prefetching for file systems succinctly:

- There is a fixed cost for performing a disk I/O, so fetching more data at a time amortizes the overhead of reading from disk.
- Modern disks often have a disk cache and perform prefetching themselves anyway.
- With more requests, the disk controller can do a better job of ordering requests to minimize seek time.
- If the application performs substantial computation, prefetching allows the file system to overlap computation and I/O.

The bulk of the previous work on file system prefetching focuses on determining *what* to prefetch. Many on-disk file systems in use today use sequential read-ahead as a simple adaptive prefetching technique because it is a simple mechanism and literally

reading ahead on the disk is cheap, if not free. Tread applies read-ahead to WheelFS in the context of a distributed file system where the underlying storage layer is not a local disk, but a remote one. There are more sophisticated schemes for predictive prefetching that take into account the usage patterns of the application using hidden Markov models or neural nets or other predictive algorithms [12, 13, 10, 11]. These systems use an algorithm to predict the applications future file accesses based on past file accesses.

Another technique, called *informed prefetching*, uses hints from the application [15, 16] to decide what data to prefetch, with the assumption that better file system performance can be achieved with information from the application. Informed prefetching meshes well WheelFS's semantic cues. Tread applies informed prefetching to both file and directory prefetching by allowing applications to control these two types of prefetching with semantic cues.

Some of the same prefetching techniques have been applied to cluster file systems. Ceph [20] aggresively prefetches meta-data for better performance and Scotch [6] uses informed prefetching to mask the latency of network and disk. GFS [5] does not perform predictive prefetching, but highlights the use of larger block sizes as a sort of static read-ahead to achieve better throughput.

## 6.2 Cooperative Caching

The technique of randomizing the prefetch order of blocks when using cooperative caching is inspired by other file distribution services that use cooperative caching. In particular, Shark[1] is a distributed file system that uses cooperative caching with random ordering for fetching file chunks.

The goal of Shark is the same as the goal of cooperative caching in WheelFS: to improve distributed file system performance for simultaneous file downloads. Shark recognizes the advantage of using a distributed file system that can be transparently mounted as a local drive. Shark still uses a centralized server but reduces the load at that server for reads by having clients fetch blocks from each other. Shark also allows

users to be mutually distrustful and still share data.

A Shark client performs a read by first asking the server for a token for the whole file and each block in the file. The tokens are a cryptographic hash of the data; the tokens serve both as permission for the reader to access a particular file and as a way for the reader to check the integrity of the chunk it gets from a proxy.

Shark's evaluation showed that Shark performs better when clients randomize the order in which they fetch file chunks. The authors found that without randomization, the fastest clients all fetched from the server and the rest of the clients fetched from the fastest clients. This phenomena may have been due to Shark's inter-proxy negotiation mechanism, but even without the negotiation mechanism, it is logical that synchronization occurs when clients all fetch the file in the same order.

## 6.3 Adaptive Rate Control

Transfer rate control is ubiquitous in the wide-area network. The Transport Control Protocol (TCP) includes one of the best known and most widely used dynamic rate scaling algorithms [19]. TCP is a protocol for reliable delivery of a stream of bytes across a network connection. TCP uses flow control to ensure that the sender is not sending too fast for the receiver to process and congestion control to ensure that the network does not experience congestion collapse.

Tread's rate-limiting algorithm tries to achieve some of the same goals. Since WheelFS is already using TCP, clients do not have to worry about reliably sending requests to the servers. But Tread still needs to control the flow of requests to the servers, so clients know that they are not sending requests faster than the servers can handle. Tread also need to avoid congestion at the server due to prefetching, so application requests can get processed quickly. Unlike TCP, our adaptive algorithm does not receive feedback from the sender, so we must use other metrics for determining how to adjust the flow of prefetch requests.

Coblitz [14] is wide-area large file distribution service that performs its own rate control for block requests. Coblitz is a content distribution network (CDN). CDNs

use a large number of nodes spread over the wide-area as a cooperative cache for web content. When users request a file cached in the CDN, a cache node serves the file instead of the origin server. CDNs reduce the load on the origin server further by having cache nodes use cooperative caching to share data with each other. CDNs generally serve large numbers of small files, so they are not well suited to serving large files. Coblitz solves this problem by splitting large files into separate smaller files. When a Coblitz node fetches these smaller files from other nodes, it dynamically scales the number of file requests it sends out at a time. The algorithm adapts by increasing the window size every time a small file request finishes faster than the average and decreasing the window size when a file request to another node fails.

# Chapter 7

# Conclusion

This thesis presented Tread, a prefetcher for the WheelFS distributed file system. Thread shows that prefetching can be applied effectively in different ways to improve the performance of a wide-area file system. Thread uses prefetching to reduce the latency of reading a file, increase the throughput of file transfers and improve sharing for cooperative caching. Thread includes two prefetching policies used by WheelFS by default: read-ahead prefetching and rate-limited prefetching. Thread also includes three prefetching policies that are controlled by the applications: whole file prefetching, directory prefetching and randomized prefetching with cooperative caching.

## 7.1 Future Work

There are two optimizations that we would like to explore for Thread and WheelFS in the future. The evaluation of Thread showed the importance of having several in-flight requests between the clients and servers for performance. Currently, the WheelFS clients do not send requests in parallel to the WheelFS server, so requests made by the application may require several synchronous requests to the servers. In the future, we want to explore adding parallelism to the WheelFS clients to improve the performance of reading without prefetching. In order to take advantage of added parallelism in the WheelFS client, we would have to prevent the operating system from breaking up application requests before sending them to WheelFS. This work would also include

exploring adaptive rate controls for the WheelFS client and the interaction of the client's rate-limiting algorithm with Tread.

The second optimization is an improvement on Tread's adaptive rate-limiting algorithm. Currently, the RPC layer hides some information that would be useful to Tread's adaptive algorithm. For example, when the WheelFS server is overloaded, it will drop requests. The RPC system takes care of automatically retrying, so Tread does not see anything other than increased latency. In the future, we may want to adapt the RPC layer to provide more information, so Tread can make more better decisions about the rate at which to send prefetching requests. This information could either be given to the server to be sent back to the client or delivered directly to the client. The RPC layer could inform the client when requests to the server fail or inform the server of number of requests being currently processed. With more information, the adaptive algorithm could be improved to scale better for different bandwidth constraints.

# Bibliography


[1] Siddhartha Annapureddy, Michael J. Freedman, and David Mazières. Shark: scaling file servers via cooperative caching. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design & Implementation*. USENIX Association, 2005. 6.2

[2] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*. ACM, 1991. 1.3

[3] Andy Bavier, Mic Bowman, Brent Chun, David Culler, Scott Karlin, Steve Muir, Larry Peterson, Timothy Roscoe, Tammo Spalink, and Mike Wawrzoniak. Operating system support for planetary-scale network services. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation*. USENIX Association, 2004. 5

[4] Richard Durstenfeld. Algorithm 235: Random permutation. *Communications of the ACM*, 7(7), 1964. 4.2

[5] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. *SIGOPS Operating Systems Review*, 37(5):29–43, 2003. 6.1

[6] G. A. Gibson, D. Stodolsky, F. W. Chang, W. V. Courtright, C. G. Demetriou, E. Ginting, M. Holland, Q. Ma, L. Neal, R. H. Patterson, J. Su, R. Youssef, and J. Zelenka. The scotch parallel storage systems. In *Proceedings of the 40th IEEE Computer Society International Conference*. IEEE Computer Society, 1995. 6.1

[7] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002. 1.1

[8] James Griffioen and Randy Appleton. Reducing file system latency using a predictive approach. In *Proceedings of the 1994 Summer USENIX Technical Conference*, 1994. 3.3.2

[9] V. Jacobson. Congestion avoidance and control. *SIGCOMM Compututer Commununication Review*, 25(1):157–187, 1995. 3.3.1

[10] Thomas M. Kroeger and Darrell D. E. Long. Predicting file system actions from prior events. In *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, 1996. 6.1

[11] Hui Lei and Dan Duchamp. An analytical approach to file prefetching. In *Proceedings of USENIX Annual Technical Conference*, Anaheim, California, 1997. USENIX Association. 3.3.2, 6.1

[12] Tara M. Madhyastha and Daniel A. Reed. Exploiting global input/output access pattern classification. In *Proceedings of the 1997 ACM/IEEE conference on Supercomputing*, San Jose, CA, 1997. ACM. 6.1

[13] Tara M. Madhyastha and Daniel A. Reed. Input/output access pattern classification using hidden markov models. In *Proceedings of the 5th workshop on I/O in parallel and distributed systems*. ACM, 1997. 6.1

[14] KyoungSoo Park and Vivek S. Pai. Scale and performance in the coblitz large-file distribution service. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design & Implementation*. USENIX Association, 2006. 6.3

[15] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Copper Mountain, Colorado, United States, 1995. ACM. 6.1

[16] R. Hugo Patterson, Garth A. Gibson, and M. Satyanarayanan. A status report on research in transparent informed prefetching. *SIGOPS Operating Systems Review*, 27(2):21–34, 1993. 6.1

[17] Elizabeth Shriver, Christopher Small, and Keith A. Smith. Why does file system prefetching work? In *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, 1999. 1.3, 6.1

[18] Jeremy Stribling, Yair Sovran, Irene Zhang, Xavid Pretzer, Jinyang Li, M. Frans Kaashoek, and Robert Morris. Flexible, wide-area storage for distributed systems with wheelfs. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, 2009. 1.1, 2, 5.3

[19] B. L. Tierney. TCP tuning guide for distributed applications on Wide-Area networks. *USENIX & SAGE Login*, 2001. 5.1.1, 6.3

[20] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: a scalable, high-performance distributed file system. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2006. 6.1

[21] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *Proceedings of*

*the 5th Symposium on Operating Systems Design and Implementation*. USENIX Association, 2002. 5

[22] Ming Zhao, Jian Zhang, and Renato Figueiredo. Distributed file system support for virtual machines in grid computing. In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*. IEEE Computer Society, 2004. 1.2