Abstract

Windows SFTP File System Driver, a high performance system for accessing files stored on a SFTP server running over a Secure SHell transport as a locally accessible name-space is described. In contrast to almost all existing solutions where a file manager utility program is utilized that is both inefficient and convoluted in operation.

An investigation is done into implementing a network file system driver (or 'redirector') under contemporary Microsoft operating systems with particular emphasis on FIFS (a Framework for Implementing user-mode File Systems). Possibilities for predictive caching where files are prefetched in advance of their expected access are examined and a novel solution proposed using a weighted best of algorithm that takes into account inter-operation lag. The Secure File Transfer Protocol is examined in detail for its suitability as a redirector medium and failings are identified and solutions proposed.

A working implementation is constructed using a multi-module design with an extensible framework for SFTP request and packet handling. The proposed predictive prefetcher is also implemented and data structures and algorithms are discussed before being tested in artificial setups resembling real world usage patterns and performance figures examined and compared to the existing solution on the market.

Windows SFTP file system driver

Table of Contents

Background Chapter

The need for a remote file system driver for Windows

Existing SFTP/SCP clients for the Windows platform mostly rely on a convoluted process to access files. First a user must navigate to their file on the SFTP/SCP server, download the file to a temporary location, perform their required actions on the file and then remember to upload the file back to the SFTP/SCP server. As well as being a very time consuming process for the user it also means increased bandwidth usage as the entire file must be re-uploaded when sometimes only a subsection of the file is changed, for example a line added to the end of a text file.

Most users will be familiar with the concept of networked file systems, that is, remote file system spaces mapped to a user's local file system space. This approach has many benefits and removes the inefficiencies of a specialist client interface, for example one benefit is it allows the use of generic system file search utilities.

The SSH architecture

The SSH or Secure Shell protocol is a way of providing a secure login and other secure network services over an insure network [5].

The architecture can be divided into three parts:

- The user authentication layer. Various methods are supported from keyboard-interactive to public key based systems.
- The connection layer allows multiple streams of information to be set up once the authentication layer is connected successfully. For example, a stream may be set up for SFTP and another for shell terminal access.
- The transport layer protocol underlies the authentication and connection layer by setting up a reliable, possibly compressed and encrypted channel for packet communication. Additionally, it handles server authentication and verification. It normally runs on top of the TCP/IP protocol as this provides it with a reliable data stream.

File management in SSH

When accessing remote files over an SSH channel there are two mainstream protocols that can be used. The first is SFTP and the second is SCP, however there are a number of distinct disadvantages of the older SCP system that make it unsuitable for a high performance file system.

Feature	***SCP***	***SFTP***
Packet confirmations	None. There is no guarantee that an operation was completely successfully.	Supported
Block access	Not supported. Only the whole file may be accessed. Not a subsection of it.	Supported
Maximum file size	4GB	64 bit file sizes.
Recursive directory deletions	Supported via a single	Multiple commands needed

Feature	*SCP*	*SFTP*
	command	for each directory to be deleted
Server support	Since version 1 (operating system dependent)	Since version 2

Table 1. (adapted from the WinSCP documentation [2])

The lack of fine grained access of files by the reading and writing of particular blocks of a file is a serious detriment to performance with SCP, this means whenever a file is accessed the whole file needs to be sent over the network link. It also presents problems in caching as only the whole file can be cached. The lack of operation confirmation in SCP and restricted file sizes would additionally make the file system unreliable and limited as well as poorly performing. For these reasons the use of SFTP is a necessity despite the fact that by using it means restricting the driver's availability to version 2 servers.

A closer look at the SFTP protocol

The SFTP protocol utilizes a simple request/reply model. Clients after the initial handshake may then send a request packet of the following form:

Length (uint32)	*Type (byte)*	*Request-ID (uint32)*	*Param 1*	*Param 2*	*Param n*
Size of packet	Comma nd	Packet reference	Command specific data		

Table 2.

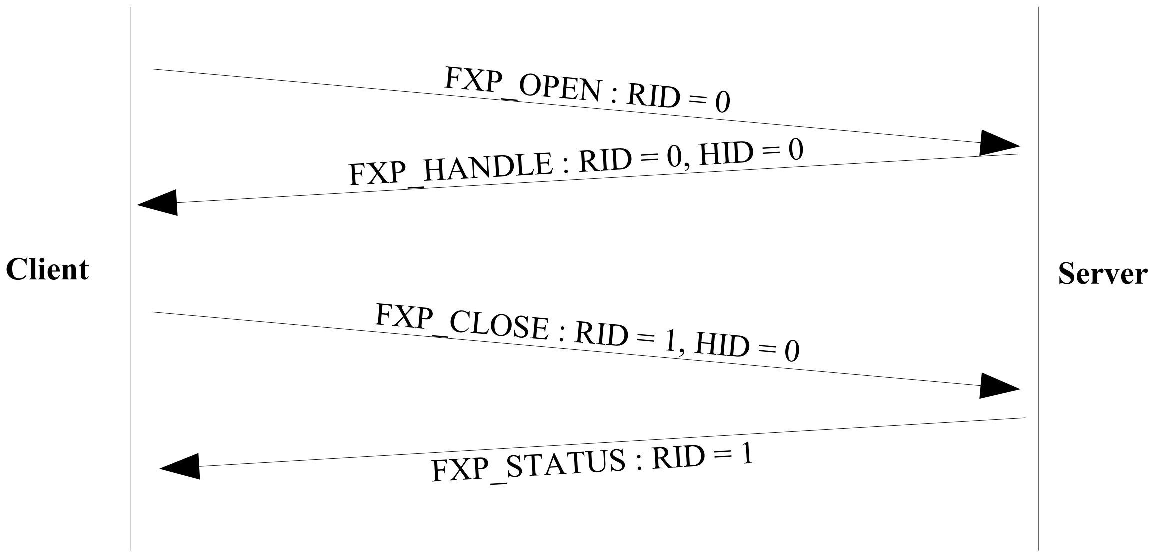


Figure 1.

The server in most instances then will reply with a packet with the same request-ID and provide an error message or a file/directory handle. This allows multiple concurrent file requests from the client which the server must fulfill and in any order it desires.

Figure 1 illustrates an example SFTP transaction whereby a file is opened (FXP_OPEN) is sent as the packet command, a handle is returned by the server and then the handle is closed where upon the server returns a status packet so that the client may recognize it was successfully completed.

The SFTP protocol contains a range of commands that allow it to implement the most common file system functionality. The different commands can be listed in five groups.

Protocol control

`SSH_FXP_INIT and SSH_FXP_VERSION`	`session setup`
`SSH_FXP_STATUS`	`server->client error codes`

Stream control

`SSH_FXP_HANDLE`	`file/directory descriptor`
`SSH_FXP_CLOSE`	`close descriptor`

File I/O

`SSH_FXP_OPEN`	`open file`
`SSH_FXP_READ and SSH_FXP_DATA`	`read file data`
`SSH_FXP_WRITE`	`write file data`
`SSH_FXP_FSETSTAT`	`set file attributes`

Directory I/O

`SSH_FXP_OPENDIR`	`open directory`
`SSH_FXP_READDIR and SSH_FXP_NAME`	`get directory entries`

Directory management

`SSH_FXP_MKDIR`	`make directory`
`SSH_FXP_RMDIR`	`remove directory`
`SSH_FXP_REMOVE`	`remove file`
`SSH_FXP_RENAME`	`rename file`
`SSH_FXP_SETSTAT`	`set file attributes`
`SSH_FXP_STAT and SSH_FXP_ATTRS`	`get file attributes`

Existing Windows SFTP File system Clients

SftpDrive

This recently released (December 2005) commercial product is distributed by Magnetk LLC. Little is known of its design however it appears to makes use of a Microsoft IFS kernel space file system driver (SftpDrive.sys) that implements SFTP functionality as well as the SSH architecture. It interfaces with a user space encryption library (OpenSSL) to provide the SSH encryption primitives.

Unix and Linux SFTP File-system Clients

LUFS SFTP module

The LUFS creator sums up the functionality of this system as follows:

> "LUFS is a hybrid user space file system framework supporting an indefinite number of file systems transparently for any application. It consists of a kernel module and an user space daemon. It delegates most of the VFS calls to a specialized daemon which handles them."[6]

An SFTP module then runs a component of the LUFS framework which utilizes the OpenSSH library for the SSH layer. It implements the full SFTP protocol and will drop down to using SCP if SFTP is not available on the server.

SSHFS (FUSE module)

The FUSE system implements a lightweight kernel space driver that interfaces with the /dev/fuse descriptor which is then used to communicate with the libfuse library which in turn interfaces with a number of FUSE file system modules.

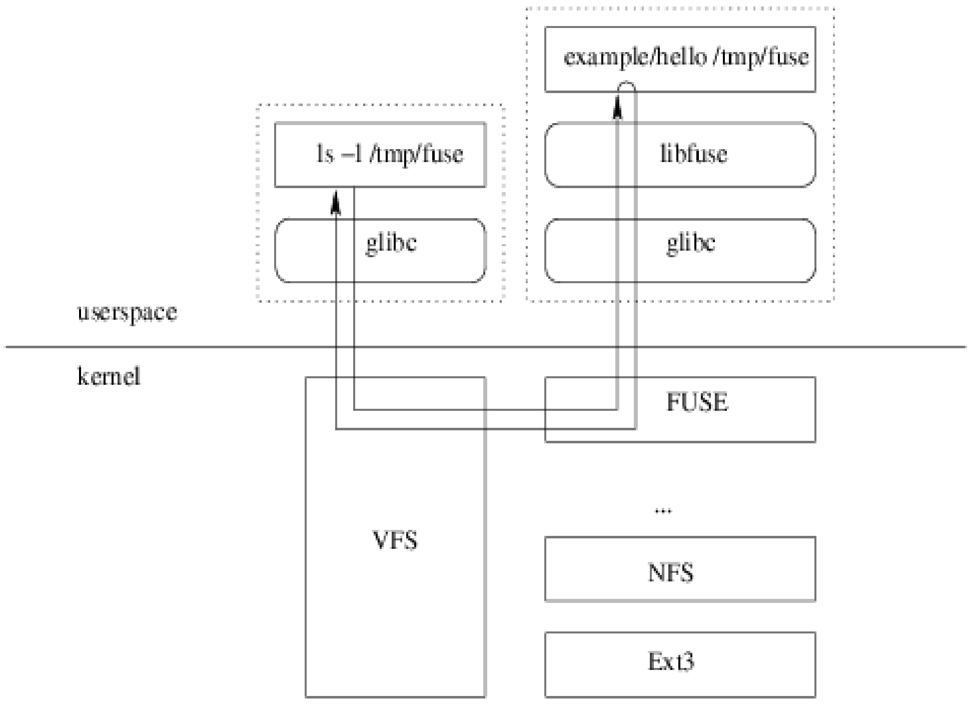


Figure 2 [7]

The FUSE SFTP module is based on the LUFS module, however, it adds a number of improvements in performance by caching directory contents and allow multiple concurrent requests to the SFTP server. Additionally, it supports full 64 bit file sizes.

SHFS (Linux kernel module)

This remote file system implementation is not a true SFTP or SCP client, rather it relies on a platform specific server handle daemon (a Perl script) or runs shell commands to get file system information (such as ls) and other shell commands to read files (e.g. dd). It does however provide good performance through the use of write-back caching, directory caching and read caching.

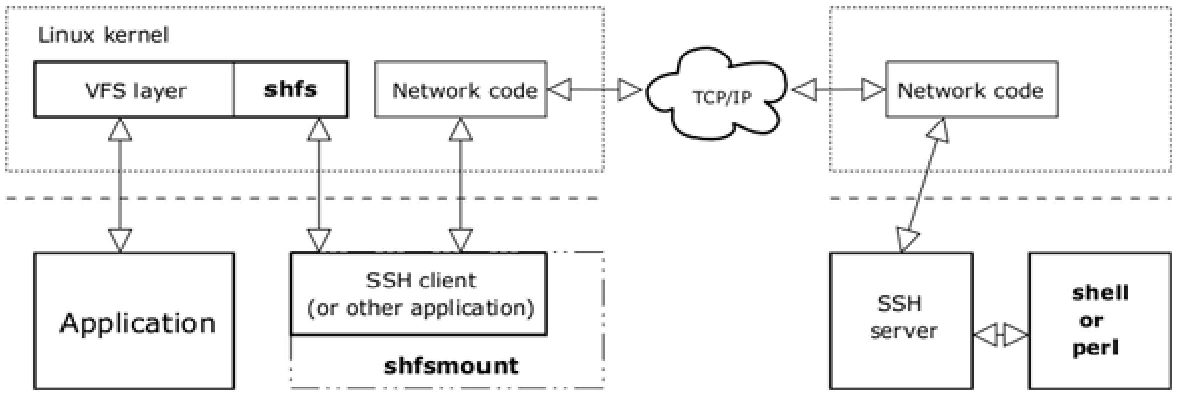


Figure 3. [8]

SSHFS [9]

This project is the beginnings of a hybrid kernel/user space SSH file system client. As seen with other Linux SSH file system clients it uses a kernel space driver that interacts with a user-space program. SSHFS uses shared memory for this purpose. Also like others it uses the OpenSSH program for the SSH layer.

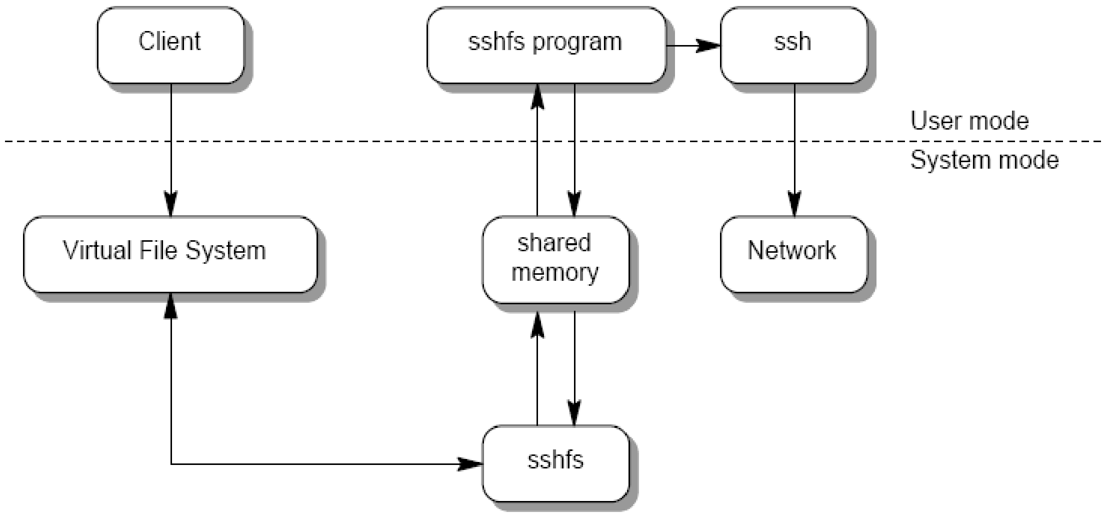


Figure 4, [9]

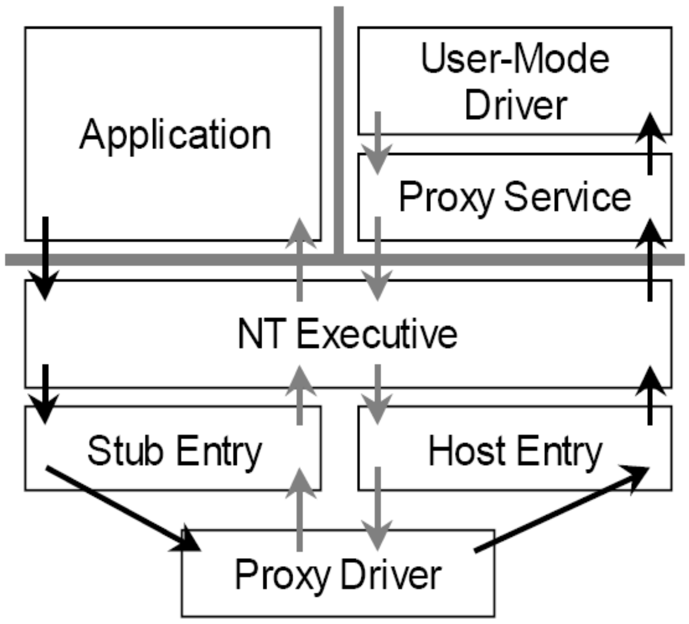
Design approach – Windows Explorer Name-space Extension

On Windows there are a number of ways of mapping remote file system spaces to the user's local name space. One popular approach is via a Windows Explorer Name space extension, the same method Microsoft uses to provide access to a number of non local or virtual file systems like PocketPC PDA file systems, this however was discounted as it did not provide a "true" file system name space as the name space was only provided to the Explorer application, not to any programming running on the system.

Design approach – the Gale Hunt Users-pace Framework

The Hunt User-mode device driver framework [10] is an experimental system developed for Windows NT 4.0 systems that allows programmers to write user mode drivers of almost any type.

Figure 5. [11]



It works by installing a Proxy Driver in the system which then dynamically creates stub device drivers when requested by a user-space Proxy Service. User mode drivers may now communicate via the Proxy Service to the Host Entry and to the Proxy Driver which translates the user-space commands to the Stub Entry virtual object.

One implemented user space driver was 'ftpfs' which in the paper Hunt [10] explains allows the user to:

> “...mount a remote FTP server as a local file system. Incoming IRPs are converted to outgoing FTP requests using the WinInet APIs.”

Performance in the system was high, indeed in some cases as Hunt explains, “by exploiting full access to user-mode resources, user-mode device drivers can achieve better performance than kernel-mode drivers.” However, although this system seems attractive it still requires a thorough understanding of the kernel-space IFS driver framework and would require the updating of the Proxy Service system as it was written for the NT 4.0 operating system and would not operate on newer versions such as Windows 2000 and XP.

Design approach – kernel file system driver

Kernel mode file system drivers are developed using Microsoft's Installable File System (IFS) interface. All file system drivers must closely integrate with the NT kernel's I/O manager and cache/memory manager. Compared to other kernel file system interfaces, for example Linux's VFS architecture it is considerably more complex and time consuming to implement a fully featured driver. Indeed, an NT file system driver is one of the hardest kernel drivers to write as “while traditional device drivers interact mostly with the I/O manager, a file system driver must also engage in complex interactions with the cache manager and the virtual memory manager while satisfying I/O requests” [1].

	Line count	***Character count***	***File count***
Linux smbfs kernel module	7,641 lines	186,228 bytes	15 files
Microsoft SMBMRX kernel driver	52,115 lines	1,750,078 bytes	54 files

Table 3.

Table 3. clearly illustrates the relative complexity of the differing implementation platforms for the two drivers that offer similar end user functionality, that is for accessing SMB networked file systems.

Design approach - FIFS

The FIFS architecture was developed by Damilo Almeida of the Massachusetts Institute for Technology in 1999 to try and create a more comfortable platform for the development of experimental and rapidly prototyping of new file system drivers for the Windows NT operating system.

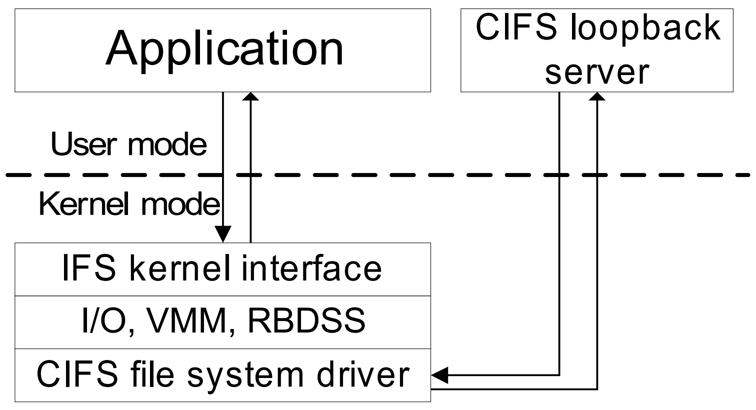


Figure 6.

The FIFS system is based on the exploitation of Window's inbuilt SMB (CIFS) network file system driver. It implements a local user-space SMB server that is linked to a number of FIFS modules that provide access to some other file system space. This way a user can have a SMB mapped file name-space on their machine that is indirectly mapped to another file system space.

Through the use of a dispatch table interface it can utilize a wide range of Windows' file system functionality. Additionally it supports file system filter layers that can fit between the server and the file system modules. For example the FIFS distribution includes a filter named FSMUNGE that splits directory names into their constituent parts so that the filter or file system driver underneath does not have to parse directory names itself.

FIFS – analysis and viability

The viability of the FIFS architecture was verified by Rimer in his implementation of the Secure File System (SFS) after he successfully achieved the majority of his goals for it on the Windows NT platform [11].

It in its present form the FIFS architecture has a number of restrictions, some of which are addressed by Rimer while others must be analyzed and overcome for an SFTP driver, the remaining restrictions are ignored in this thesis as they do not apply to our goals.

1. Incompatibility with Windows 2000 and XP

   The emergence of these two platforms in recent years as the de-facto standard with Windows XP by itself running on 72% of the world's computers by some estimates [3] makes it a necessity for FIFS to support these platforms if it is to fulfill its goal of being generally available for current Windows systems.

2. Reliance on proprietary closed Microsoft libraries

   FIFS in its current state relies on various SMB header files that are distributed with the commercially available Windows Device Development Kit (WinDDK) . This is a hindrance in the development of a fully free driver. As the only usage of these files is for their structural descriptions of SMB packets and that these packet formats are publicly documented it should be viable to replicate their functionality in open source headers.

3. No authentication of clients

   The FIFS system in its current state allows any user to on the local machine to connect to the loop back server and access a loaded file system. This is a serious restriction in a multi-user OS like Windows XP, in the context of this thesis it would mean any user would be allowed to access a SFTP mounting created by any other user. As all mountings must be previously authenticated with a stored password or key this means all users have complete access to the secure file spaces of others.

   This thesis will address this flaw with the design and implementation of a “pass-through” authentication system introduced in [1].

4. No IOCTLs for file system modules

   The Windows NT architecture allows special I/O control commands on files to implement file system specific functionality. The SMB protocol also allows IOCTLs to be communicated through its protocol. The FIFS system, however, fails to make use of this operating system feature and does not allow file system modules to implement it. In this thesis this aspect of FIFS is not considered for improvement as the SFTP protocol does not provide any special IOCTLs.

5. No locking

   Although Windows NT and the SMB protocol allow the use of “opportunistic locks” these are not implemented by the FIFS server as the version of the SMB protocol it implements does not support this functionality. Locking is a very useful tool for the goal of high caching performance and the SFTP protocol [12] does support file and byte range locking. However, it is unfortunate that the one of the most popular [4] Unix SSH/SFTP server implementations only provides up to version 3 of the SFTP protocol whilst file locking support was added in version 5 and byte range locking in version 6. As this particular implementation accounts for 90% of SSH daemons it would not be appropriate to use a protocol version higher than 3.

6. No caching

   The FIFS distribution makes no attempt at any kind of caching. Rimer's SFS module implements an FSCACHE filter which provides basic caching support of directory contents and read and write-through file caching.

   To provide a high performance SFTP client better caching needs to be implemented. However, due to lack of locking support under the SFTP protocol there are some difficulties in this area. Pertinent questions are:

   1. How is it possible to determine when a directory entries cache is out of date?
   2. How would the driver know if its cache of a file page is out of date?
   3. What happens when a file page is written back to server when that file has been modified by another process?

7. No support for case sensitive file names

   In SFS/NT [11] Rimer attempts to overcome this issue which has its roots in the fact that despite Windows NT allowing different case characters in file and directory names they are all treated by the I/O subsystem and the SMB protocol as being case insensitive. This presents a complication for file systems that are case sensitive such as SFTP and SFS. Rimer's solution is to map the case insensitive name to a case sensitive name by looking up the case insensitive name in the parent directory in a case insensitive manner and then use the first match. If there is more than one match then access to the file is denied as it is impossible to determine which file is truly requested.

   Rimer introduces a possible improvement to overcome this problem that requires the addition of a suffix to file names designating each case name variation of a file when given as a directory listing to a user. Thus, when a mangled name is accessed it may be mapped back to the original case sensitive name.

8. No symbolic link support

   Windows does not provide any support for symlinks like some operating systems like Linux does. As such FIFS does not attempt to provide an interface for them. However, as SFTP may be used on file systems that use symlinks the protocol exposes an interface to control and manipulate them. A Windows client need not concern itself with manipulating symlinks, it would only be necessary to follow a symlink to its final target. This could be achieved through the addition of a new filter in the FIFS framework.

Predictive prefetching

The fundamental idea of prefetching, where an agent predicts future file accesses and places them in the cache in advance of the user's access of them, is something that many people have attempted to design with a wide variety of ideas being put to use in an effort to create accurate prediction systems.

Predictive prefetching is founded on the presumption that in many cases of file reads there exists a clear pattern of accesses that can be inferred from historical tracking of accesses. For example, take the case of the 'make' command being run on a project involving a number of source files and a number of header files.

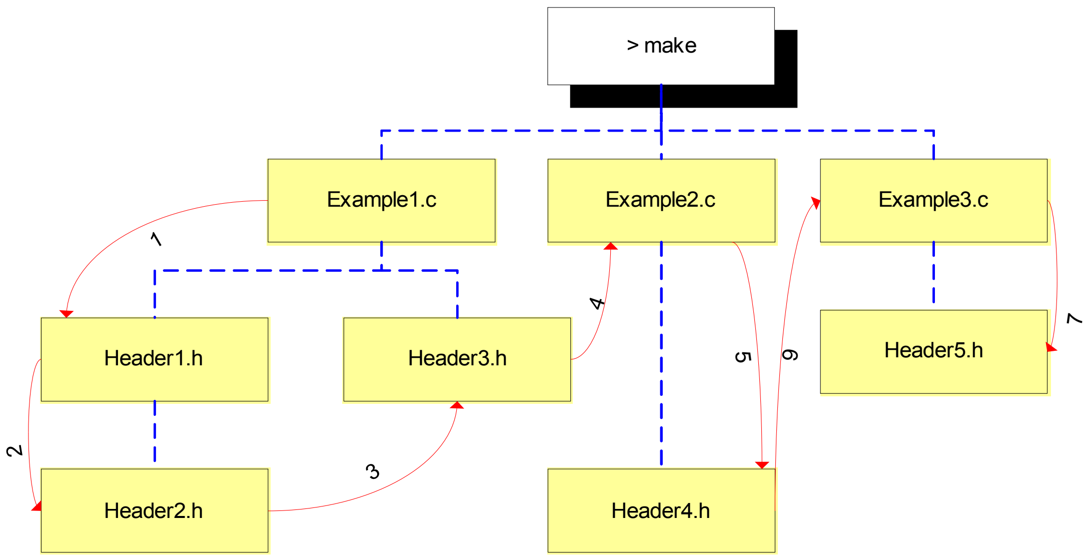


Figure 7.

Logically, their relationship can be represented by a tree like structure illustrated in figure 7, there also exists a consistent pattern of accesses here indicated by the red arrows. Predictive prefetcher patterns may also arise in a number of other scenarios:

- Executing applications that have to load configuration files and libraries.
- Opening a project into an IDE requiring many different files.
- A user browsing a folder may often traverse the file structure in a familiar way.
- Batch files and scripts often impose order on file executions.

This kind of prefetching is of particular benefit to situations where there is a very high latency, for example, Internet based file systems. In these types of situations the client can fetch two or more files at the same time with minimal link time cost but with theoretical local access time advantages.

It might be expected that in multi-process environments that access patterns would be obscured by overlapping accesses thus making accurate prefetches impossible, however Griffioen through the use of trace logs [13] finds that even in multi-process systems that one file access follows a previous access with 94% accuracy.

Various methods of prefetching have been proposed. Griffioen in [13] investigated the use of 'Probability Graphs' where nodes represented files and arcs represented the probability of another file being the successor of a file. Results were impressive for small cache sizes with cache miss rates reduced by up to 150%. In 'The Design of the See Predictive Caching System' Kuenning introduces the concept of 'semantic distance' and discusses some of the implications of various measurements of determining 'distance' between files before implementing a system that predictively clusters distributed files for offline access on mobile machines.

Lei and Duchamp in [16] use access trees and describe a method of calculating the similarity of trees before using this similarity index to prefetch the tree.

Amer et al describe their 'Recent Popularity' or Best-K-of-M predictor which attempts to use dynamic information to provide a stable prediction choice by recording the last M successors for a file and choosing a prediction if it occurs at least K times. They compare their predictor against the older 'Noah' predictor described in [20]. The 'Noah' predictor builds on the 'Last-Successor' predictor where the last successor of a file is used as the predicted successor to add a notion of stability by recording the last prediction and only offering a new prediction if the last successor has been consistent for a certain time period (the stability parameter)

The very high cache hit rates of the Best K-Of-M predictor described in [13] shows how a relatively simplistic system can outperform some of the more complex techniques described by Kroeger in [13]. In the case of prefetching it seems as though recording the last N successors of a file is all that is necessary.

An entirely different approach to prefetching is delegate prediction to user level programs that are required to inform the prefetch manager of their future accesses. This technique does guarantee high accuracy in some cases however it does not attempt to predict more dynamic events that are closely tied to user behaviour.

Requirements

In designing a large project of this type, there is a need to create a list of requirements of what is being aimed to be achieved. This project naturally lends itself to be split into various modules, loosely based on their distance through the subset of the protocol stack, from the SMB FIFS layer to the FIFS SFTP interface to the SFTP application protocol to the SSH secure transport layer. Additionally, separate consideration will be given for the prefetching system.

LibSSH

To perform SSH transport tasks a library named LibSSH should be created that will be used by SFTP library. Due to the large size of the SSH protocol the possibility of using an existing SSH library will be investigated for the design phase rather than necessarily creating a complete implementation. However, in either case the library should be generic and not tied to the SFTP protocol in any way. It will have the following requirements:

1. Provide the secure socket transport
2. Connect to SSH2 servers with a secure encryption layer
3. Allow key based authentication and password authentication
4. Provide server verification
5. Ability to start SSH2 subsystems
6. Data channel for subsystems to write blocks of data synchronously
7. Data channel for subsystems to wait for incoming data and buffer data and make data available when requested
8. Multiple connections and data channels

SFTP library

The SFTP library will interact with the SSH library to provide an easy to use method of accessing an SFTP server through a fully object orientated interface:

1. Provide fully standards compliant, reliable access to SFTP all revisions of current SFTP servers
2. Allow multiple outgoing requests and manage associated incoming data concurrently
3. Object orientated interface, based on a directory model that allows operations like list entries and make folder/file
4. Directory entries that can be opened, closed, read, written
5. Ability to get and set attributes for entries like file size and permissions
6. Abstracted logging interface
7. Allow easy addition of new operations
8. Standardized, extensible SFTP packet framework

FIFS framework

The FIFS framework has been verified as working by the implementation of the experimental NFS file system driver by the original developers of the system, however, as discussed, it requires a number of changes:

1. Configured for FIFS SFTP requirements

2. Operation under Microsoft Windows 2000 and Microsoft Windows XP
3. Implement additional SMB requests as required

FIFS SFTP module

The SFTP module is the bridge between the FIFS framework and the SFTP library and will require careful design for optimum performance. Additionally, it must interface with the caching and prefetching system.

1. Implement all file system request handlers in the dispatch interface
2. Implement file system descriptor interface
3. Provide easily configurable interface to server settings
4. Handle errors or network problems gracefully

Predictive caching system

The caching system should be based on a traditional object/page cache but introduce a prefetcher layer. Consideration will be given in the design section to creating an innovative and higher accuracy predictor.

1. Retain file data and directory listings
2. Customizable caching parameters
3. Prefetch files and directories in advance of their access
4. Discard inconsistent cache pages as necessary
5. Efficiently replace cache objects
6. Keep a local cache of file data and directory listings

Specification and system design

Figure 8 illustrates the proposed overall design of the SFTP driver system. Each module of the design will now be discussed in greater detail.

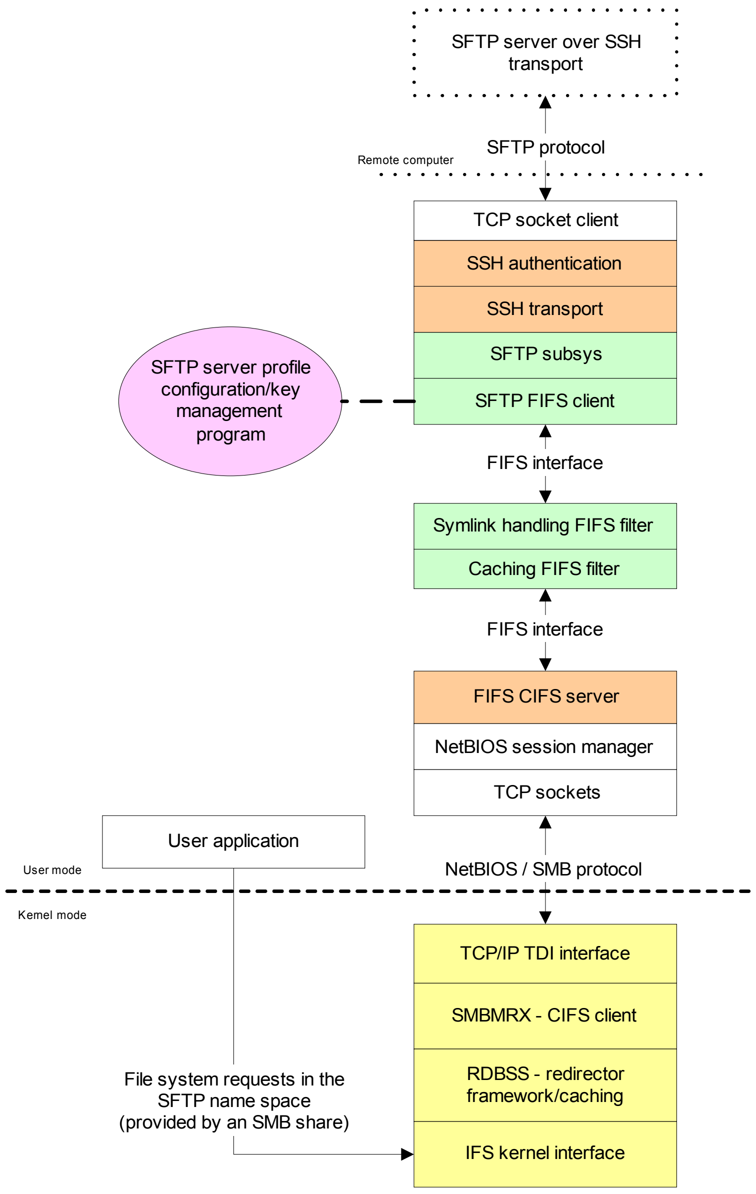


Figure 8.

LibSSH

In order to separate the SSH aspects of the SFTP protocol so that it was not tied closely to each other a SSH library module was designed. Having to handle SSH details in the SFTP module would have been unnecessary and not followed the the logical separation of the SFTP protocol from the secure transport layer. Indeed, theoretically it would be possible to run the SFTP protocol over any other layer, as such the flexibility in separating the SSH layer from the SFTP protocol was a necessity in the system design.

As the SSH protocol is a large and complex system that already was fully developed, it was decided that it would it would both be very time consuming and ultimately pointless to re-implement the SSH stack. Therefore, two libraries were looked at, the Unix OpenSSH library and the Windows PuTTY program. Although the OpenSSH library had a better program interface it was not targeted towards Windows and would have to have to have significant portions rewritten or necessitated the use of a Unix emulation layer such as Cygwin. PuTTY on the other hand was written from the ground up for Windows but does not actually have a library interface, it was closely tied to its GUI.

The SSH connection handler should represent one connection as a class object. The class will allow a connection to be initially configured by either a SSH private key filename being given or using the keyboard-interactive scheme with a stored password, which although is a security risk would be allowed for flexibility purposes. Once the server details are configured for the object the connection may be initialized by the underlying PuTTY backend using the *init()* PuTTY backend function. To handle socket initialization, server authentication and key-exchange the backend needs to be activated. This should be done synchronously in the LibSSH connection function.

Any errors during the connection phrase should be returned back and logged via an abstract logging interface given at object creation. The logging interface should allow log of differing levels of severity be output to any device implemented by a concrete log handler class.

The connection interface should transparently buffer data from the SSH connection. PuTTY passes data received over the transport layer to the global C function *from_backend(void * frontend, int is_stderr, char * data, int datalen)*. The interface should save the segment of data given for future access. A function should be provided that allows data of a certain length to be extracted from the buffer, or fail if there is not sufficient data is available, this function should return immediately. In order to handle the asynchronous nature of the SFTP protocol a function should be provided that waits for a given number of bytes to arrive from the server before returning, in a certain time period. This will allow the SFTP library to wait for the header of a packet before requested the remaining data of the packet by which time the size of such a packet will be known. Further details of the workings of this method will be discussed in the SFTP library section.

To send data a function shall be provided that given a data segment will pass it to the PuTTY backend using the *send()* function. The PuTTY *send()* function returns immediately and adds the data to the socket's outgoing buffer for future transmission. Guarantee of transmission or of any error is not necessary. Errors wills be determined by incorrect or non existent replies from the SFTP server. Additional utility functions for data management shall be provided, to flush the incoming buffer and to access a segment of the incoming buffer without deleting it from the buffer and checking the size of the incoming buffer.

An alternative approach to the data handling aspect of the data transport layer would have been to use the SFTP class as an observer of incoming data, by providing an interface for it to implement. This idea was discarded, however, as it would have added additional complexity to the SFTP module when the data handling part was clearly separate, the only benefit of doing so would have been to cut out a small layer between the SSH transport and SFTP layer.

SFTP library

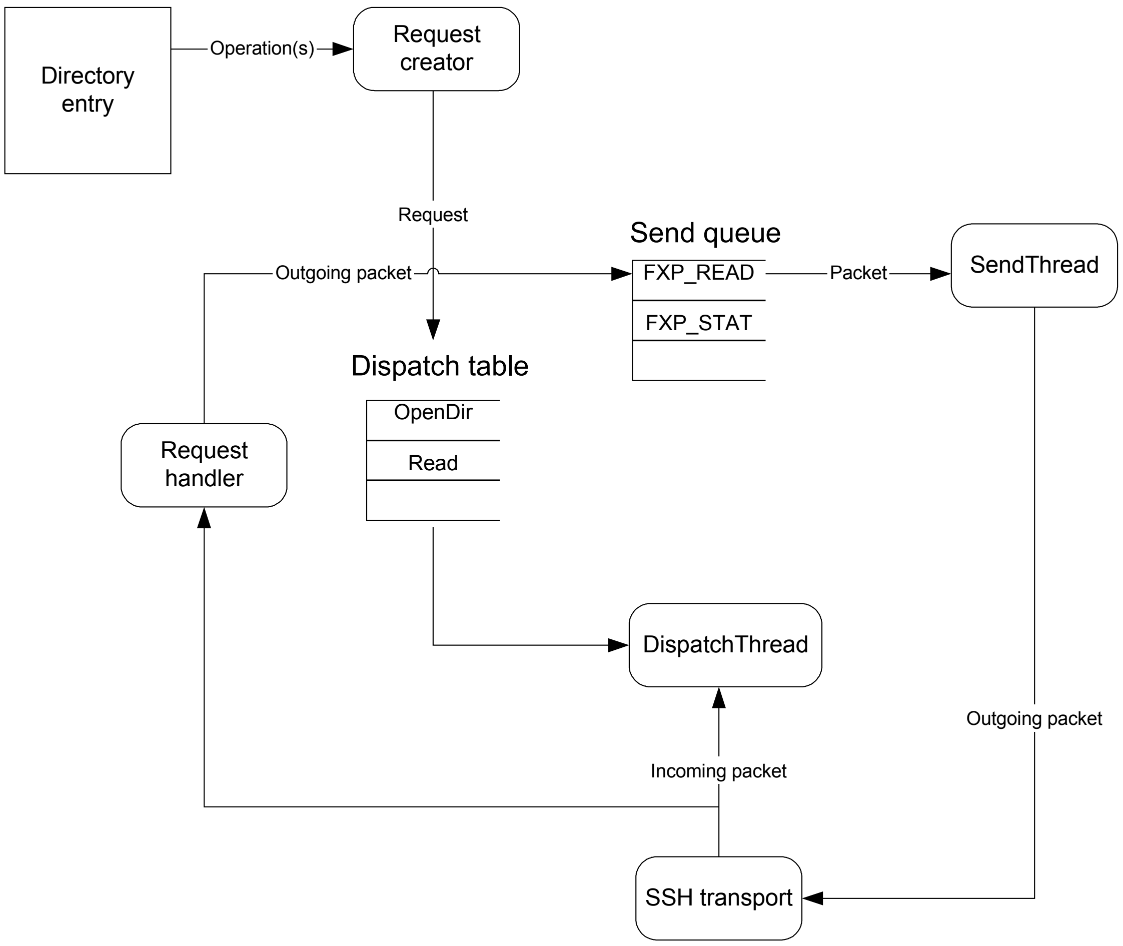
Crucial to SFTP is the concept of a filing system, as it is a file management protocol. It would be natural then to center the SFTP library around the concept of a treed file system rather than provide a myriad of top level functions to process that directly handled canonical file names. Therefore the SFTP library should be organized around the concept of a "directory entry" with the library providing only the root directory entry initially. The root entry should provided details of the file system capacity and free space. All entries should then provide a number of functions to perform various functions relating to the entry. Specifically:

Open	Create a handle to the entry on the SFTP server with a given permission and access mode set
Create	Create a new entry in a parent directory entry
Close	Close a previously opened handle to the entry
GetAttributes	Retrieve (or refresh) the stored attributes for the entry. I.e. size, flags, permissions, modification/access/creation times and ownership
SetAttributes	Set the stored attributes for an entry
Read	Read a segment of a previously opened entry
Write	Write to a segment of a previously opened entry
Reopen	For an entry that has been closed reopen a handle to it using the last given Open mode and permission set
GetListing	For an entry that is also a directory get a listing of its child entries
Delete	Delete the representation of an entry on an SFTP server, including all children
MakeDirectory	Create a new directory entry in a parent directory
Rename	Rename the representation of an entry on an SFTP server

Table 4.

All operations on directory entries should be handled asynchronously. All should return immediately with a request identifier object that will later be fulfilled. Calling functions may wait on a request to complete or request another operation on another entry (or the same entry) and await the completion of any request. The request response model shall provide for the possibility of errors during request processing in a request independent way. Requests shall be recorded by the SFTP manager object that will link each request to an SFTP request packet ID. Incoming packets will be dispatched to the correct request object for further processing (for example, the sending of more packets or to mark the completion of the request). This architecture is illustrated in figure 9.

Figure 9.



The dispatch packet handler should run as a concurrent thread and interface with the transport layer. It shall await for a minimum of 4 bytes (the packet length part of an SFTP packet header), once this is present it will examine its value and await the remaining length of the packet with a set timeout, if the packet's remaining data does not arrive in the given time the request is aborted and the request marked as erroneousness.

SFTP packets should be represented by a class hierarchy with an SFTPPacket base class at the top. The base class should provide generic methods of setting and getting SFTP packet fields in an easy to access and program manner. It should be possible to create a packet object using a raw data segment or create one from scratch.

Four types of data need to be supported. Unsigned 8 bit integers, unsigned 32 bit integers, unsigned 64 bit integers and UTF8 strings.

To add data to a packet two alternatives were considered. The first method was to have a set of functions that appended the different data formats to the packet in sequential order, the second method would use a intermediary data structure to create a list of data types and values which a function would then process when needed into the raw SFTP packet. The second method would make it easier to access data fields from the packet once they were added but add extra time for processing once the packet needed to be transmitted. Additionally, the second method would require extra processing to create the intermediary data structures to hold the fields. The first method would keep an area on the heap to contain the entire packet and extra fields when necessary from the area, as well as dynamically reallocate the area should an added field not fit.

A complication arises in efficiently accessing fields in the packet. Some packets like *SSH_FXP_NAME* have a complex nested structure that means it is not possible to index a field using an absolute offset, they require relative offsets to other fields, others have variable length fields like UTF8 strings. It would be possible to read through the entire packet data to get a particular field but this is inefficient. A lookup table created once linking field IDs to an absolute address in the packet data would therefore be a necessary design addition to the base class.

To implemented the directory entry operations listed in table x the following packet types will be implemented:

```
SFTPPacket

       SSH_FXP_INIT          create
       SSH_FXP_VERSION       access
       SSH_FXP_OPEN          create
       SSH_FXP_CLOSE         create
       SSH_FXP_READ          create
       SSH_FXP_WRITE         create
       SSH_FXP_FSTAT         create
       SSH_FXP_SETSTAT       create
       SSH_FXP_FSETSTAT      create
       SSH_FXP_OPENDIR       create
       SSH_FXP_READDIR       create
       SSH_FXP_REMOVE        create
       SSH_FXP_MKDIR         create
       SSH_FXP_RMDIR         create
       SSH_FXP_STAT          create
       SSH_FXP_RENAME        create

       SSH_FXP_STATUS        access
       SSH_FXP_HANDLE        access
       SSH_FXP_DATA          create/access

       AttribsBasePacket

              SSH_FXP_NAME          access
              SSH_FXP_ATTRS         create/access
```

Each would be a relatively light-weight wrapper around the packet data to either access packet data, create a packet with specific parameters, or both.

FIFS framework

The key component of this task is to achieve the fully working operation of the framework under current Windows operating system versions. Unfortunately, there is little room for design here and the problems and solutions must be determined by a trial and error process of analyzing the the running of FIFS and examining its interaction with the Windows SMB client. It may be possible to use existing working open source SMB servers like Samba to compare details of how operations are handled to ensure FIFS operates correctly.

FIFS SFTP module

The SFTP module will be the bridge between the FIFS server and the SFTP library, with a secondary link to the cache manager. Its design is limited by the need for it to implement the *FileSystemDispatch* interface. Files in FIFS are referenced by a 32 bit unsigned integer. This handle needs to be linked to a directory entry in the SFTP server. This would ideally be done using a hash map from handle to entry. The maximum number of handles stored must be bounded by a maximum handle count so that the Windows operating system will close files when it receives an 'no-free-handle' error to stop run-away file accesses eliminating available memory.

With regards to integration with the cache manager and prefetcher the SFTP module should allow caching parameters to configured via registry entry settings that will be set using an independent GUI program.

Predictive caching system

In [19] the 'Recent Popularity' predictor is described as a method of prediction that allows "great flexibility in balancing meta data requirements, specific accuracy and prediction coverage" by selecting the best K of M recorded successors. In table 5 [19] the operation of this predictor is illustrated.

	File Access Sequence	
S:	A B C D B C D B D B D B C B C B C A B A B A B A B	
	Per-File Successors	Successor Counts
A:	B,B,B,B,B	B(5)
B:	C,C,D,D,C,C,C,A,A,A	A(3),C(5),D(2)
C:	D,D,B,B,A	A(1),B(2),D(2)
D:	B,B,B,B	B(4)
	Best 3 *of* 6 *(B|S) = A*	
	Best 4 of 6 *(B|S) = NULL*	
	Best 4 of 9 (B|S) = C	

Table 5, adapted from [19]

A new predictor will be introduced that maintains the adjustable meta data requirements of Recent Popularity but modifies the prediction stability method away from simple popularity to a weighted popularity based on a number of factors.

Given *R* unique recorded successors of count *N* for a file X that can be referenced by r_{Xi}, the predicted M successor function, s(*X*), can be given by choosing the best *M* of *N* recorded successors. The definition of "best" will now be expanded on. Initially, the best can simply be thought of as

$$P(r_{Xi}|R)$$

Or the probability of a unique successor being the correct successor from the list of previous *O* recored open operations. The *M* successors will then be:

$$s(X) = \max_{i=1, i \notin V}^{N} (P(r_{Xi}|R))$$

However, this simple probability choice does not take into account factors of context and stability of the successor. First of all by weighting the probability by the mean inter-operation interval time between the recorded open operation and the the predecessor's opening long-term relevance can be applied. The longer the inter-operation interval the less likely the successor is actually connected to the predecessor. The inter-operation interval weight is thus defined as

$$\frac{1}{t_i - T}$$

where t_i *is the open time of the successor and T is the open time of the predecessor*

Additionally, greater weight should be given to recent recorded operations than those from the distance past

$$\frac{1}{t_i - D}$$

where *D* is the current time

To take into the global context of a successor the "best" factor needs to be adjusted by the prevalence of the successor

$$\frac{c_i}{C}$$

where c_i is the number of times the successor has ever been recorded and C is the total number of recorded operations

In summary, the complete prediction for an entry is thus:

$$\max_{i=1, i \notin V}^{N} \left(\frac{1}{O}\left(\sum_{j=1}^{O} \frac{c_i MP(r_{Xi}|R)}{C(t_j - T)(t_j - D)}\right)\right)$$

where M is a damping factor on the global context's weight

Name mangling layer

A problem identified in [1] manifests itself because of the case-insensitive nature of the SMB protocol and of current Windows file I/O implementations, although case in a filename is preserved it is not identifying. Many Unix implementations do use case as an identifying characteristic of file names. For example the file *"File"* is different to the file named *"file"*. This presents a difficulty for a file system driver accessing these similarly named files as it is impossible to determine which file the user intended to be accessed.

[1] proposes a solution whereby similar case folded names are mangled to make them identifying without their differing case. In the previous example "File" may become *"File~1"* and *"file"* may become *"file~2"*. This, however, is clearly not a perfect solution as it would for instance break automatic scripts that relied on referencing case identified names.

An alternative solution might be to force user interaction when an ambiguous file is referenced, requesting from the user which file they intended to access by presenting a list of alternatives to the user to select from. This laborious nature of this approach may be alleviated by recording the response of the user for future accesses. This, however, leads to the question of how a particular access is indexed. A suitable options here is to use the previous *N* file accesses as the reference point leading up to an ambiguous access, as this information is already being used for precaching. It does significantly provide no guarantee of uniquely referencing the file, a safety feature might be necessary to deny the ability to write to, or delete, ambiguous files.

Symlink handling

Symbolic links are a type of file that map one file or folder to another file or folder. Most often used in Unix operating systems they are handled transparently by the operating system as if they were the target file rather than merely a link to the file. SFTP contains special provisions for handling symbolic links or "symlinks", namely the *SSH_FXP_READLINK* packet . In combination with a directory listing marking a file as of the symlink type the target of that symlink can be determined. Additionally, the *SSH_FXP_SYMLINK* packet can be used to create a new symlink to an existing entry.

Support for symlinks on Windows is patchy and depends on the file system used. NTFS can create symlinks to folders (called Junction Points) and hard links for files (like symlinks but instead a direct link for files belonging to the same volume with no intermediary hop entry.) Current versions of the operating system do not allow the creation or modification of links through the standard file system management utilities but console utilities and 3rd party solutions exist that allow this. These programs only however work directly on the NTFS system, there is no equivalent standard system for managing symlinks over the SMB protocol.

The issues of symlink handling as it pertains to an SFTP file system driver are thus:

Following symlinks. When a symlink is accessed it should be transparently mapped to the destination. This is trivial to implement and merely requires that the FIFS create function check whether the access is to a symlink and if it is get its final destination and link the SFTP handle to that request. An issue with this design is that there is no way for the user to determine if the file they are accessing is a symlinked file, unlike with most Unix systems. Like case sensitivity this may be solved by mangling symlinked files with a new named, for example "file-link", however, again this presents additional problems. As current versions of Windows do not attempt to signify the presence of symlinks to the user using normal utilities this issue may be considered irrelevant, it is only necessary to show symlinks using special programs. The NTFS Junction Point system requires direct access to the NTFS volume and therefore existing utilities are not suitable, a custom management program is necessary. The issue is how to communicate symlink information to the custom program. One solution is to override the FIFS system and use a program that directly interfaces with the SFTP server, alternatively SMB provides an IPC system whereby the server may implement an arbitrary interface to be used by a 3rd party program. It would be possible to design a protocol of getting symlink information. Additionally, it could be used to create and modify symlinks.

Deleting symlink folders. When a user deletes a symlinked file or folder it is only required that the link is deleted, not the destination file or folder. This must be handled in the symlink system by requiring any calls to the FIFS delete interface to check the entry type.

Creating/modifying symlinks. As discussed for following symlinks there is no intrinsic system in Windows 2000 or XP of doing this, special utilities must be implemented.

Cache coherency

In ensuring a cache system is guaranteed to be accurate when retrieving cached objects it must fulfill a number of requirements:

1. All cached directory listings must be up to date.
2. Cached file pages are coherent with the actual data on the file volume.
3. Written data is sent to the file system without conflicting writes from other processes.

Maintaining directory listing correctness is something that only can be guaranteed with the file system informing any listening clients of a change to that listing. The client may then update its cache of the directory. It is, however, not necessarily a requirement that a directory listing is guaranteed to be up to date, it may be sufficient for it to be up to 30 seconds out of date, in which case it would be acceptable for a client to maintain the time a directory listing cache was last obtained, and then check every 30 seconds whether the modification date for the directory entry was never than the stored entry, which if it was true then a new listing should be retrieved.

To maintain cache page coherency the situation is a little more complicated. To guarantee the cached page will not be changed it must be write locked so that no other process may change it. This is a significant requirement that means only one process may access a section of a file at a time, which depending on the time that the cache page "lives" for (or is kept for) might be many minutes. An alternative to block level write locks is an "opportunistic lock". This is a type of lock that whereby a client requests a lock for a file or section of a file which should no other process have such a lock, is granted. However, should another process already have a lock then that lock is revoked and the original client that created the lock is informed, the new client is also denied the request. In this way the client may enable caching for that file if it gains a lock, or otherwise it will resort to direct read requests. Similarly, in the event of that a message informing it of a lost lock it must abandon the cache for the file and resort to direct reads.

In the case of opportunistic locks the cache manage may when receiving a write request just update the cache page. When the lock is lost it must write any dirty (overwritten pages) back to the server.

The use of opportunistic locks and directory change notifications would be the ideal situation for use with the SFTP driver. However, there are a number of obstacles to this:

1. The SFTP protocol does not support the concept of directory change callbacks, something which until recently the commonly used Linux kernel did not even support natively

2. Later versions of the SFTP protocol introduce block level write, read, and read/write locks.

This makes cache control approachable, albeit with some highly undesirable characteristics. Additionally, although this is part of the SFTP version 6 onwards, the SFTP server used by the majority of servers (OpenSSH) only supports SFTP version 3.

These problems mean that without changes to existing SFTP servers there is no good alternative to ensuring cache coherency. For the purposes of the design this crucial aspect of the caching system is ignored. In the hope of encouraging changes to the SFTP protocol to support these features that are important to the development of high performance file systems that use SFTP two (optional) amendments to the latest released draft standard are proposed:

- Directory change notification callbacks (*appendix 1.1*)
- Opportunistic block level locks (*appendix 1.2*)

In the absence of any servers supporting these proposed amendments, cache control in this design will be limited to checking the modification dates of directory listings so they are valid within the last x amount of seconds and that file cache pages are valid by checking the file level modification date every x amount of seconds (discarding the entire set of cache pages for an object should it have changed since the last check). All writes should be write-through in an effort to maintain cache coherency.

Implementation discussion

The implementation of this project was completed in C++ using Microsoft Visual Studio .net 2003. This choice of development environment was a natural decision to make as the FIFS framework was designed in C++ using Microsoft Visual Studio. A number of projects were created which were as follows:

- *'server'* – the FIFS system based on the original FIFS framework
- *'fssftp'* – the FIFS SFTP driver, SFTP library, and caching system
- *'netapi32'* – the packet tracing layer based on an original FIFS project
- *'LibSSH'* – the SSH transport. Incorporating the PuTTY backend.

- *'prefetchDBView'* – utility program to view the prefetch database
- *'cacheTest'* – test program for the cache manager
- *'testSFTP'* – test program for the SFTP library
- *'benchmark'* – console program to simulate file access scenarios for benchmarking purposes

Wherever possible the Standard C++ Template Library (STL) was used to reduce implementation time. Additionally, the 'Strings' package authored by Cail Lomecb was used for standardized string handling of multiple character sets and dynamically sized string buffers. Finally, the OU Thread wrapper class written by Vijay Mathew Pandyalakal was utilized in a modified form for thread control.

System architecture

In order to illustrate the overall system architecture an example trace of a call to open the root directory (\) of a mounted volume will now be described:

The *create(parentName, name, flags, attr, phandle)* FIFS dispatch handler is called. This locks the function to prevent concurrent requests arriving and resulting in object inconsistencies.

create() then calls *findEntry* after getting the parent handle using the *phandle* parameter. *FindEntry* checks the parent request has completed and then compares each name in the directory listing to one being searched for before returning the correct entry if one is found.

Create() now adds this entry as the successor to previous call to *create()* by calling *AddSuccessor(lastName, fullName, time, interval)* on the prefetcher database manager.

Next *CreateHandle* is called which allocates a handle for the request before calling *getFile()* which attempts to either get a cached copy of a file or requests a new handle from the SFTP library. It uses *getCacheObject()* to find the entry if available or otherwise in this case opens a directory handle by calling *ReqGetDirectory* for the given directory entry object. A cache object is then created using the *createObject* function of the cache manager. The request is not waiting for competition, this is checked when the directory listing is actually read using the *readdir()* FIFS framework dispatch function.

After *getFile()* has returned the prefetcher is called to request successor items. Here *getEntry* is used to map the string file name into a *DirectoryEntry* object, this object is then passed to *getFile()* and the request is added to cache open list.

Finally, *handlePrecacheQueue* is called to handle queued prefetch reads and opens. This uses two loops. The first iterates through the open queue to call check *isComplete*() on each item, which if successful it will queue a read for the first x pages of the file if it's not already in cache by creating an *AsyncReadReq* object and adding it to the end of the read queue. The second loop performs a similar task with the read queue except for each completed item it puts the read data segment in the file's cache object using *PutPage()*

LibSSH

The Secure SHell module presented no significant problems. A clean C++ interface to the PuTTY backend was aimed for in *SecureShellConnection*, however, due to the design of PuTTY some compromises had to be made. For example, PuTTY calls a global C *SSHFatalError* function on connection problems. The LibSSH module passes this message onto to an observer class in the SFTP library via the use of the *SecureShellInterface*

interface class. This class is given before any SecureShellConnection may be created using a global initialization function. Ideally, the LibSSH module would capture all errors and determine which type they were (read/send/control) and buffer these errors for when a call to the respective member function was called in a *SecureShellConnection* object. However, this technique presents difficulties for when there are multiple *SecureShellConnection* objects as it is not possible to determine the source in the *SSHFatalError* function. The current implementation of the LibSSH module currently only supports authentication using keyboard-interactive, public key encryption would be simple to add with some changes to the *Config* configuration class in *OpenConnection()*.

To handle incoming data a segment is passed to *FromBackend()* from the PuTTY backend, a *DataChunk* object is created for each class which is added to a queue. Data can now be extracted by a user class using the *GetData()* function which will attempt to extract up to a certain number of bytes of data from the queue. This function supports extraction from overlapping and subsets of chunks.

To handle logging the *LoggingInterface* class is implemented which defines a virtual function to output messages of predefined types along with an optional descriptive string. The SFTP library in its current implementation merely prints these messages onto standard out however it would be simple to modify the logging object to write to a file.

SFTP library

One area not considered in the design was the concurrency issues of the dispatch request system and the integration with the synchronous SSH transport.

The SFTP class makes use of four mutexes and two semaphores to control access to data structures and organize access to the SSH transport. The two semaphores are used to signal when there are entries in the dispatch table and the sender queue respectively and are waited on in the dispatch handler thread. Initially, the implementation used two separate threads, one for handling the dispatch table and another for the send queue, however, this resulted in problems whereby both competed to gain access to the SSH transport (via one of the the mutexes) thereby providing poor dispatch and sending performance.

The implementation of the *SFTPPacket* base class for packet handling required a decision to be made on how to access fields of a packet in a generic and extensible manner. To support this a framework whereby the specific packet class could define a table of fields it expected to find in a packet, each row containing a unique identifier for the field, and information on the type of the field. This table was then given the member function of *SFTPPacket* for it to create a *hash_map* based lookup table from the unique identifier to an offset in the packet data. Further functions were then introduced to get the value of particular fields via their unique ID field.

Listing 1 shows the implementation of the dispatch controller thread system:

```
void DispatchThread::run() {
        HANDLE objects[2];
        objects[0] = sftp->senderSem->getHandle();
        objects[1] = sftp->dispatchSem->getHandle();

        while (sftp->GetActive() != SFTP_SHUTDOWN_DISPATCH) {
                DWORD result = WaitForMultipleObjects(2, objects, FALSE, 500);

                switch (result) {
                        case WAIT_OBJECT_0: { sendPacket(); break;}
                        case WAIT_OBJECT_0 + 1: { handleRequests(); break; }
                }
        }

        sftp->SetActive(SFTP_INACTIVE);
}
```

```
void DispatchThread::handleRequests() {
        sftp->waitForAnyPacket(SFTP::DEFAULT_TIMEOUT);

        // Check packet list for request IDs that match ones that are pending in the dispatch table
        Thread::wait("DispatchTableMutex");
        vector<DispatchInterface *>::iterator entry;
        bool found = false;

        for(entry = sftp->dispatchTable.begin(); entry != sftp->dispatchTable.end(); entry++) {
                DispatchInterface *req = *entry;
                SFTPPacket *packet = sftp->getPacketReply(req->getRequestIDWaiting());

                if (!req->isComplete() && packet != NULL) {
                        // Got reply for a pending dispatch entry
                        if (req->start(packet)) {
                                // Delete from table and mark as done
                                entry = sftp->dispatchTable.erase(entry) - 1;
                                req->setComplete();
                                found = true;
                                break;
                        }
                }
        }


        // Try again?
        if (!found) sftp->dispatchSem->release();

        Thread::release("DispatchTableMutex");
}

void DispatchThread::sendPacket() {
        Thread::wait("QueueMutex");
        SFTPPacket *packet = sftp->outgoingQueue.front();
        sftp->outgoingQueue.pop();
        unsigned int count = sftp->outgoingQueue.size();
        Thread::release("QueueMutex");

        // Actually send the packet
        unsigned char *data = packet->GetPacketData();
        unsigned int length = packet->GetPacketLength();

        char* debugPacket = new char[length * 3 + 200];
        packet->GetDebugString(debugPacket);
        sftp->logger->Log(LoggingInterface::Information, "Sending packet %d (%s)", packet,
debugPacket);
        delete debugPacket;

        Thread::wait("SSH", 500000);
        sftp->channel->SendData(data, length + 4); // 4 extra for length word not counted in packet
length
        Thread::release("SSH");

        // More packets to send
        if (count) sftp->senderSem->release();

        delete packet;}
```

Listing 1.

The remaining parts of the SFTP module were implemented as designed, with a *DirectoryEntry* class created that supported a number of operations, including the capability to get a listing of child *DirectoryEntry* objects.

FIFS system

As previously discussed, originally the system was designed for Windows NT. Since then Microsoft has made a number of changes to the way Windows operates, if not for the SMB protocol version used by FIFS, that meant it was not in a working state. It was a time consuming process to analyze the problems, and included a need to study the SMB protocol and existing SMB servers like Samba. This was a highly time consuming process, requiring in-depth investigation of the complete SMB protocol, which was not always authoritative in the face of various quirks in the Microsoft implementation.

The following obstacles were found to letting the FIFS framework operate under Microsoft Windows XP or under some conditions:

- Multiple adapters – the FIFS system did not correctly bind to the correct network adapter when setting up the NetBIOS link when there were multiple adapters present on a machine.

- The *TreeConnect* SMB function accepted connections for all service requests. Newer versions of Windows introduced different types of IPC services. It was fixed so only accept file system ("A:") type connect requests and denied all other types.
- The Trans2 SMB packet handler function implemented the offsets and parameter specification incorrectly by copying the Setup block from request packet to response packet. This extraneous data confused the SMB client.
- Added support for *Echo* command.
- The *OpenAndX* SMB function needed an Action command. Set to 0x8000 (File opened by another user, or mode not supported by server) when server requests exclusive access (which cannot be guaranteed.)
- Errors were not being correctly processed. The byte count, and word count parameters were invalid. This caused Windows to stall. All erroneous SMB packets were also required by the SMB client to have to include empty response section.

netapi32

To help with debugging a clone version of the Microsoft library *netapi32.dll* was made that expanded on the layer provided by FIFS that implemented a custom, and mostly incomplete packet tracer. The new version intercepted a function call and outputted the packet to a packet capture file with the use of the *WinPcap* library before forwarding the call on to the “real” Microsoft function. This packet capture file was than analyzed with the Ethereal program and its powerful SMB protocol decoder.

A base packet (as shown in listing 2) was created by extracting the TCP and NetBIOS headers from a trace of the Samba SMB server using Ethereal. This base packet is then modified at run time to add the SMB packet.

```
static unsigned char basePacket[] = {0x00, 0x40, 0x63, 0xdb, 0x15, 0x34, 0x00, 0x0f,
       0xea, 0x73, 0x60, 0xb1, 0x08, 0x00, 0x45, 0x00, 0x00, 0x90, 0x36, 0x3c, 0x40,
       0x00, 0x80, 0x06, 0x38, 0x63, 0xc0, 0xa8, 0x05, 0x46, 0xc0, 0xa8, 0x05, 0x32,
       0x05, 0xfc, 0x01, 0xbd, 0x63, 0xaa, 0x4e, 0xa5, 0x80, 0xfb, 0x19, 0x9d, 0x50,
       0x18, 0xfd, 0xb3, 0x8c, 0x4b, 0x00, 0x00, 0x00, 0x00, 0x00, 0x64};

// Dump packet
header.len = dwSize + sizeof(basePacket);
packet = calloc(1, header.len);
if (packet) {
       memcpy(packet, basePacket, sizeof(basePacket));
       memcpy(packet + sizeof(basePacket), pSmb, dwSize);

       *((unsigned short*)(packet + 0x10)) = htons(44 + dwSize); // IP length
       *((unsigned short*)(packet + 0x37)) = htons(8 + dwSize); // NetBIOS length

       header.caplen = header.len;
       gettimeofday(&header.ts);
       free(packet);
       pcap_dump(dumpfile, &header, packet);
}
```

Listing 2.

FIFS SFTP module

The *Fs_SFTP* class implements the FIFS *FileSystem* class to provide an interface to any number of *FsDT_SFTP* objects which are created dynamically based on the user configuration of the FIFS framework and what server configurations are required. The driver looks up server parameters from the Windows registry entry for the mount. The *FS_SFTP* is also concerned with maintaining a pool of SFTP connections indexed in a hash table by the server name, port, and connected user name. This way there is only ever one uniquely defined connection to a server which may be used by any number of similar mount points. It also brings to the user the advantage of faster connect times as after the first mount, the connection phase, which with SSH and SFTP can be a number of seconds, is eliminated.

The *FSDT_SFTP* class implements the FIFS file system dispatch interface to interact with the SFTP library. It maintains three main data structures. Most crucially is the file handle lookup table, each entry in the lookup table consists of a *DirectoryEntry* object, a *DispatchInterface* object and a *CacheObject* object. The *DirectoryEntry* object is used primarily for looking up entries in the create function from the parent folder (a handle).

The *DispatchRequest* object is linked to either a *OpenDirRequest* or a *ReadFileRquest* in this implementation but may be linked to any other type of request in the future. Only one request may be linked to a handle at a time as only one operation may be performed on a handle at a time as per the FIFS specification. When a request is fulfilled by the *FsDT_SFTP* interface handlers check that the request has been completed by the dispatch handler thread before accessing the request object's data. In some cases there will be no dispatch request linked to a handle, in this case the cache object contains all required data for the handle.

The cache object is either a *CacheFileObject* or a *CacheDirectoryObject* depending on what type of handle it is linked to. Both object types will be discussed later in the implementation section.

The other key data structures are the pending asynchronous read request list and the precache asynchronous read request list. Both are implemented as linked lists and are used by the prefetcher to queue successor open requests. The open list is checked whenever the FIFS framework opens or creates a file and checks whether any requests have completed, any completed requests are added to the cache system for future usage. Additionally, when an asynchronous open request has successfully completed a *ReadAsyncReq* is created, which specifies details of the read, such as length, offset, and linked data segment to read data in to. In the FIFS read handler this read request list is checked in a similar manner to the open list for completed operations. Any completed operations are added to the linked cache object's page. The number of bytes to prefetch is a configurable parameter that can range from the first few kilobytes to the complete file depending on the user's requirements of how to balance speed/bandwidth usage.

The read function of the FIFS dispatch handler is of particular interest. It requires a complex interaction between the FIFS framework, the caching system, and the SFTP library.

```
DWORD FsDT_SFTP::read(fhandle_t handle, UINT64 offset, UINT64* pcount, void* buffer) {
        [prolog - concurrency. State verification]

        DWORD error = ERROR_SUCCESS;
        INT64 left = *pcount;
        char* data = (char*)buffer;
        UINT64 total = 0;

        CacheFileObject *object = (CacheFileObject*)handles[handle].object;

        [make sure open request is complete]

        unsigned int count;

        unsigned long long fileSize = handles[handle].entry->GetSize();
        while (!error && left > 0 && offset < fileSize) {
                if (object) {
                        // Try and get this page from the file's cache
                        count = min(left, CACHE_PAGE_SIZE);
                        count = object->GetPage(offset, count, (unsigned int*)data);
                }
                if (count == 0) {
                        unsigned int pageOffset = floor((double)offset / CACHE_PAGE_SIZE) *
CACHE_PAGE_SIZE;
                        unsigned int dataOffset = offset - pageOffset;

                        // Read either what is left or up to the maximum SFTP packet size
                        count = min(left + dataOffset, floor((double)MAX_READ_SIZE / CACHE_PAGE_SIZE)
*

                                CACHE_PAGE_SIZE);

                        if (object && !object->GetRequest()) {
                                [file was closed. reopen file to meet request]
                        }

                        ReadFileReq* readReq = handles[handle].req->GetEntry().ReadFile(pageOffset,
count);

                        readReq->waitUntilComplete(DEFAULT_WAIT);

                        [check read errors]
                        else {
                                count = readReq->SegSize() - dataOffset;
                                memcpy(data, readReq->GetFileDataPointer() + dataOffset, count);

                                for (int pageIndex = pageOffset ; pageIndex < pageOffset + readReq-
>SegSize() ;                                    pageIndex += CACHE_PAGE_SIZE) {
                                        unsigned int pageLength = min(CACHE_PAGE_SIZE, (pageOffset +
                                                        readReq->SegSize()) - pageIndex);
                                        object->PutPage(pageIndex, pageLength, data + (pageIndex -
                                                        pageOffset));
                                }
                        }
                }

                total += count;
                offset += count;
                data += count;
                left -= count;
        }

        *pcount = total;
        return error;

}
```

Listing 3.

Predictive caching module

The *CacheManager* is a relatively straightforward LRU paging caching manager that contains no novel techniques yet still presents some interesting software engineering design choices.

Cache objects are derived from the *CacheObject* base class. The *CacheDirectoryObject* contains a list of *DirectoryEntry* objects for a parent directory while the *CacheFileObject* class contains information on the directory entry it is caching along with an array of links created based on the cached object's file size. Each link points to *CachePage* object, or null if the page is not in cache. The *CachePage* object is an object containing a fixed size array of bytes to hold cache data and a number of associated parameters. One of which is a pointer back to the *CacheFileObject* and the offset into its page index array for where the *CachePage* is being used. This way when the *CacheManager* allocates this page to a new object the previous owner of the page will have the page index at the offset set to null to signify the fact that is no longer cached.

A complete set of all *CachePage* objects is held by the *CacheManager* as a fixed size array. The *CacheManager* also has a list object of *CachePages* sorted by the order in which they were last accessed so as to be able to relatively efficiently decide which page to use when one is requested by a *CacheFileObject*. In addition the *CacheManager* uses a hash map to map object names to their *CacheObject* objects. A complication here was that for a single object name there could be in some cases be two associated objects, a directory may have a directory list cached as well as an cached handle to the directory “file” to get modification/size information. It was considered that it might be necessary to have two hash maps, one for *CacheFileObjects* and one for *CacheDirectoryObjects*, however this was discounted as it required additional memory and complicated the design. The chosen solution was to to require callers of the *CacheManager* to notify it of what type of object it wanted to find. The identifying mark for objects was modified to include an indication of what type of object it was and this mark was properly appended when an object was inserted into the hash map.

In implementing the previously discussed prediction system a “prefetch database” design was created to allow optimized successor record keeping as well as fast successor prediction. Listing 4 shows the implementation of the predictor algorithm:

```
void PrefetchFile::PredictSuccessor(const String &name, StringBuffer &result) {
	jumpToName(name);

	// Read in successors block
	SuccessorBlock blocks[MAX_SUCC_COUNT];
	FilePrefetchInfo successors[MAX_SUCC_COUNT];

	fseek(file, PREFETCH_BLOCK_OFFSET(name), SEEK_CUR);
	fread(blocks, sizeof(SuccessorBlock), succCount, file);

	readSuccessors(blocks, successors);

	FilePrefetchInfo *uniqueSuccessors[MAX_SUCC_COUNT];
	unsigned int uniqueSuccessorsCount = 0;
	double successorProb[MAX_SUCC_COUNT];

	unsigned long long currentDate = getCurrentTime();

	for (unsigned int i = 0 ; i < GetSuccessorCount() ; i++) {
		// Skip null successor records
		if (successors[i].name.length() == 0) continue;

		// Look for successor in recorded successors
		unsigned int j;
		for (j = 0 ; j < uniqueSuccessorsCount ; j++)
			if (uniqueSuccessors[j]->name == successors[i].name) break;

		// Not seen yet
		if (j == uniqueSuccessorsCount) {
			j = uniqueSuccessorsCount;
			uniqueSuccessors[j] = &successors[i];
			uniqueSuccessorsCount++;
			successorProb[j] = 0;
		}

		// Calculate weight
		successorProb[j] += (successors[i].accessCount * PREFETCH_ACCCESS_COUNT_M) /
			(successors[i].interval * (currentDate - successors[i].date) *
(TotalAccesses() + 1));
	}

	// Get max
	unsigned int max = 0;
	for (unsigned int i = 0 ; i < uniqueSuccessorsCount ; i++)
		if (successorProb[i] > successorProb[max]) max = i;

	if (uniqueSuccessorsCount) result.append(uniqueSuccessors[max]->name);
}
```

Listing 4.

Prefetch database

The prefetch file was designed to minimize the number of reads when accessing successor information and allow new successor data to be added with minimal writes. To simplify the design of the file it was decided that it would not be necessary to ever delete successor data or file entries from the file as no situation could be conceived where this would be necessary as the prefetch algorithm takes into account the time of recorded successors when deciding what to prefetch. Finally, space requirements were given a lower a priority than the first two but consideration was still given to making the structure as space efficient as possible.

None of the implementation requires any of the file to be read into memory in the long term, all requests are handled by directly referring to the file. This is not normally a performance penalty as the operating system will have cached the commonly accessed parts of the file and the total number of file operations is very low (as analyzed later). Future implementations might be wise in explicitly caching some parts of the file in memory to reduce file system access overhead.

The prefetch file structure consists of a header block that stores the total number of file accesses recorded by the file, as well as data for the parameter of a stored hash table. The hash table is used to map file names to a file entry block in the file. Each hash bucket links to a linked list of pointers to entry blocks, an entry block consists of the full file name of the entry, the access count for the entry and a list of successor blocks. Each successor block points to the entry block of the successor as well as the time the successor was accessed and the interval between the successor being opened and the predecessor's open time.

Offset	**Type**	**Description**
	Header block	
0	*uint*	Hash table size
4	*uint*	Total number of successors stored
8	*ulint*	Total number of file/folder accesses recorded
	Hash block	
16	*uint*	Offset of first entry of hash bucket 0
16 + n	*uint*	Offset of first entry of hash bucket n
	Entry block (offset o)	
o + 0	*cstring*	Object name (0 terminated, len = n)
o + n	*uint*	Access count
o + n + 4	*uint*	Next object for hash bucket (or 0 = none)
o + n + 8	*NA*	Successor block 0 of c
o + n + 8 + b	*NA*	Successor block b of c
	Successor block (offset o)	
o + 0	*uint*	Offset to entry block of successor
o + 4	*ulint*	Date successor was accessed
o + 12	*uint*	Interval between originator and successor open

Table 6.

In order to assess the performance of this file format, the average access time to read an entry and its successors will now be determined:

*Access time = (header read) + (hash bucket reads) * n + (hash bucket jumps) * n + (entry block reads) * (1 + c) + (successor jumps) * c + (successor block reads) * c*

When:

n = average number of entries in hash bucket + 1
c = number of successors

Access time = reads(2 + n + 2c) + jumps(n + c)

The access is time is constant across all entries.

Size complexity of the database structure is as follows:

*File size = 16 + 4 * (hash size) + ((name length) + 9 + 16 * (record count)) * (file count)*

For example when hash size = 7919, record count = 6 and average file name length of 200 the constant storage size is 31,692 bytes plus O(305C). For example for a 10,000 file file system the prefetch size could reach up to 30MB.

Results and achievements

Benchmarks

To ascertain the performance of the implementation as well as judge the effectiveness of the prefetching system it was necessary to perform a range of benchmarks. Three metrics were looked at, the average number of milliseconds to perform an "operation", the cache object hit rate, and the cache page hit rate.

Test methodology

All tests were repeated a number of times to verify results. After each test round the server was restarted to clear the cache, and the prefetch database was cleared so as not to interfere with the next test round. Each test round sometimes consisted of a number of repeated sub tests where the server was not restarted and the caching system would be utilized, for these types of tests the caching hit rate figures are for the first run of the tests, not for the entire run of repeated sub tests.

Six types of tests were performed:

1. Deterministic directory reading. This test read in a arbitrary directory (in the test case the folder /usr) and then chose an a directory from that root directory another directory to read which was then added to a list of directories. The directory chosen was determined deterministically based on the iteration count. This was to show the effectiveness of the predictor in the "best-possible" scenario.
2. Deterministic + random directory reading. Similar to the first test however this test mixed in random directory reads with the deterministically chosen directory reads in an effort to show how the prefetcher would work in real-life scenarios where there is not always a definite prediction.
3. This test combined test 1 with reading file content. Files chosen for reading were deterministically chosen based on iteration count. The number of bytes read was set at a value less than the maximum prefetch size.
4. As test 3 but with random file reads.
5. As test 3 but with a 1 second delay between each file open.
6. As test 4 but with a 1 second delay between each file open.

Some or all of the tests were performed on three types of SFTP servers

- LAN server. A SFTP server directly connected to the client system. Average latency of less than 1ms.
- WAN server. A geographically close SFTP server connected to the client via the Internet. Average latency of 25ms.
- High latency WAN server. A geographically remote SFTP server connected to the client via the Internet. Average latency of 180ms.

Tests were done not only using the project's SFTP driver but also two other systems when available. The SftpDrive program (v1.5) and the Microsoft SMB client connecting to a

Samba server mapped to the same file space as the SFTP server.

All tests using the project's SFTP driver make reference to a number of parameters for caching control, prefetcher control. These are as follows:

Variable	*Meaning*
I	Iteration count of the prefetch database. Equivalent to the number of times a test set was run and prediction data allowed to be recorded.
P	Number of 1kB pages in the cache manager.
O	Maximum number of objects allowed by the cache manager.
L	Prefetch lookahead value. A value of 1 means the successor of an access is predicted. A value of 2 means that the successor of the predicted successor is also predicted.
R	Prefetch operation record type. Either “FD” when both file open operations and directory list operations are stored as successor events or “F” when just file open events are stored. In both cases the record operation history window was 6 events.

Table 7.

Test scenario limitations

Although the tests were formulated to show how the driver will operate in the best case as well as in less desirable circumstances they still do not provide an effective analysis of performance of the prefetcher in a real world situation. Amer et al. and others commonly use a long term trace file built up from adding a file operation tracer on a live working machine and then using this trace file to analyze the performance of caching algorithm:

> “ Simulations were run on file system traces gathered using Carnegie Mellon University’s DFSTrace system. The tests covered five systems for durations ranging from a single day to over a year. The traces represent varied workloads, particularly mozart a personal workstation, ives, a system with the largest number of users, dvorak a system with the largest proportion of write activity, and barber a server with the highest number of system calls per second.”

This type of testing, unfortunately, was not possible due to the time constraints of the project however it would certainly be useful to compare the model to existing prefetcher designs. Although, the artificial scenario cache hit rates would seem to compare favorably with existing solutions.

Low latency results

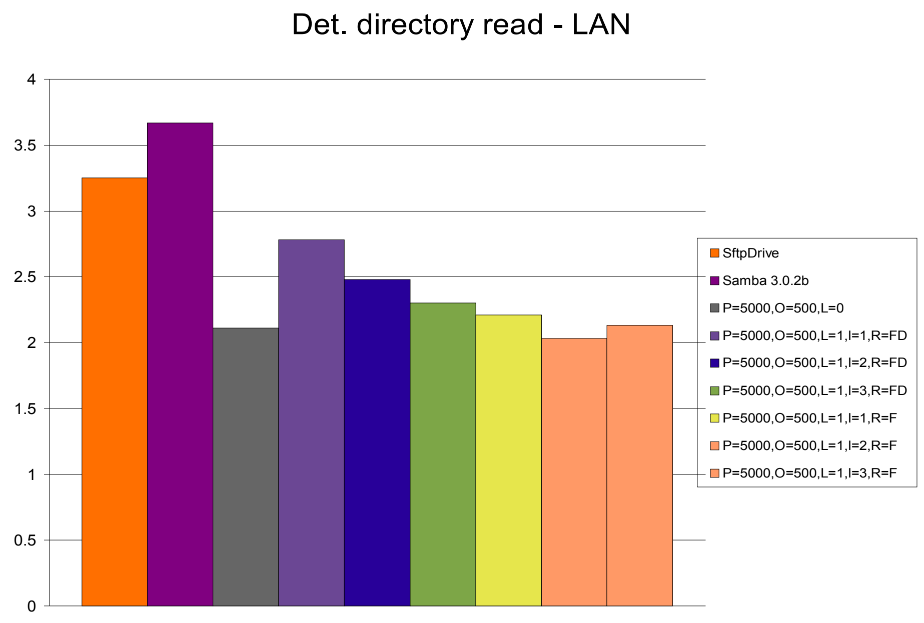


Figure 10.

Figure 10 illustrates the efficiency of the project's directory read system regardless of whether prefetching is actually enabled. Indeed, prefetching enabled actually results in a slight performance decrease.

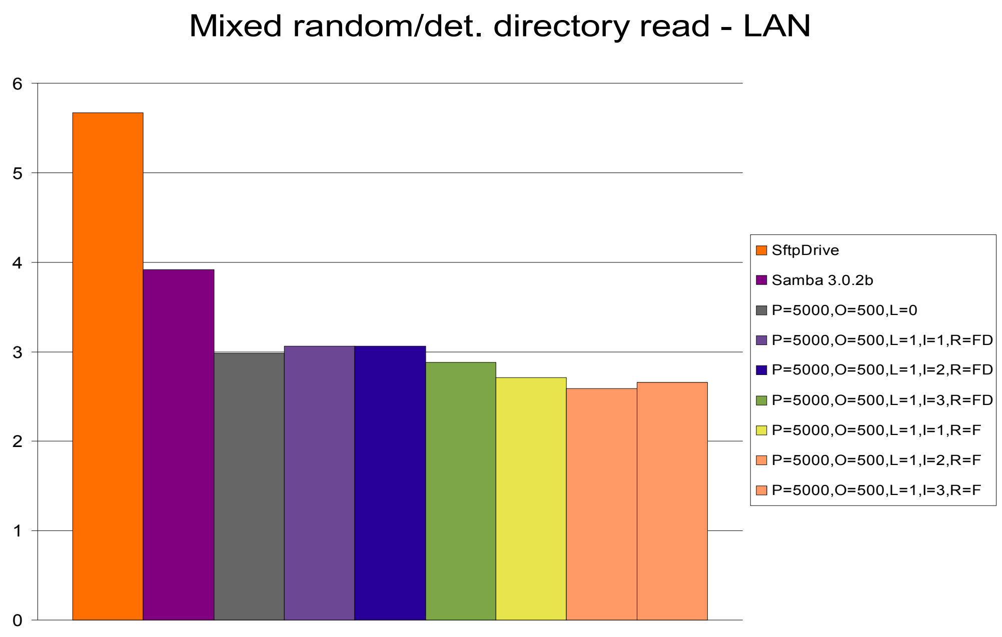


Figure 11.

Again, in figure 11 the project's directory read performance is very high. Even outperforming both natively implemented kernel-space file system drivers. This performance advantage was theorized in [10] that because FIFS drivers have full access to user-space resources, which are more often used in networked based file system drivers, especially ones like SFTP which need large security layers.

Medium latency results

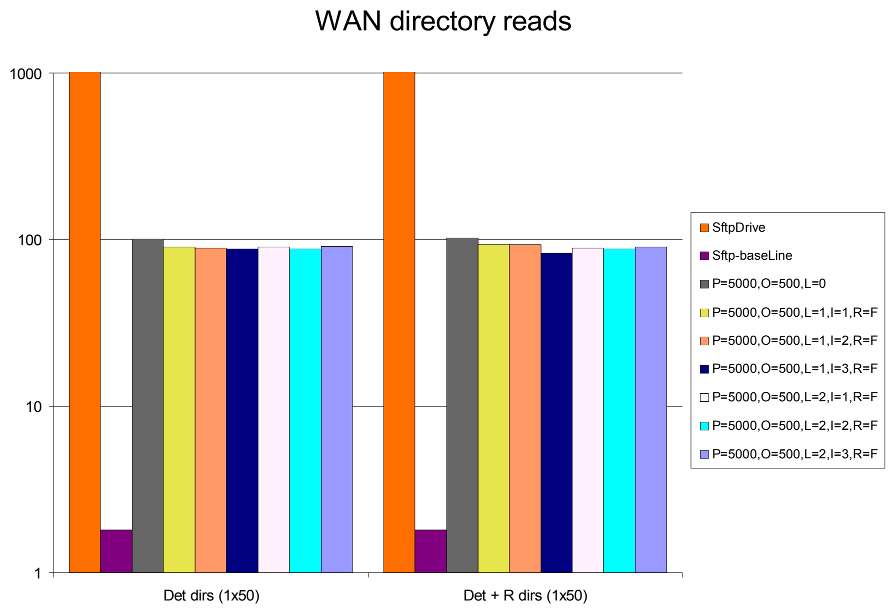


Figure 12.

Figure 12 shows how when server latency increases the performance advantage of this project is considerable compared to existing solutions. The reason for this is unknown however it might be due to a very minimal level of caching by the SftpDrive product

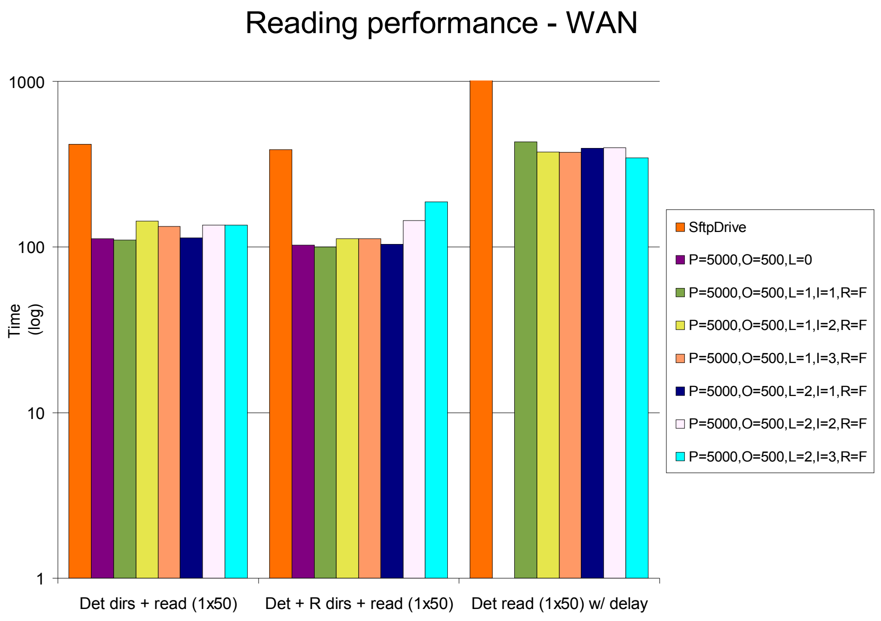


Figure 13.

In figure 13 the project continues to have very good performance in a WAN scenario. In was interesting to see here that with prefetching enabled there was no performance increase, rather the opposite, performance went down by a small amount. It was theorized that the reason for this was that the fast link to the server meant that a file read operation was started, at which point a prefetch is started, however, the first read operation completes very quickly and the test framework continues with the next read before the prefetch completes. Thereby requiring a new (redundant) read to meet the request.

High latency results

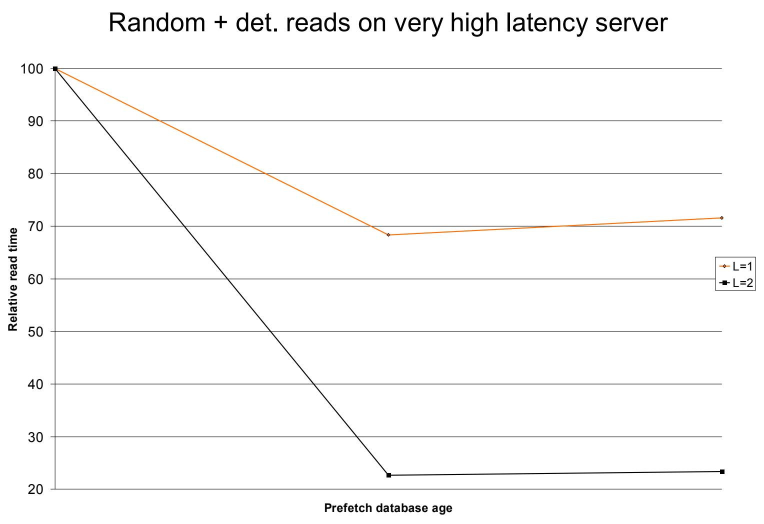


Figure 14.

By using a high latency server the true benefits of prefetching can be seen. On the high latency server the prefetch request completes within the same period that the predecessor read completes, allowing the succeeding read operation to complete from cache. When a clear pattern exists using a lookahead value of 2 means even larger performance increases, however, again, this test was designed with an unrealistically clear access pattern.

Cache performance

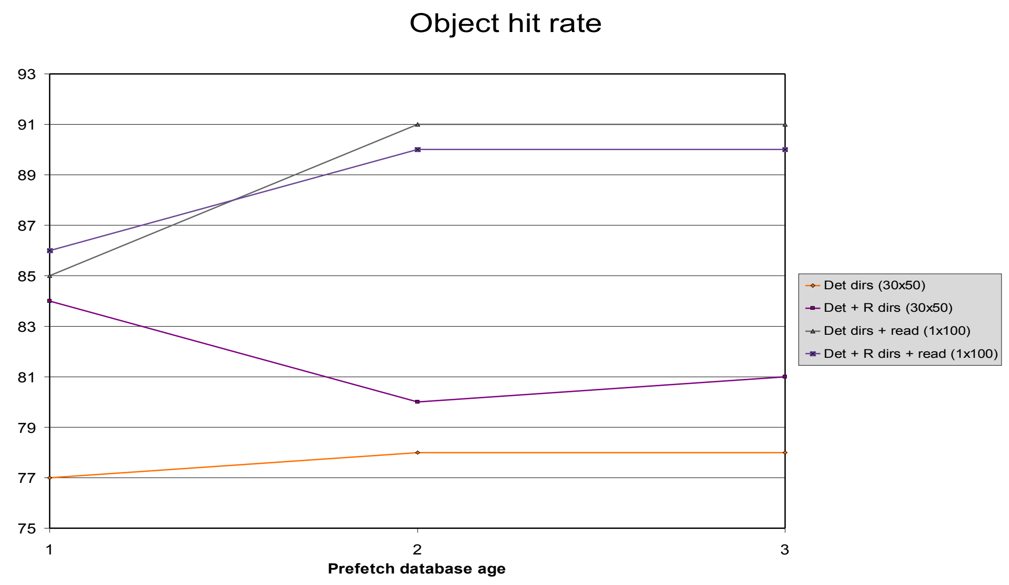


Figure 15.

Figure 15 shows how when using a history record type of Files only there is a minimal improvement in object hit rate when only accessing directories as no history is created this way. When files are read the design means that the parent directory of the opened file and any parents folder above this are preread. Therefore this drastically improves the hit rate as the folders are retrieved as a side effect of the file read operations.

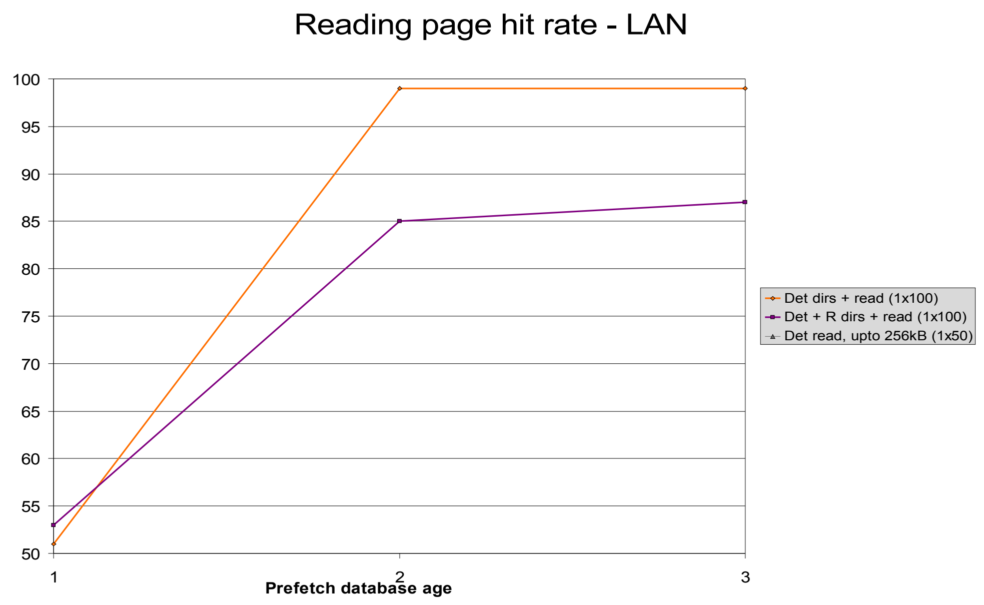
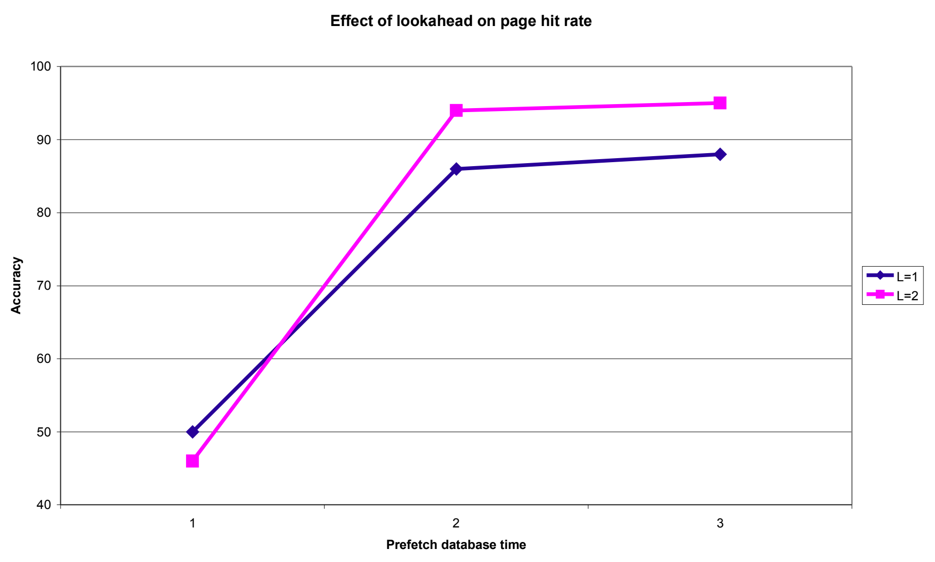


Figure 16.

When there is a definite pattern the page hit rate is as expected very high, while when random data is mixed in it takes a number of database iterations before a sufficient level of stability is reached in the history data for accurate predictions to be made.

Figure 17.



In Figure 17 the effect of different lookahead values is examined. As expected increasing the lookahead value with a clear data access pattern will cause the cache hit rate to go up, in the test case by roughly 10 percentage points. Conceivably the lookahead could be increased up to a point where the the average delay between accesses means that the number of prefetches cannot be completed during that interval. In the real world there will of course be a cost to this, in terms of incorrect prefetches and the bandwidth they use up. A refinement of the prefetcher might be to dynamically adjust the lookahead value depending on the confidence it has on the prediction. This might be achieved by keeping track of in how many instances a recorded successor was prefetched for a given predecessor and that that successor was an accurate prediction.

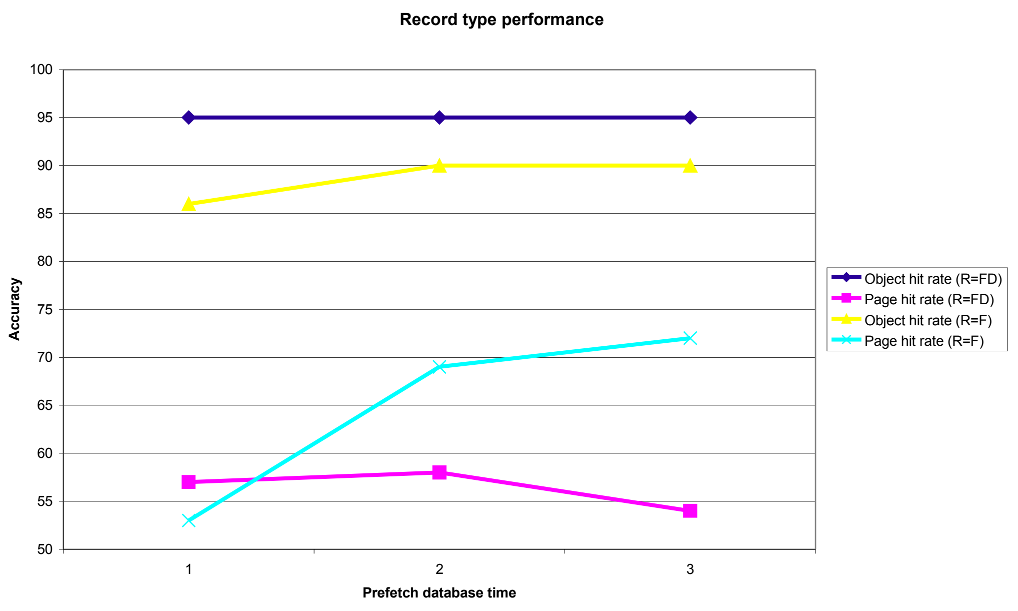


Figure 18.

During the early stages of the implementation of the prefetcher disappointing performance was observed for the page hit rate. Despite a definite pattern existing in one of the tests the hit rate failed to show any improvements over no precaching. When a trace of the file access was examined the reason was identified; the manner in which Windows accesses and traverses files meant that in many cases a directory was opened, a file inside the directory opened, then the parent directory opened a second time before the original file was read. This meant that when both directory opens and file opens were recorded as successor information the pattern was obfuscated by the repeated directory opens. Changing the successor to record only opens to files meant there was a clear link from one file to another with no intervening directory opens. The significant improvements in cache page hit rates demonstrate this crucial aspect of a prefetcher's behavior. The downside of recording just file opens however is also seen, no directories are prefetched. However, this is less of a concern in this particular implementation than it seems, as to open a file the SFTP driver must get a listing of the parent directories of the file, thereby as a side effect prefetching many relevant directories.

Conclusions and future work

This project had aimed to do five things:

1. Bring the FIFS framework up to date with changes to the Windows operating systems
2. Demonstrate the use of the FIFS framework as a usable solution for networked file system drivers
3. Investigate possibilities for predictive caching within networked file systems drivers
4. Verify the completeness and usability of the Secure File Transfer Protocol in networked file system drivers
5. Create a novel, complete, and useful alternative approach to securely accessing remote files

With respect to the first aim, the project successfully analyzed the required changes. Overcoming difficulties involved with tracing packets by the improvement of the NetBIOS packet dumping layer inserted between the operating system and FIFS. It is hoped that this approach to analyzing the FIFS system will be of use to any future developers of FIFS to keep up to date with Microsoft changes or to add new features to FIFS.

In implementing the SFTP driver for FIFS and proving its practicality this project has clearly fulfilled the second aim. It is worth noting however, that FIFS is not the ideal solution for an optimum performance driver, as the results for LAN servers show, but is better suited to high latency, low bandwidth applications. This means that it is ideally suited to Internet-medium based file systems or other non bulk-performance crucial applications. Bulk performance, however, is by no means severely crippled, achieving on average half the speed of a natively implemented driver. The considerable benefits of implementing a user space driver in contrast to delving into kernel mode systems which require a whole new level of understanding for most developers means that the FIFS framework is overall a very desirable platform.

This project has attempted to demonstrate, practically, the improvements that predictive caching can make to networked file systems, especially with very high latency mediums. Results show that significant improvements in performance are attained with the design of the Weighted Best-K-of-M prediction algorithm that exploits inter operation time periods to gain a higher degree of stability to spurious data in trying to model future file accesses based on a single previous file accesses, as well as taking into account global context.

In verifying the suitability of the Secure File Transfer Protocol as a networked file system driver protocol this project has highlighted a number of serious concerns and attempted to propose solutions to them that might be worked into future versions of the protocol. In spite of this alternative solutions to the problems encountered were implemented that utilize available features.

In creating a novel utility this project failed due to the recent release of the SftpDrive program, however it was shown that the project produced a version that had a considerable performance in most areas, especially when accessing high latency SFTP servers. Due to time constraints it was not possible to produce a complete file system implementation however because of the implementation's request framework and packet system this would be a relatively straight forward task to do given sufficient time, and is certainly something that once completed, along with further finishing for server management would result in a commercially viable product that would compete very well with the existing solution on the market. Other parts of the project that were designed, specifically, case sensitivity and symlink handling were not implemented but would be a necessity for a finished polished product.

Future versions of the system might be able to make further performance enhancements by investigating possibilities for predictive reading of files by monitoring read offset and size patterns rather than statically prefetching a certain number of bytes from files like in the current version. Performance might also be increased by the use of better cache replacement policies than LRU. Extensive work has been done in this area since the original design of the LRU, for example Megiddo and Mocha [21] considered the problems of cache management and responded with the Adaptive Replacement Cache (ARC) that was shown to have many advantages of more naïve cache managers.

References

[1] Almeida, Danilo: FIFS: A Framework for Implementing User-Mode File Systems in Windows NT. In Proceedings of the 3rd USENIX Windows NT Symposium (1999), available at http://web.mit.edu/fifs/www/dalmeida-usenixnt-1999.ps.

[2] WinSCP authors: Transfer Protocols (http://winscp.net/eng/docs/protocols)

[3] Refsnes Data: Browser Statisics (http://www.w3schools.com/browsers/browsers_stats.asp, 2006)

[4] OpenBSD team: SSH usage profiling (http://www.openssh.com/usage/graphs.html, 2004)

[5] Ylonen, T: The Secure Shell (SSH) Protocol Architecture (http://www.ietf.org/rfc/rfc4251.txt, 2006)

[6] Malita, Florin: LUFS (http://lufs.sourceforge.net)

[7] Szeredi, Miklos: FUSE – SSH filesystem (http://fuse.sourceforge.net/sshfs.html)

[8] Spousta, Mirsoslav: SHFS (http://shfs.sourceforge.net/)

[9] Petkovic, Zvezdan: SSHfs: A Remote Secure File System Based on ssh (http://www.cs.wm.edu/~zvezdan/research/780project/sshfs.pdf, 1998)

[10] Hunt, Galen C.: Creating User-Mode Device Drivers with a Proxy. USENIX – NT (1997) (University of Rochester, http://research.microsoft.com/~galenh/Publications/HuntUsenixNt97.pdf, 1999)

[11] Rimer, Matthew Scott: The Secure File System Under Windows NT (Master's Thesis, MIT, http://www.pdos.lcs.mit.edu/exo/theses/rimer/thesis.ps, 1998)

[12] Galbraith, J. et all: SSH File Transfer Protocol (SSH Communications Security Group, http://tools.ietf.org/wg/secsh/draft-ietf-secsh-filexfer/draft-ietf-secsh-filexfer-04.txt, 2002)

[13] J. Griffioen, R. Appleton: Reducing File System Latency using a Predictive Approach. In USENIX Annual Technical Conference (1994)

[14] Kroeger, Long: Predicting Future File-System Actions from Prior Events (1996)

[15] G.H. Kuenning: The Design of the Seer Predictive Caching System. In Mobile Computing Systems and Applications, pages 37-43 (1994)

[16] H. Lei, D. Duchamp: An Analytical Approach to File Prefetching. In USENIX Annual Technical Conference (1997)

[17] J, -F Paris, A. Amer, D. D. E. Long: A Stochastic Approach to File Access Prediction (http://www.cs.pitt.edu/s-citi/publications/2003/snapi2003.pdf, 2003)

[18] K. Brant: Using Multiple Experts for File Prediction (http://www.cs.ucsc.edu/~kbrandt/pubs/cs242_proj.pdf, 2004)

[19] A. Amer, D. D. E. Long, J, -F Paris, R. C. Burns: File Access Prediction with Adjustable Accuracy. In Performance, Computing, and Communications Conference, pages 131-140 (2002)

[20] A. Amer and D. D. E. Long. Noah: Low-cost file access prediction through pairs. In Proceedings of 20th International Performance, Computing, and Communications Conference, pages 27-33 (2001)

[21] N. Megiddo, S. Mocha: ARC: A Self-Tuning, Low Overhead Replacement Cache. In USENIX File & Storage Technologies Conference (FAST) (2003)

Appendix I: SFTP amendments

Directory change notification

```
Inserted:

8.10.  Directory change notification

   In order to receive notifications, the client issues a SSH_FXP_SETNOTIFY
   request. The client MUST have opened the directory previously using the SSH_FXP_OPENDIR
request.

       byte   SSH_FXP_SETNOTIFY
       uint32 request-id
       string handle
       uint32 duration
       uint32 monitor types

   handle
      'handle' is a handle returned by SSH_FXP_OPENDIR.  If 'handle' is
      an ordinary file handle returned by SSH_FXP_OPEN, the server MUST
      return SSH_FX_INVALID_HANDLE.

   duration
       'duration' is the time in microseconds from the reception of the
       request during which that the client MUST be provided with notifications of changes
to the entries of the directory. Changes must be notified via SSH_FXP_CHANGENOTIFICATION
messages.

   monitor types
       A bit field of change types that the client is requesting to be informed of.
Multiple types are allowed.

   The server responds to this request with a SSH_FXP_STATUS message. Servers SHOULD
return SSH_FX_OK if directory change notifications are supported or SSH_FX_OP_UNSUPPORTED
if not.

   Notifications of changes to the directory are issued with zero or more
   SSH_FXPCHANGENOTIFICATION messages.

        byte   SSH_FXP_CHANGENOTIFICATION
        uint32 request-id
        string handle
        uint32 change type

   request-id
       The 'request-id' of the the SSH_FXP_SETNOTIFY request from the client

   handle
       The 'handle' to the directory that is being monitored.
   change type

       The change that has occurred in the directory. The server may

       include multiple change types or send each change separately. Each notification
means one more of each type of change has occurred.


   Change type codes:


       SSH_FX_CHANGE_REMOVED        1

       SSH_FX_CHANGE_INSERTED              2

       SSH_FX_CHANGE_ACCESSED              4

       SSH_FX_CHANGE_MODIFIED              8

       SSH_FX_CHANGE_ROOT_REMOVED   16


    SSH_FX_CHANGE_REMOVED

       Indicates that an entry has been removed from the directory.


    SSH_FX_CHANGE_INSERTED

       Indicates that a new entry has been created in the directory.


    SSH_FX_CHANGE_ACCESSED

       Indicates that an entry has been opened in the directory.


    SSH_FX_CHANGE_MODIFIED

       Indicates that an entry was written to in the directory or

       that it has had some of its attributes modified.


    SSH_FX_CHANGE_ROOT_REMOVED

       Indicates that the containing directory was removed from the parent directory.


Insert the following paragraph at the end of 8.1.3.


    If the handle is one previously returned in an SSH_FXP_OPENDIR message and it is one
that the client has requested directory change notifications with a SSH_FXP_SETNOTIFY
message then the server MUST cancel the change notification request and no longer provide
the client with change messages for the opened handle.
```

Opportunistic locks

```
Change the paragraph in 5.4 headed 'supported-block-vector'

      16-bit masks specifying which combinations of blocking flags are
      supported.  Each bit corresponds to one combination of the
      SSH_FXF_BLOCK_READ, SSH_FXF_BLOCK_WRITE, SSH_FXF_BLOCK_DELETE,
      SSH_FXF_BLOCK_ADVISORY, and SSH_FXF_BLOCK_OP bits from Section
      7.1.1.3, with a set bit corresponding to a supported combination
      and a clear bit an unsupported combination.  The index of a bit,
      bit zero being the least significant bit, viewed as a five-bit number,
      corresponds to a combination of flag bits, shifted right so that
      BLOCK_READ is the least significant bit.  The combination
      `no blocking flags' MUST be supported, so the low bit will always be
      set.

Append to 8.1.1.3

       SSH_FXF_BLOCK_OP          = 0x00010000

Append after paragraph headed 'SSH_FXF_BLOCK_ADVISORY'

  SSH_FXF_BLOCK_OP

      If this bit is set, the above BLOCK modes are opportunistic.  In
      opportunistic mode, the server must lock the file or directory using
      the above BLOCK modes until another client or process requests any type
      of matching lock mode (READ, WRITE, or DELETE), at which point the
      original opportunistic lock is relinquished and the new lock request is
      denied.

Append to 8.8

   The SSH_FXP_LOCK_LOST response is sent to clients by the server in the special case of
   when a SSH_FXP_BLOCK with the SSH_FXP_BLOCK_OP flag set is requested and the lock
   is subsequently lost to another client or process.

       byte   SSH_FXP_LOCK_LOST
       uint32 request-id
       string handle
       uint64 offset
       uint64 length

   handle
      'handle' is a handle returned by SSH_FXP_OPEN or SSH_FXP_OPENDIR.

   offset
      Beginning of the byte-range lock that was lost.
```

```
length
   Number of bytes in the lock range that was lost. Zero if the entire file was locked.
```

Appendix II: Benchmark result tables

LAN server	SftpDrive	Samba 3.0.2b	P=0,O=0, L=0	Sftp-baseLine			P=5000,O=500,L=0		
Det dirs	3.25	3.67		1.8	-	-	2.11	78	-
Det + R dirs	5.67	3.92		1.8	-	-	2.98	84	-
Det dirs + read	2.92	2.95		11	-	-	14.17	85	54
Det + R dirs +	2.75	2.91		11	-	-	13.98	85	54
Det read, upto 256kB (1x50)									
WAN server									
Det dirs (1x50)	1221	NA		1.8	-	-	101	81	-
Det + R dirs	1210	NA		1.8	-	-	102	80	-
Det dirs + read	418	NA		11	-	-	112	84	49
Det + R dirs +	387	NA		11	-	-	103	85	50
Det read	1144	NA		11	-	-		88	49
Very high latency server									

File + folder record			File + folder record			File + folder record		
P=5000,O=500,L=1,			P=5000,O=500			P=5000,O=500		
2.78	90	-	2.48	91	-	2.3	91	-
3.5	95	-	3.06	95	-	2.88	95	-
16.4	93	54	14	94	55	13.9	94	55
22.4	95	57	13.78	95	58	14.24	95	54
123	90	-	104	91	-	106	91	-
100	91	-	94	92	-	86	92	-
104	93	48	116	94	50	108	94	50

File record			File record			File record		
P=5000,O=5			P=5000,O=50			P=5000,O=50		
2.21	77	-	2.03	78	-	2.13	78	-
2.71	84	-	2.59	80	-	2.66	81	-
15.23	85	51	21.35	91	99	21.18	91	99
13.71	86	53	27.23	90	85	28.9	90	87
90	81	-	89	82	-	88	82	-
93	81	-	93	82	-	83	82	-
110	84	49	144	89	99	133	90	99
100	85	50	112	89	86	112	88	88
431	89	49	375	93	99	374	94	99
5060			3460			3624		

File record			File record			File record		
P=5000,O=5			P=5000,O=50			P=5000,O=5		
90	83	-	88	82	-	91	82	-
89	82	-	88	82	-	90	81	-
114	84	49	136	92	99	136	92	99
104	86	46	145	91	94	187	91	95
394	88	48	398	93	91	345	94	89
3539			803			828		

List of figures

List of tables