Operating System and File System Monitoring : A Comparison of Passive Network Monitoring with Full Kernel Instrumentation Techniques

by

Andrew W. Moore

A thesis submitted in fulfillment of the requirements for the degree of Master of Computing

Department of Robotics and Digital Technology

MONASH UNIVERSITY

1995

Contents

List of Tables

List of Figures

Abstract

System monitoring is the process by which information about computer systems and users of those systems is collected. It is carried out typically to assist in improving the operation of current systems and to aid in the development of future computer systems. This thesis compares and contrasts two system-monitoring techniques: full kernel instrumentation, where information is obtained through the placing of instrumentation code into the computer's operating system (kernel), and passive network monitoring where data associated with one or more computer systems attached to a network are extracted by monitoring the communications traffic exchanged by those computers with others on the network.

In order to achieve this comparison and the contrast of the two system-monitoring techniques, two such systems were implemented, operated in tandem and the results then compared. A kernel-instrumentation system was ported from an earlier revision of the UNIX operating system to a more up-to-date version. This involved developing a working knowledge of the kernel system in general and the development of programs to process the records generated by the kernel instrumentation into a format that could be then be compared with the passive network monitoring system. The passive system was substantially complete as received although some support software was written. Processing software also had to be written for summarising the records of each technique and for producing a comprehensive analysis of the trace record. An understanding of the rule base of the post-processing software was achieved by thorough investigation including instrumentation and trial operation. A simplified cache simulator was constructed which, although it could not be integrated into the passive network monitoring software, aided in an understanding of the cache system.

Full kernel instrumentation has been a popular technique because of its comprehensive nature but it suffers, among other things, from the necessity to have the kernel source-code available, from the need to make changes to the system being monitored and from the impact it can have on that system. In contrast, passive network monitoring can be conducted in a non-invasive, platform-independent manner which involves no changes to the operation of the monitored machine. Passive network monitoring has the potential to be used in place of full kernel instrumentation for many tasks, and even though it was unable to give results comparable with full kernel instrumentation in all cases, it is able to give good predictions of many values when compared with those derived from full kernel instrumentation. This was particularly true in areas related to the writing of data. This thesis also notes discrepancies between the result from the two system-monitoring methods and discusses ways in which those discrepancies can be reduced or eliminated.

Acknowledgements

Firstly, I would like to thank my parents, Alan and Dorothy Moore, and my brother, Nicholas, for their endless and unwavering love and encouragement. Mere words cannot begin to express my appreciation and love.

I could not have had a better combination of mentors than my supervisors, Tony McGregor and Jim Breen, They are both true and steady friends who willingly give their time and guidance.

This thesis would have been at least different and, perhaps, impossible without access to a number of programs. The starting point of this thesis was when Matt Blaze made the original `rpcspy/nfstrace` suite of software available to me. He then passed on changes and additions as well as words of advice and encouragement. I am more than grateful to him.

Songnian Zhou and Chris Siebenmann gave me access to the original `snooper` code and gave unstintingly of their time to answer many questions and to offer advice. Michael Dahlin willingly assisted me by supplying changes his work group had made to `rpcspy/nfstrace`. My thanks to you all.

I am grateful to Randy Appleton, Mary Baker, Charles Briggs, Dan Eaves, Rick Floyd, Simon Hill, Rick Macklem, John Ousterhout, Alan Rollow and Margo Seltzer for unhesitatingly giving of their enthusiasm, sought-after papers, ideas and the confirmation that the work of this thesis was worthwhile.

I owe an eternal debt to the staff of the Department of Robotics and Digital Technology and to the users of the machines used for testing for putting up with crashing clients and low performance networks.

My thanks to Andrew Lysikatos for organising access to the Hewlett Packard network analyser and to Kathy Ching for giving so much of her time to assist in obtaining the much-needed source code which made this thesis possible.

I extend thanks to my friends, in particular Michelle Judson, Cameron Blackwood, Rik Harris and Chris Beecroft, for making the past years both interesting and enjoyable and to Cameron and Chris for proof-reading.

Finally, special thanks to Ralphe Neill, who willingly read through and commented on early revisions of this thesis.

Chapter 1

Introduction

1.1 System monitoring

System monitoring is indispensable for the development and refinement of computer systems. Systems monitoring is also important in providing assistance in the day-to-day operations of the systems. It is only through system monitoring that it is possible to quantify changes in the operation of a system or to determine how well a new implementation meets its specifications.

Workload statistics, such as the average amount of data a user transfers in a given time are important factors when evaluating both current and future systems. Differences in such values, occurring when older studies are compared with more current ones, can indicate the reasons for such changes and allow meaningful projections of future trends. For example, a projection of the amount of data transferred per user in a given time can then be used for determining minimum required network-bandwidths and disk transfer rates.

The original AT&T UNIX [81] operating system and its descendants [52] are widely used, particularly in research and educational establishments, and have been the subject of many systems-monitoring studies. In recent years, the area of distributed systems has grown rapidly and distributed operating systems based upon UNIX or on a UNIX-style framework have become common. A widely-used system which allows distributed file access is the Network File System (NFS) [87]. There have been relatively few studies of NFS but they have had a wide impact because of the system's popularity.

In this thesis, a method of analysing distributed systems, passive network monitoring, is compared with full kernel instrumentation and the results of each system are compared. Similarities and differences in the data are noted and discussed. Disparities between the two systems are analysed and explained, and methods by which the results of passive network monitoring can be made to parallel more closely those of kernel instrumentation and achieve greater accuracy are outlined.

1.2 Kernel instrumentation

A common method of system monitoring involves the instrumentation of the operating system. Operating system instrumentation or kernel instrumentation requires the installation of extra instructions into the kernel to record desired information about the operation of the kernel and the services it provides.

However, kernel instrumentation has a number of drawbacks, as seen in following list (adapted from Mogul et al. [65]):

- code which is to reside in the kernel is difficult to write and debug,
- kernel source-code is not always available,
- each time an error is found, the kernel must be recompiled and the machine rebooted,
- errors in the kernel code are likely to cause system crashes,
- functionally-independent kernel modules may have complex interactions over shared resources,
- kernel-code debugging cannot be done during normal machine operation; specific development time must be scheduled, resulting in inconvenience for users sharing the system and odd work hours for system programmers,
- sophisticated debugging and monitoring facilities available for developing user-level programs may not be available for developing kernel code.

A particular set of kernel instrumentation will suit only one version of a specific operating system. Each operating system and each version of each operating system requires specific code to be written for it.

It is common for the results of kernel instrumentation to be recorded by the instrumented machine. This might be difficult in an environment of networked machines which did not have local disks. Kernel-instrumentation data would have to be collected from many machines simultaneously when monitoring a distributed system and each machine might have to be equipped with a local disk. Furthermore, the collection of kernel instrumentation data on the machine instrumented will change the results themselves.

1.3 Passive network monitoring

In many cases, the statistics collected by kernel instrumentation may also be collected satisfactorily using passive network monitoring.

Use of distributed systems has increased in popularity in recent years. The reasons for this are, firstly, the cost of high performance workstations and memory components has decreased, and high-speed computer network technologies are now widely available at moderate cost. Additionally, the interactive service obtained from large centralised computer services is often of poor quality with long, unpredictable response times, restricted user interfaces and difficulty in configuring hardware and software to users' needs. Finally an increasingly diverse range of applications and facilities is required by users. As a result of this, the option of using a network of many smaller, more powerful workstations sharing common file systems and printing resources has become more popular than the installation of a single large computer resource.

Because of these developments, distributed systems are in increasingly common use, and with an increasingly common usage, there has developed a need to monitor distributed systems. Distributed systems are not as easily monitored using traditional techniques of kernel instrumentation, this is because problems such as the complexity in monitoring, in a distributed system each machine needs to be instrumented and the trace information must be collected from each machine and then the traces from each machine synchronized together. Additionally, kernel instrumentation has drawbacks associated with needing access to the operating system source-code and the need to instrument the source-code itself.

As a result of having many vital activities conducted over a local network, passive network monitoring, a method of monitoring distributed systems, can often be used in place of kernel instrumentation. Passive network monitoring has many advantages over kernel instrumentation such as the fact that no modifications to the operation of the distributed system are required. It is independent of the hardware on which the distributed system is based and of the operating system itself, and can collect data synchronously and simultaneously about every machine on a particular network, with the collection being independent of the machines being monitored.

While there are drawbacks to passive network monitoring, in particular a potential lack of accuracy, this thesis seeks to demonstrate that it can be a valid alternative to kernel instrumentation for certain system-monitoring work. Additionally, the current drawbacks of passive network monitoring can be removed to provide greater accuracy and utility, and to yield a system that could replace kernel instrumentation in many areas. The passive network monitoring system analysed in depth is designed to report information about NFS-based distributed file systems, although passive network monitoring can be used for the instrumentation of any network-based system.

1.4 Work performed

In order to achieve this comparison and contrast of the two system monitoring techniques, two such systems were implemented, operated in tandem and the results then compared.

A kernel instrumentation system was ported from a version developed for an older revision of the UNIX operating system, into a more up-to-date UNIX revision. This port involved developing a working knowledge of the kernel system in general and the installation and updating of the instrumentation system itself. During this development, unresolved problems in the original implementation were also dealt with, including a mutual exclusion problem in data-recording. An additional, substantial, part of the kernel instrumentation system was the development of program code to process the records generated by the instrumentation system into a format that could be then be compared with the passive network monitoring system.

The passive network monitoring system was substantially complete as received, although an amount of enhanced software was added. In particular, software was written to assist in the management of the large amount of data potentially produced; this software incorporated check-pointing and compression routines. An understanding of the rule base of the post processing software (`nfstrace`) was achieved by thorough investigation including instrumentation and trial operation. The relationship between network utilisation and passive network monitoring efficiency was also determined.

For both systems an extended period was spent ensuring the correct operation of each technique. This was principally done through trial operation, particularly in a controlled environment.

In order to compare and contrast the two monitoring methods, processing software was written to summarize the records provided by each technique and to produce comprehensive analyses of the trace records. This software also required extensive assessment to ensure correct operation.

In developing a plan of improvements that could be made to the passive network monitoring system, a simplified cache simulator was constructed. While not able to be integrated into the passive network monitoring software (`nfstrace`), this system aided in an understanding of the cache system and was a valuable tool in the development of the cache simulator concept.

1.5 Thesis organisation

The rest of this document is organised as follows.

Chapter 2 provides background information about approaches to the monitoring of operating systems, with particular reference to passive network monitoring and kernel instrumentation techniques. The chapter also provides background material on aspects of operating systems in general and NFS-based distributed file systems in particular.

Chapter 3 discusses the place of system-monitoring research and discusses various studies categorised by the monitoring techniques used. Additionally, this chapter discusses research that has used the results of other system-monitoring studies. By looking at work following from system-monitoring studies seeking to show that a significant contribution of system-monitoring research is the further research opportunities it may reveal. Finally, this chapter discusses certain concepts specific to the monitoring of operating systems and file systems in particular, such as the file open-close session.

In Chapter 4 the implementation of full kernel instrumentation is outlined. The full kernel instrumentation system `snooper` is discussed with particular reference to the modifications required to port the software to the required system. The methods used to transform the results of `snooper` into an appropriate format are also discussed, along with the effects the `snooper` modifications had on the monitored system.

The passive network monitoring system, `rpcspy/nfstrace`, is outlined in Chapter 5. This chapter discusses the operation of the passive network monitoring system, the relationship between its two principal components and the impact passive network monitoring may have on a monitored system.

Chapter 6 presents and analyses the results of a comparison of the kernel instrumentation and passive network monitoring implemented on a single machine instrumented with both systems. The comparison is based on measures commonly used in systems-monitoring research such as the amount of data a user transfers in a given time, the duration of an open-close session and the total amount data transferred by a computer system in a given time. This chapter establishes the areas of shortcoming in the passive network monitoring implementation and the divergence in the results.

Chapter 7 discusses changes that can be made to improve the accuracy of `nfstrace` and decrease the difference in the results obtained by each monitoring system. The chapter then discusses future possibilities for system monitoring using the passive network monitoring technique.

Chapter 8 presents a summary of the results and findings of this thesis and discusses potential future work both to extend the comparisons made for this thesis and the future uses of passive network monitoring.

Chapter 2

Background

2.1 The UNIX file system

2.1.1 Files

The UNIX operating system and its derivatives have the file as their basic construct [81, 52]. A file can be any collection of data. It could be the text of a thesis or the machine instructions of a program. A directory is a file containing reference information about the location of other files.

Associated with a file is the data it contains and an index node (usually referred to as an i-node). An i-node contains information about the file such as which user owns it, its size, where on the disk the file's data is located, when the file was last accessed, etc.

Figure 1 shows the relationship between a buffer of data (`buffer`), at the user level, the system block buffers and the physical disk sectors (sector `1` and `2`). The sector is the smallest working unit of the physical device. All operations involving the physical device must involve sequential runs of data of this size. The block cache of the UNIX system also works in sequential bytes of data of this size. In this diagram a user buffer is shown to be part of a file. That file extends over four disk sectors. The user's buffer of interest (`buffer`) extends over two of these sectors.

Although a user may wish to change only a single byte on a disk, the underlying hardware can read and write only in integral units of physical storage, sectors. The system must, therefore, read the sector containing the byte to be modified, replace the

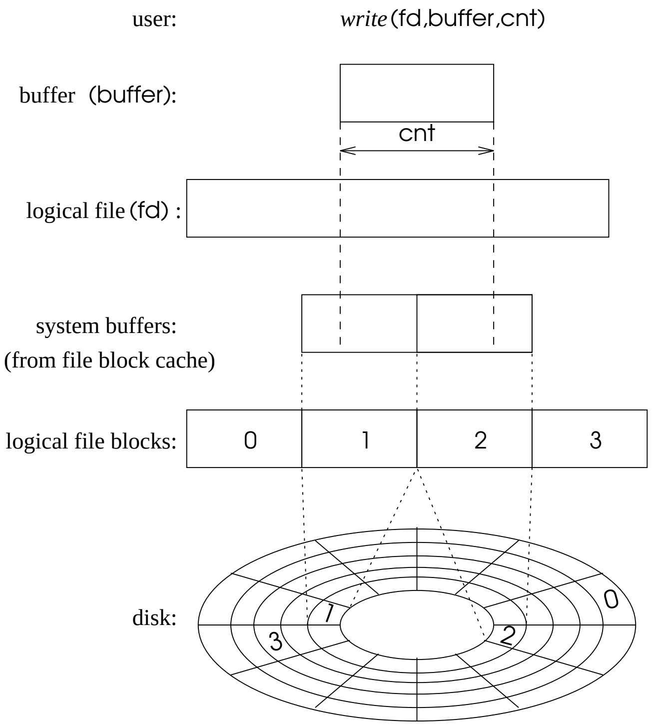

Figure 1: This figure shows the relationship between a buffer of data, at the user level, the system block buffers and the physical sectors of the disk.

affected byte and write the sector back to the disk.

Processes may need to read data in sizes smaller than a disk block. The first time a small read is required from a particular disk block, that block will be transferred from the disk into a kernel buffer. Successive reads of parts of the same block then usually require only copying from the kernel buffer to the memory of the user process.

Multiple small writes are treated similarly. A cache buffer is allocated when the first write to a disk block is made and succeeding writes to parts of the same block are then likely to require only copying into the kernel buffer with no disk I/O.

In addition to providing the abstraction of arbitrary alignments of reads and writes, the block-buffer cache reduces the number of disk I/O transfers required by accesses to the file system. System-parameter files, commands and directories are read repeatedly so their data blocks are usually in the buffer cache when they are needed. The kernel does not need to read them from the disk every time they are requested.

The situation is more complicated in the case of cached writes. The data on the disk will be incorrect and data will be lost if the system crashes while data for a particular block is in the cache and has not yet been written to disk. (Critical system-data, such as directories, are written synchronously to disk to ensure file system consistency). Block writes are forced periodically for dirty buffers, to alleviate this potential problem of data loss.

2.1.2 File structures

UNIX organises files into *tree* structures called file systems. Figure 2 shows a single file system. The single file system has an inverse tree structure.

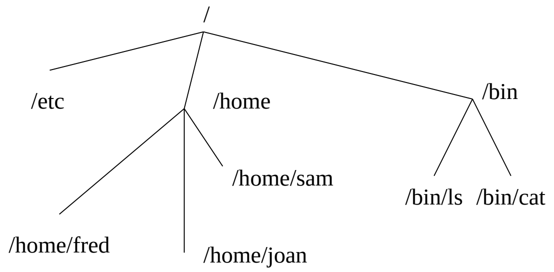

Figure 2: A single file system tree.

UNIX enables file systems to be *grafted* together to form (from the user's perspective) large tree-structures of files. The user need not know on which disk a file is physically located to be able to access that file. Figure 3 shows how several, different file systems can be grafted together into a single tree structure. File system 2 is *mounted* onto file system 1. The directory **/home/joan** is referred to as the *mount point* of file system 2. The resulting composite file-system tree is shown as file system 3.

Under UNIX, there are a number of different types of file systems supported. For example, a file system could be located on local disk drives, CD-ROMs or on a remote system, accessed via networks using a network based file system such as the Network File System (NFS). As a result, file systems grafted together as in Figure 3 may be of different types as well as residing on different disks or parts of disks.

2.1.3 The vfs/v-node interface

In order to implement various types of file systems under UNIX without requiring users to modify their programs, or make substantial changes to the operating system

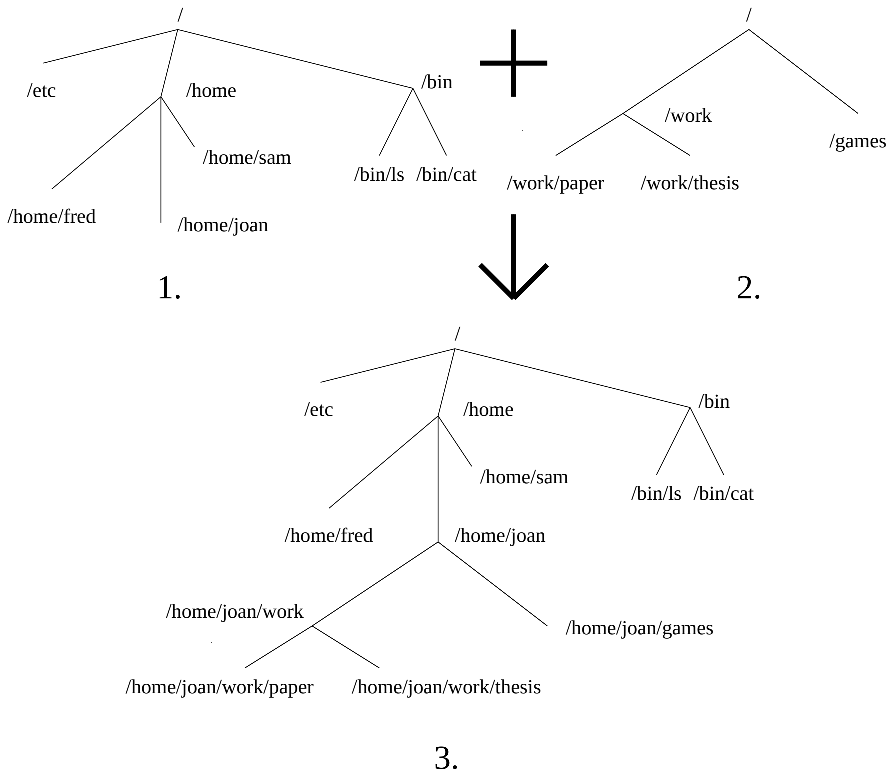

Figure 3: File system 2 is grafted onto file system 1, to form the composite tree shown as file system 3.

itself, Berkeley Software Distribution (BSD) and variants of UNIX, often BSD-derived, such as SunOS and Ultrix use the vfs/v-node system. vfs, which stands for virtual file system, combined with the v-node, a virtual i-node, enables the operating system to perform a generic set of operations on a particular file independently of the type of file-system upon which it resides.

Prior to the introduction of the vfs/v-node system, the contents of an i-node were identical whether in memory or on disk. With the introduction of the v-node, additional data were added to the structure when in memory, making the v-node a *super* version of the i-node. In this thesis there will also be reference to the term g-node; this is the Digital Ultrix term for a v-node. Ultrix also refers to the virtual file system, vfs, as the generic file system or gfs. A g-node or v-node can be considered equivalent. In turn, both a g-node or v-node can be considered as *super* i-nodes.

Using the vfs/v-node interface, the kernel directs commands to an appropriate, file-system-specific part of the operating system. When a request is made on a particular file the kernel will use regular file-system operations to access that file if that file is on a file system which is local to the user. For a file on a networked file system the kernel will use network operations for its access.

In the NFS, a server makes file systems available for use by users on client machines. A machine is said to be a client of another if it mounts a file system physically located on other machine. The file system requests are passed through a network from the client to the server. The server then performs the requested operation and returns the result to the client. For example, in a read operation performed on the client, the read operation would be transferred to the server. The operation would be processed by the server and the results returned to the client.

Figure 4 depicts how the client and server communicate through the network. The server performs the operations on its local file system that were requested by the client in exactly the same manner as they would have been if they had been requested by users on the server itself. The client is shown with a local file system in addition to the connection to the server although such a file system is not a necessity for a client.

Remote Procedure Call (RPC) and the eXternal Data Representation (XDR) are used in the communications of the NFS client and server [55, 54]. They are discussed more fully in Section 2.2.1.

2.2 NFS

The following section describes the NFS. Much of this information has been drawn from *NFS Implementation* by Sandberg et al. [87], the *NFS:Network File System Protocol Specification* by Nowicki [70] and *Overview of the Sun Network File System* by Walsh et al. [112].

The NFS protocol, as well as the standards for Remote Procedure Call (RPC) [55] and eXternal Data Representation (XDR) [54] were developed by Sun Microsystems. To enable a wide-scale implementation and use of this system, Sun Microsystems made these standards publicly available, and has made available to operating system developers a reference implementation of the NFS system.

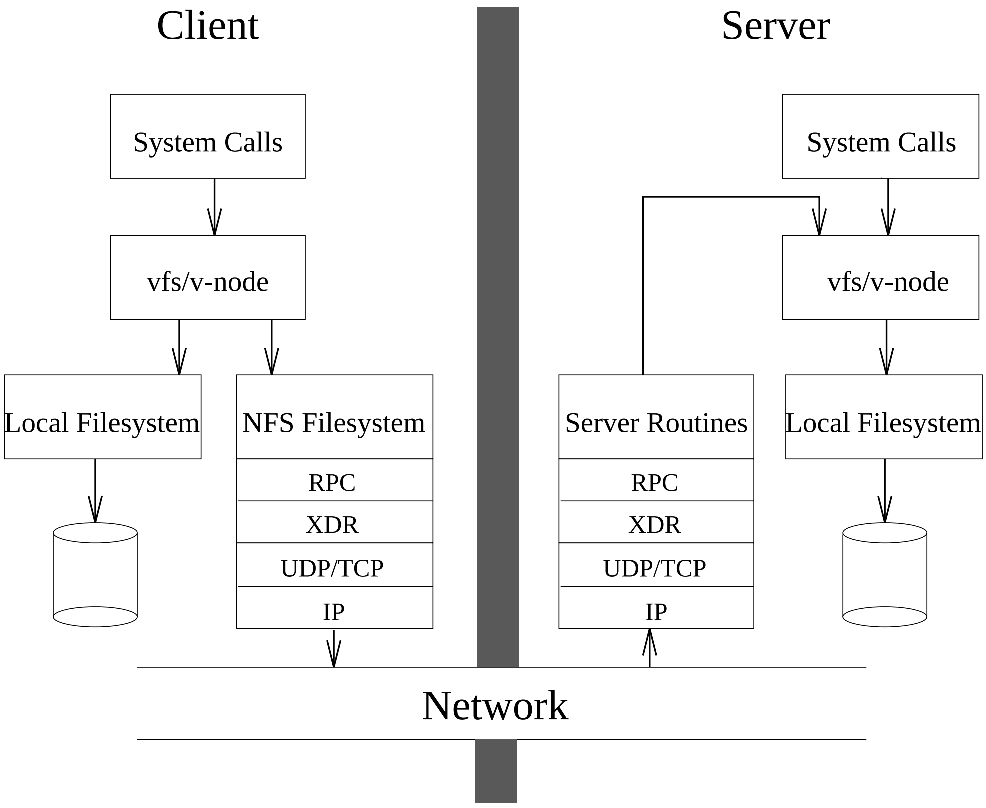

Figure 4: A block diagram of a typical NFS client and server layout.

The design goals for NFS included:

- **machine and operating system independence:** while NFS was designed under UNIX, it should be (and has been) implementable under most operating systems,
- **malfunction recovery:** an objective of NFS was to minimise the difficulty of recovering from a server or client malfunction,
- **UNIX file semantics to be maintained on clients:** that is, to maintain transparent access to UNIX machines, clients must maintain UNIX file system semantics,
- **reasonable performance:** NFS would not become commonly used if existing networking utilities were faster to use. NFS was expected to be no slower than about 80% the speed of a local disk.

These goals lead to the design of a stateless distributed file system. By stateless it is meant that every request made of an NFS server is totally self-contained (idempotent) and repeatable.

2.2.1 NFS protocol

The NFS protocol [70] uses the Sun Remote Procedure Call (RPC) mechanism [55]. The use of the RPC system insulates NFS from the intricacies of server-client communications, data formats and communications reliability, thus allowing it to deal exclusively with file-system-related matters.

An RPC call is synchronous and, as a result, the RPC call will block until it can be completed. A function run from a remote machine will wait for the results to be returned from the remote machine before resuming program execution. This results in an RPC call behaving like a local procedure call and, with a few important exceptions such as handling machine-specific parameters, can be treated as such.

Statelessness has important advantages for crash recovery. In a state-oriented Distributed File System (DFS), a server crash would mean the loss of all information about the state of files which clients may have been accessing. It is for this reason that elaborate, crash-recovery protocols are established for servers to recover information about the DFS's state before the crash. This, of course, adds to the complexity to the overall distributed system.

NFS is stateless, each procedure call must contain all the information (parameters) necessary to complete a call. Additionally, the server does not keep track of any past requests. This results in uncomplicated crash recovery. When a server crashes, a client resends NFS requests until a response is received. The server itself does no crash recovery specifically for NFS. Sandberg et al. note that a client would not be able to tell the difference between a server that had crashed and recovered, and a server that was slow to respond.

Table 1 shows the NFS system calls. While most file system operations are represented here, notable omissions include `open` and `close`. The changes introduced by state-oriented operations such as `open` and `close` are kept on the client only, such

null()	do nothing.
lookup()	returns a new file handle and attributes for the named file in a directory.
create()	creates a new file handle.
remove()	removes a file from a directory.
getattr()	returns file attributes.
setattr()	set a file's attributes (permissions, owner, etc).
read()	returns a number of bytes from a particular offset into a file.
write()	writes a number of bytes at a particular offset into a file.
rename()	renames a file.
link()	creates a hard link on the remote file system.
symlink()	creates a symbolic link on the remote file system.
readlink()	reads the string associated with the symbolic file name.
mkdir()	creates a new directory.
rmdir()	removes an existing directory.
readdir()	returns a number of bytes of directory entries from a particular directory.
statfs()	returns information about a file system.

Table 1: NFS file system calls

state-oriented operations are not sent on to the server. As a result, when these operations occur on a client, there is no specific associated NFS activity between client and server.

For NFS to provide transparent, remote access to file systems it must also be independent of system-architecture issues. The eXternal Data Representation (XDR) standard was designed to facilitate communication between computers that use different data representations. This standard overcomes the differences between the way data is represented on different computer architectures. For example, computers can vary in the way each represents the concept of an integer such as varying the order and the number of bits, octets, etc. XDR has specifications for common building blocks from which other values can be created including integers, character strings and floating-point numbers. By using XDR, complex data structures can be machine and language independent.

An NFS call has a related pair of messages; a request and the response (either acknowledged or declined). Each of the messages (a pair for each of the system calls in Table 1) has arguments and returns parameters appropriate to their particular function.

For example, the `read` request message passes arguments relating to a file's identification, where the data is to be read from (an offset into the file) and the amount of data to be read. The `read` reply message contains (in the success case) the attributes of the read file, as well as the data. Because each RPC operation can have only 8 Kbytes of data associated with it, a read (or a write, etc.) request for more than 8 Kbytes of data is broken (by NFS) into two or more RPC requests.

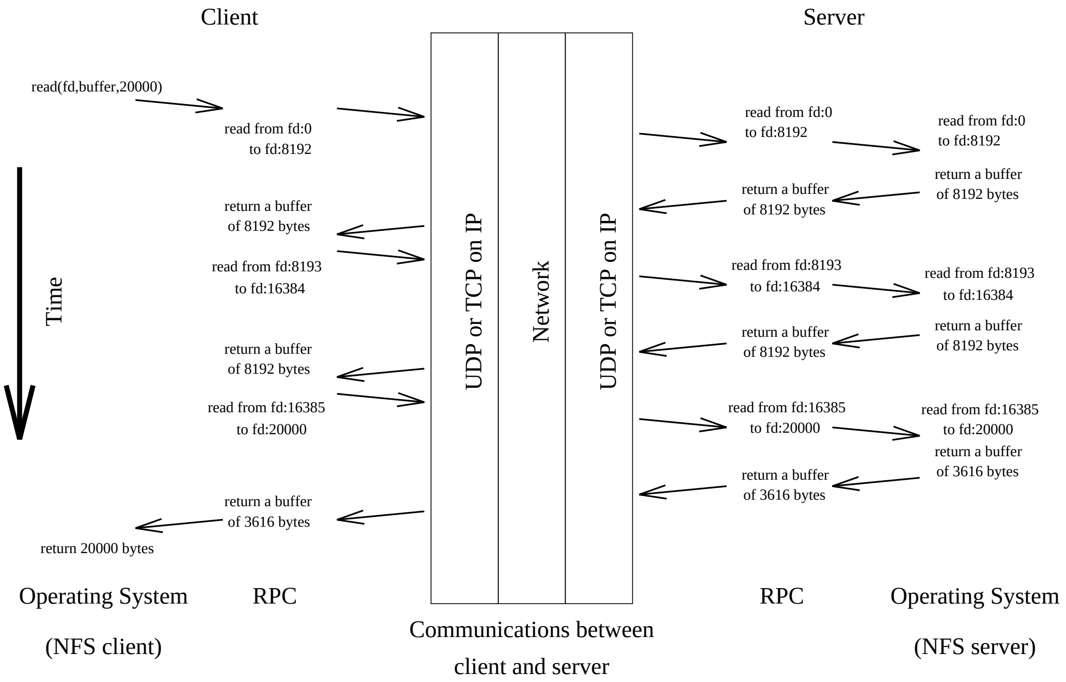

Figure 5: This figure shows how a single operating system request, too large to traverse the RPC layer, is segmented and reassembled by NFS for processing by RPC.

Figure 5 shows this relationship between RPC and NFS more clearly. The NFS interface will, as required, segment a request into manageable pieces (8 Kbytes in this example). These requests are received by the server which, in turn, sends back the results. The NFS interface will then reassemble the replies and return this data to the user via the operating system. Both the requests and replies travel via a communications system. The common communications system used is the Unreliable Datagram Protocol (UDP) [78] or, more recently ([56, 100]), the Transmission Control Protocol (TCP) [77]. Each of these then use the Internet Protocol (IP) [76]. The UDP and TCP communications layers may each fragment/de-fragment the packet from the previous layer but this process is not shown in Figure 5.

2.2.2 Server

As has already been stated, the NFS server is stateless. When serving requests, the standard implies that an NFS server must commit any modified data to stable storage before returning results. For a UNIX system this means that requests which modify the file system must flush all modified data to disk before returning from the (RPC) call. As a result, for a `write` call both the data blocks of the file and the block containing the i-node must all be flushed if there have been any modifications [87].

2.2.3 Client

The client side provides the transparent interface to NFS. For transparent access to remote file systems to function, the locations of files must be independent of the file naming structure. In NFS, the remote server's hostname is looked-up once when the file system is mounted. However, the disadvantage is that remote files are not available to the client until the `mount` is done.

2.2.4 NFS file references

From the perspective of the client, each v-node individually identifies a particular file. The v-node contains enough information to determine which type of file system a file is on, for example: a local disk, a local CD-ROM or on a file system of a specific NFS server.

For NFS, the v-node references a structure called a filehandle which is always provided by the server and used by the client. From the client's perspective the filehandle information is opaque; the client is not required to decode the contents of the filehandle. The file handle can contain whatever information the server requires to identify an individual file, e.g. which the system the file is on. The filehandle implemented in UNIX also contains a reference to the file itself (typically the i-node of the file on the server's file system) and a generation counter to ensure that the client is referring to the correct version of the server's file. Thus, the filehandle forms a unique identification of the file that can be used by both the server and client.

2.3 File system operations

A system call is the interface between a user's program and the operating system. It is the means by which a user's program can perform file system operations such as writing data into files, creating directories, etc.

We will take the `read` system call as an example. This routine, which will read a nominated number of bytes from a particular file. This call forms the front-end of a set of operations that access the desired file. Once the appropriate data has been retrieved, the read routine places it into the buffer nominated by the user.

In order to read data from a file residing on a local UNIX file system, the kernel will gather up the `read` operands from the system call and pass them to the generic `read` interface. These are then passed to the local file system (called UFS, for UNIX File System although most commonly the BSD Fast File System (FFS) [60]) `read` routines. The appropriate type of file-system-`read` routine is automatically determined because it is part of the information denoting the file.

As a result of each file being made up of a number of blocks, the UFS-specific `read` routine will process each of the blocks that makes up the data to be read. If the block size is 8 Kbytes and the read request is 20 Kbytes this could involve up to four blocks: two complete blocks and one partial block at each end. The UFS `read` routine first checks if the block from which it wants data is already in a system buffer. If it is, the data are copied into the buffer space nominated by the user.

Figure 6 shows the flow the kernel will follow to obtain a block of data. For this particular `read` operation, the block resides on a local disk.

In comparison, Figure 7 shows the flow the kernel will follow to obtain a block of data from a file residing on an NFS server. The system cache is checked for the required data. If it is not found, the data is read from the server. If the data is in the cache, and the cache copy is recent, this is returned to the user. If the data is in the cache and the copy was not made recently, a check is made with the server to ensure the copy held in the cache is the latest available. This check will result in either the cache copy being supplied (if it is the latest available). Or it will result in a new copy

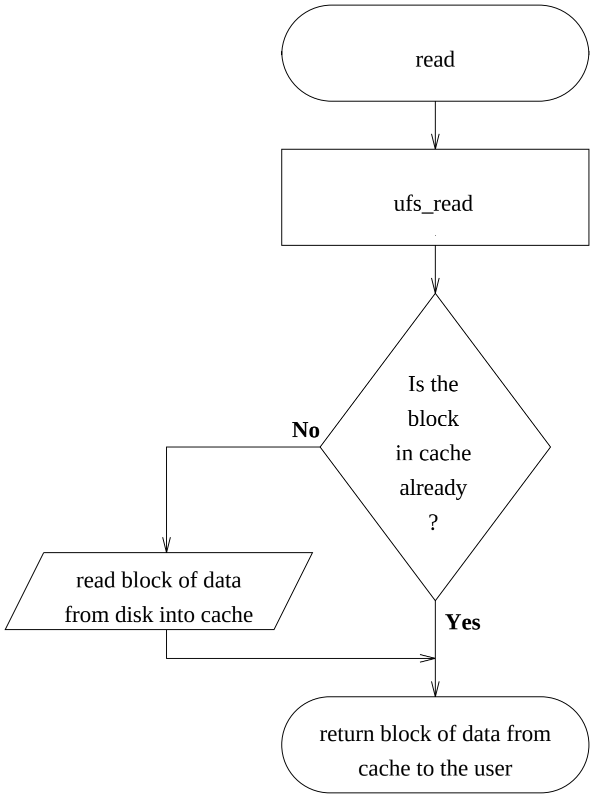

Figure 6: A chart of the flow of actions required in returning a block of data to the user. In this case the block resides on a local disk.

replacing the client's cache copy and this data will, in turn, be supplied to the user.

2.3.1 The block cache

As explained in Section 2.1.1, the block cache improves the performance of disk related activities. In addition to buffering up pending `write` operations and saving data from previous `read` operations, the block cache implementation in UNIX also performs a *read-ahead* to improve performance.

Read-ahead is a technique where the operating system will read the next block to the one actually requested into cache in anticipation of it being required in the immediate future. Numerous studies on caching have found that file accesses tend to be sequential and the possibility of consecutive blocks being accessed in a file is very high [73, 8, 98, 109]. As a result, read-ahead is a very effective technique for improving performance.

The *write-behind*, where modified blocks of data are not immediately written to the

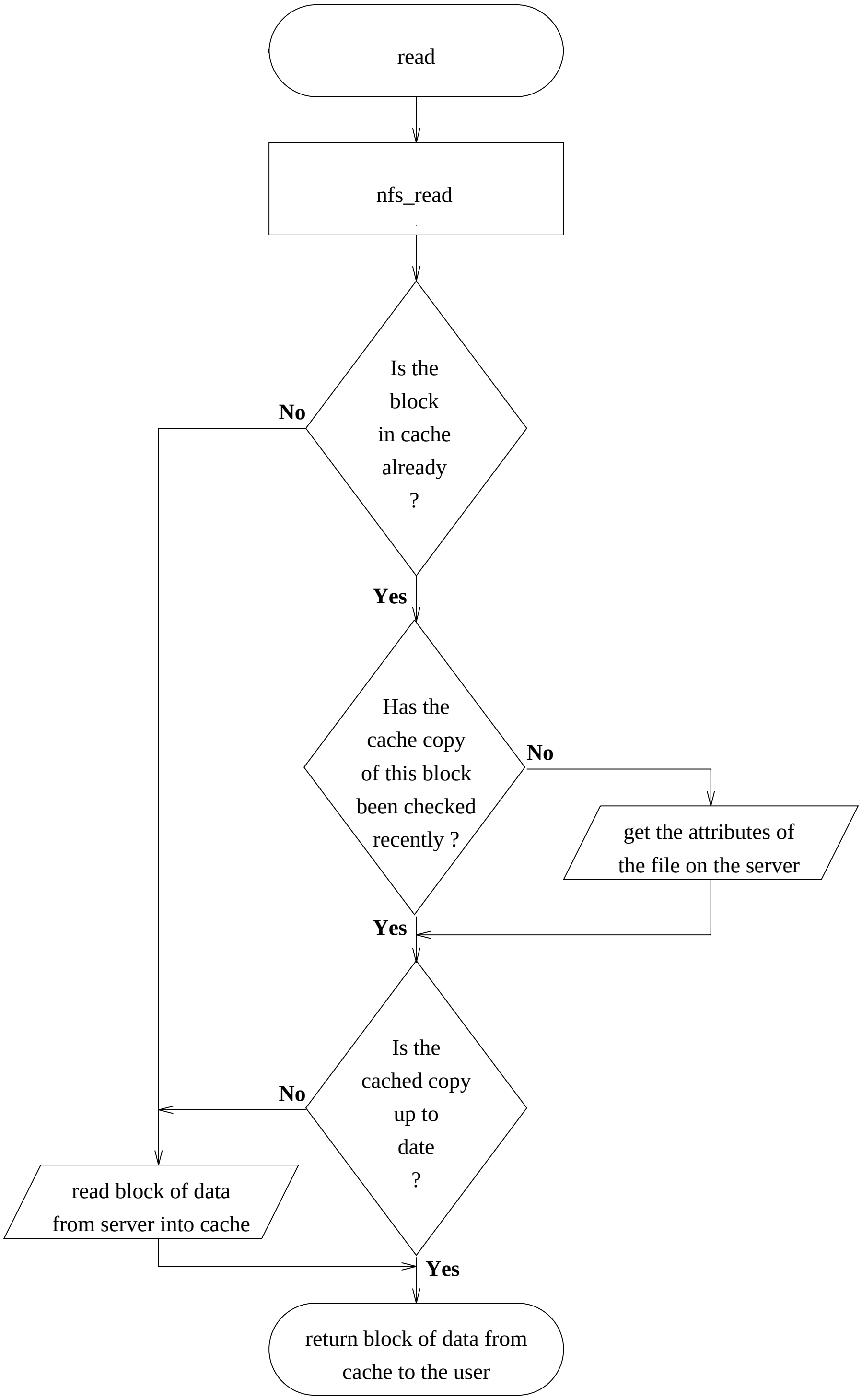

Figure 7: A chart of the flow of actions required in returning to the user a block of data from a file residing on a networked file system (NFS).

disk drive, has an advantage in distributed systems, apart from increasing the speed of clients, as programs that write data no longer do so synchronously with the disk. *Write-behind* also decreases the activities of the server and the communications traffic between server and client. However, the *write-behind* facility of the buffer cache has a disadvantage in a distributed system. Modifications made to a file by one client will not be visible to another client reading the file. In order to minimise the amount of time during which clients will access incorrect data, and also to minimise the chances of data loss due to write-behind, NFS uses a modified write-behind technique. In NFS, the closing of a file forces any unwritten data associated with that file to be synchronised with the server's disk.

The existence of the block cache does result in a difference in operations between the logical operations of users requesting data from the cache and the blocks themselves being written to and read from the file systems. An example of this is that two consecutive read operations on a particular file (for example, by a program executed two times in succession) may involve the complete reading of the data into the cache only once with that data being available for the second set of requests without additional disk access.

2.4 Operating system monitoring

This section will broadly cover the different techniques of system monitoring; the following chapter will cover particular methods and work derived from these methods in more detail.

System monitoring is increasingly important for the evaluation of operating systems. Examples of the uses of system monitoring include assisting in the assessment of an operating systems performance, or the validation of a particular sub-system, e.g. the correct implementation of an NFS server.

Modern operating systems have grown in size and complexity resulting in considerable difficulty building an understanding of the complete system. In the past, an operating system may have been understood by the study of the program code used;

in a modern operating system this code may be many hundreds of thousands of lines of code, and the code itself may not be readily available for proprietary and copyright reasons. As a result, system monitoring is often considered necessary to reveal aspects of the system and users' behavior, and to provide information on how to the improve performance of existing systems as well as assisting in the design of future systems.

There are four aspects that system monitoring must address in its design and implementation for it to be useful. These aspects, which may conflict with each other, are listed below in a summary adapted from Zhou et al. [117]:

- **Comprehensiveness:** a monitor system must provide enough correct information for a complete picture to be built up from the data captured. For example, the monitoring of file system should give information on all aspects of the usage of the file system and not only the operations resulting from one particular activity or user.

- **Flexibility:** ideally, a monitor system should be able to satisfy different monitoring needs. It should be able to trace the whole computer system, parts of the computer system, the actions of a particular user and of a particular program.

- **Minimal Impact:** the use of a monitoring system should involve minimal changes to the computer. In system monitoring it is most important that the system being monitored has the same behavior as when it is not being monitored. Minimising the changes required for monitoring has the additional advantage of reducing the chance of errors being introduced into the operation of the system being monitored.

- **Convenience of Analysis:** the output of the system monitoring, in its final form, should be of use to the researcher. Ideally the output of the analysis should require little or no post-collection processing. For example, complex cross-correlation of trace records should, if required at all, be only a single task required just once at the end of the trace.

System monitoring often takes two broad, interrelated forms: gathering information about the system itself, and gathering information about the behavior of the users of the particular system.

For some forms of system monitoring, such as kernel instrumentation or network monitoring where the machines are monitored *in situ*, the users can be considered to be load generators, using and causing the machines to behave in particular ways, the characteristics of which are being collected by the system monitor. This method can reveal the behavior of a system when in normal use. In this way the monitoring of operating systems involves extracting behavior of a system and commonly this information reflects the behavior of the users.

Several different techniques are used to monitor operating systems in general and, of particular interest in this study, the file system. In Figure 8 several system-monitoring techniques are illustrated. Shaded regions represent the area each system monitoring technique involves. A technique such as network monitoring has its access restricted to the data in the network, while kernel instrumentation could potentially involve any part of the operating systems kernel.

2.4.1 Benchmarks and load generators

Load generators refer to a type of program which exists to generate a load on a system. Commonly a load generator recreates the behavior of one or more users by duplicating the operations of the users. In this way a load generator could be built to operate at a high level, duplicating the exact commands a user has executed (e.g. `ls`, `make`, `cp`) or can operate at a lower level, simulating the logical operations. An example of this would be opening a file and reading or writing to it, as a simulation of using an editor.

Because an artificial load generator commonly seeks to recreate the behavior patterns of users, the building of such load generators makes use of the behavior of users that have been previously monitored. As an example of the use of previous studies, it was noted that the use of small files is common in UNIX [73, 8, 88] so a benchmark might be designed to replicate this characteristic by dealing with a large number of small files.

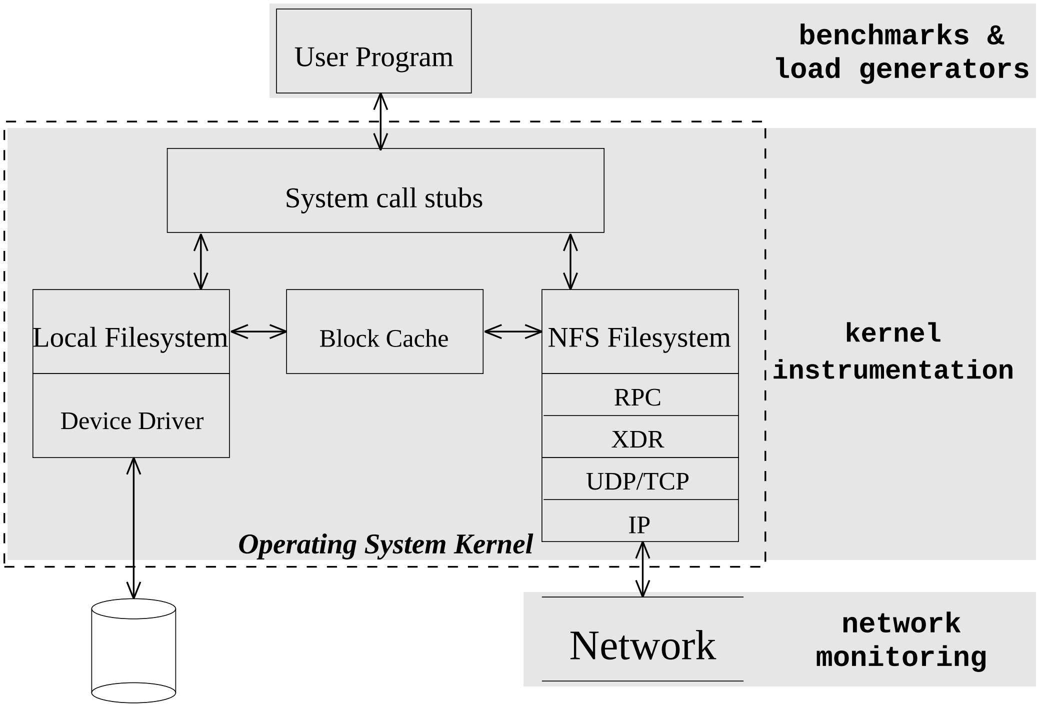

Figure 8: An illustration of several system monitoring techniques. Shaded areas represent the scope each technique can potentially access.

Load generators take a number of forms. For example, a load generator might be timed over its operation and the time taken can then be used to compare the same load generator running in different environments, thereby comparing the environments. This sort of load generator is usually termed a benchmarking program. Another example would be where the load generator tests several different types of operations. In this way, the load generator is exercising different alternatives, perhaps testing a wide set of operations on a file system.

The uses of benchmarks in particular and load generators in general include:

- testing the performance of a machine before and after modifications,
- testing a new implementation of an operating system attempting all operations to ensure they are functional,
- imposing an artificial load on a system, commonly a number of loads together are used to *stress*-test a system.

2.4.2 Log files

Log files are records of information collected for such activities as auditing and resource control. Log files can give potentially valuable information about the operation of the system and the activities of users. They provide the basis for useful studies even though commonly kept for other purposes and containing only the simplest of information.

Commonly used log files include:

System logs which commonly record information at a user level; which users were logged in and how long were they logged in are typical contents of user logs.

Program accounting logs which keep records on which programs are executed on a system, how much time was used by a particular program, how much memory was used, how many disk accesses were made, who ran the program, how long the program ran, etc.

Miscellaneous logs, such as the recording of old files as they are moved from secondary to tertiary storage.

Each log can help characterise the behavior of users and the particular usage a machine has had.

2.4.3 Snap-shots

Taking a *snap-shot* of the system is the process of making a static copy of the required information at a specified time. Such a snap-shot might be taken regularly, perhaps hourly, so that the differences between two (or more) successive snap-shots may be compared. As an alternative, a snap-shot may be taken only a single time. Such data could be used to give such things as average file size on the system at that moment.

A snap-shot requires little interference in the overall operation of the operating system. Typically, data can be collected for a snap-shot using tools built with user programs that require no alterations to the system, or, more-rarely, with special tools. However, such studies, particularly of the file system, are best done on a quiescent system so as not to be perturbed by changes (deletions, creations, file-moves and so on) during the collection of the data.

Snap-shots are most often made from the user level and, because of this, similar information can be collected, with the same user tools, from a wide variety of operating-system implementations.

Taking snap-shots as a form of ongoing system monitoring does have one major drawback; it cannot be performed continuously. It gives only a static picture of the system at any one time. The result is that using this technique will tend not to reveal short-term trends. This may not be a problem in studies of long-term trends but taking snap-shots is not considered an appropriate technique where there is need for information over the shorter term.

2.4.4 Network monitoring

With the common use of networks for distributed-computing environments, computer interconnectivity has become as important to a computer system as more-traditional components such as disk drives and CPUs. In distributed-computing environment the communications channel can be carrying disk traffic or inter-process communications traffic, monitoring the network between machines not only reveals information about the communications itself, but other activities of machines, for example, file system operations and the communication of processes. As a result of this, network monitoring has become a useful technique by which the activities of network-attached systems can be monitored.

Local area computer networks (LAN) are commonly Ethernet [61] or IEEE 802.3 [2][1] based but other network types are also in common usage [3, 4, 44].

The popularity of LANs has led to large numbers of people using computer networks as part of their daily work. The networks could be providing users with access to remote printing services or remote computer access, but particularly with UNIX systems, networks also commonly provide access to central file servers. In more advanced distributed systems, networks are the communications channel for the sharing

[1]Ethernet and IEEE 802.3 refer to two related but slightly different LAN standards.

of processor load as well as more elaborate network-based file systems.

In many local network systems, communications traffic is easily accessible. All information passed using the cable is available for anyone with the appropriate tools to intercept and monitor. On a segment of an Ethernet-based network (to which many computers may be connected) the network can carry only one packet from any computer at a time. The communication between the machines is forced into a linear time sequence. All data packets are consecutive and no network event can occur simultaneously with another.

Figure 9 shows a network monitoring system connected to a network used by other computers. If the network monitor can extract the communications traffic between other machines, the monitor can record this information and process it either then or later, by analysing information from the communications between the other systems.

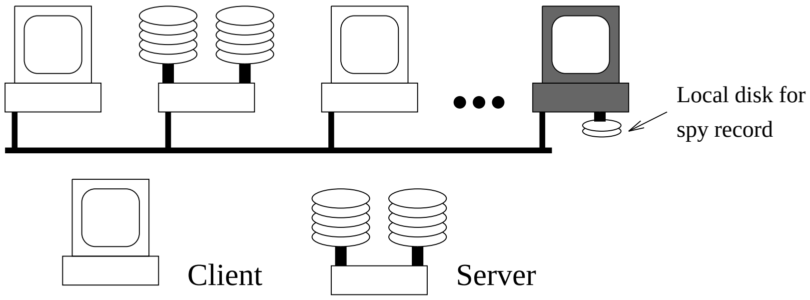

Figure 9: A network monitor system connected to a local area network can monitor and record the conversations between other machines that use the same network to communicate.

At one stage, such a network monitor would have been specially built for the task but modern workstations have been shown to be effective as network monitors [63]. They also often have the capacity to do rudimentary processing of the captured data in real-time.

This thesis discusses the use of network monitoring for gaining information about a distributed file system, although a network monitor could intercept communications of any sort passing through the appropriate network. The level at which a network monitor program can intercept communications between machines is shown in Figure 8. As shown in this figure, the network monitor will typically record the results to local disk, rather than generate extra communications traffic to be monitored, by sending the results to remote disk.

Network monitoring does place several requirements on the system it is monitoring. First, the information of interest (for example, communications traffic in a distributed file system) must pass via the network. This means that passive network monitoring would be of no use for the monitoring of file-system operations between a machine and disks attached directly to it because the operations between the machine and its disks would not pass through the network.

Secondly, network monitoring relies completely on the ability to derive the appropriate information from the communications traffic between machines on the network. For NFS, this is the communications traffic between clients and a file server. However, in the case of an NFS-style file system, not all of the actions that occur on a client will cause operations on the network. In particular, the use of a cache to increase performance in a file system will filter requests from a client to the server so a network-based monitor will not be able to record these requests. The use of processing heuristics based around the particular protocol used across the network is needed to make informed estimates on when the cache was used. This becomes necessary in situations when all that was sent across the network was a test to ensure that the contents of the cache were correct.

The network-monitoring technique has the particular advantage of being non-intrusive on the operation of other equipment (other systems being monitored). This includes both the recording of data from the network and allowing the recording of logs and so on to occur on a machine that is independent of the monitored systems.

An additional advantage of network monitoring is the simultaneous collection of information about potentially many different clients and servers. Any machine that uses the network may have data about it collected.

2.4.5 Kernel instrumentation

Kernel instrumentation consists of inserting extra code into parts of an operating system to collect and record statistics or more comprehensive data about the operation of the kernel.

Figure 8 shows the scope of kernel instrumentation. As the diagram indicates, kernel instrumentation could potentially record information about any operation of the operating-system kernel.

There are two commonly-used techniques for kernel instrumentation: a full instrumentation and use of kernel variables.

Full kernel instrumentation

For full kernel instrumentation, the instrumentation code is inserted into every part of the kernel that performs a function of interest. Such instrumentation is usually complex and involves adding code to many different parts of the kernel. An example of the variety of areas that may require modification is the reading of blocks from a file. Not only does the read system call cause blocks to be read from a file, but in some circumstances the write system call can cause this to occur also. Additionally, the virtual memory system on occasion needs to be able to read blocks from executable files. As this example demonstrates, even a seemingly easy example can be more complicated than it first appears.

Kernel code is limited by the fact that it is *in* the kernel, i.e. it must be crafted specially not to impact heavily on either the size or time-constraints placed on the kernel. It is also difficult to debug because every potential change to the system can, cause the system to crash, often without warning and sometimes without enough information to easily track the source of the error.

Kernel instrumentation also has a serious lack of portability. The instrumentation designed for one particular version of the operating system is not fitted easily to the kernel of another operating system. Indeed, it is often not even fitted easily to another version of the same operating system. Kernel instrumentation also presupposes access to the source-code for the kernel of a particular operating system. Such source-code is often difficult to obtain being restricted by high prices, non-disclosure agreements or simple lack of availability.

Owing to an effort by kernel-instrumentation implementors to minimise the impact of the monitoring system on the machine, the post-processing phase of a kernel trace can be both time consuming and complex. This process often involves the matching of records, for example, open and close records, to calculate the duration of file activity.

Finally, kernel instrumentation involves the careful management of the data generated by the trace mechanism itself. The data collection can, if poorly implemented, skew the results and change the operation of what is being monitored.

A full kernel-tracing system can suffer from the very characteristic that makes it so advantageous, that is, the sheer volume of generated data can quickly overwhelm local resources.

Despite these problems, kernel instrumentation has the potential to give an exact record of what occurred in the kernel of a system and, as a result, is commonly used when high precision is required.

Kernel variables

What has been termed kernel variables refers to the placing in the kernel of various variables. These variables record the activity of various parts of the kernel, for example, the number of times a page was found in the cache or the number of times a `read` system call occurred. These variables are often used in the development phase of the kernel and this extra instrumentation has simply never been removed. The most common method of using this form of information is to have a user process regularly obtain the values stored in the various variables and collect them in a file for later analysis.

Because such variables were placed in the kernel during its development, they might be of little use because they do not give the information that is needed. They might be incomplete (a read counter without a write counter), or little understood. Additionally, there is often no easy way to find out about their existence.

Access to the kernel source-code is required if kernel variables and variables are to be added or modified and, as has been mentioned above, kernel source-code is often unavailable. In such cases, this form of instrumentation relies totally on any pre-existing variables. Kupfer [50], in discussing various kernel instrumentation available in the Berkeley UNIX 4.2BSD kernel, comments on the ease with which various kernel values and the user programs that use them can become useless, incorrect or failure-prone, serving no useful purpose as a result.

Kernels instrumented in such a way do have a major advantage over other techniques, as the variables can be accessed easily by user programs.

Often the amount of data involved with kernel variables is small enough that the collection and storage is a modest or trivial problem. However, like kernel instrumentation, it is possible for the action of accessing and recording results to skew the values we wish to measure. As an example, if a program collecting statistics about the activity of the file system records its data to disk too often, it will quickly dominate the statistics it is collecting.

2.4.6 Specialist hardware

Specialist hardware is equipment designed specifically to collect information about a particular aspect of a machine, e.g. disk transfers; it could be attached to the same interface as the disk drive and all instructions to the disk drive would then be recorded by the specialist hardware. The data recorded by specialist hardware may well require an immense post-collection processing task although that does depend greatly on what is being monitored, how the specialist hardware performs this task and the nature of the final record the specialist hardware records.

Specialist hardware is, by its nature, extremely task-specific. For example, an analyser for a SCSI interface could not be adapted easily to any other task. Furthermore, the design and debugging stages of specialist hardware could be most-complex unless some type of modular, general-purpose equipment were used.

While having drawbacks that could be of great consequence, specialist hardware can be fast and accurate. It can record difficult-to-measure values at the circuit and interface level; measurements that might not be possible using another system monitoring technique. Additionally, good design of the monitoring hardware could make the system completely non-intrusive, introducing no changes at all to the system being monitored. A special-purpose network monitor could also be considered in this category.

Chapter 3

Related Research

This chapter examines the reasons for analysing the performance of operating-systems and how such needs directly motivate research into obtaining information on operating systems.

The first section discusses the types of data that system monitoring can make available to researchers. The second section describes the desire for analysing operating-systems with emphasis on how this research contributes towards the development and refinement of existing systems. The third section discusses previous research conducted using the techniques described in the preceding chapter. The final section covers briefly several publications for which the authors have made use of the raw data, results and conclusions of previous publications to assist in their own research.

The assessment of systems makes a significant contribution in the development of new systems, as well as an important contribution in the process of refining existing systems. In addition, system monitoring can usefully contribute to a system's effective day-to-day operation. In the development and redevelopment of systems a substantial number of studies have been performed, and while most studies are not directly comparable, the variety of studies has meant the development of a number of different monitoring methods.

3.1 Data from system-monitoring research

System-monitoring produces information that is either used and interpreted by the original researchers or made available for others to work with.

This data can be the characteristics of a group of users such as:

- the average and maximum number of users using a system in a given time,
- the average and maximum number of files read by a user in a given time,
- the average and maximum amount of data transferred by users in a given time, or
- the average type of access users perform on a file.

or the data can be the characteristics of the system such as:

- the number of processes over a given time,
- the average lifetime of a process, or
- file-system information such as the average length of a file.

The data are then used by researchers, showing trends in user and system behavior as well as identifying areas in which systems can be improved.

3.1.1 Types of data

There is a considerable variety in the types of data produced by researchers. Often the data are tailored to answer a small group of questions but sometimes, the data can be used by a number of different researchers. Data collected can include traces and the results from those traces (in the case of trace-driven analyses), statistical analysis of the user and system, or accurate timing information about accesses by software to various hardware systems.

3.1.2 The open-close session

The open-close session has been a central concept in research of file systems [73, 8, 108, 117, 116, 58]. An open-close session is the access by one user to a file through one particular program bounded by one set of open-close operations. If a program has a file open n times simultaneously, it is considered that there are n simultaneous open-close sessions. The open-close session bounds the read and write operations performed on a file for a user. Thus an open-close session has a duration and a session record will generally record the amount of data read from or written to a file.

The execution of a program can be considered to be bounded by the opening of the program's file at the beginning of execution and the closing of that file at the end of execution. In this way the execution of a program can be considered to cause an open-close session as well although, in this case, data are read from the file only during the course of the execution.

Figure 10 shows a variety of open-close sessions. Case 1 could be the reading of a demand-paged executable; initially data are read from the file and then, during the course of the open-close session, more data are read as those pages are needed. Case 2 could be an example of a configuration file that is read once only and closed when the program terminates. Case 3 could be an output file, first opened when the program starts up and then, when the data are produced, written to and closed at program completion. Case 4 could be a file being edited; it is first read by the editor and then, when the changes had been made, written to disk and the file closed.

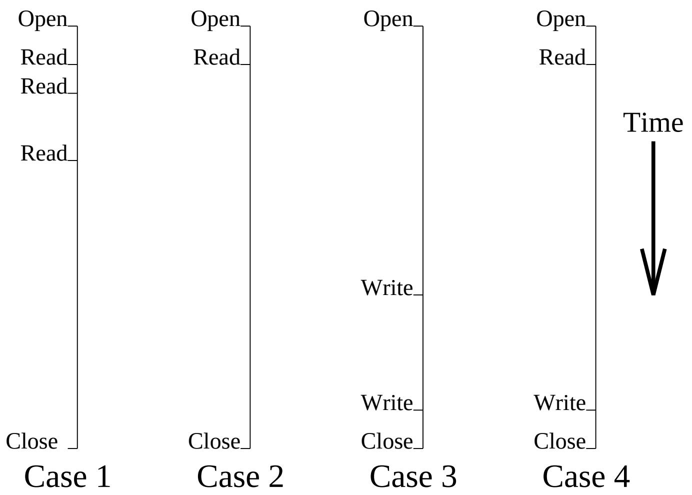

Figure 10: A variety of open-close sessions with read, write or read/write activity.

The characteristics of open-close sessions tell us a great deal about the files on a file system and the way users utilise those files. The average length of an accessed file has been used in research on the optimum block size in the file system [73, 60, 43].

The length of time a file is accessed, that is the duration of the open-close session, has been used in research related to the sizing of file caches and also to research on file-lifetime [73, 8, 108]. The total quantity of data transferred in an average open-close session is used in publications on cache characteristics in addition to simulations of caches [98, 73]. A related measure, the number of open-close sessions transferring a given amount of data, has also found use [58, 68, 24].

Open-close-session data can also reveal such information as the frequency-of-use of files, the characteristics of file access (read-only vs. write-only vs. read-write accesses) and the amount of data read from or written to files during the course of an open-close session.

3.2 Prior system monitoring research

3.2.1 Benchmarks and load generators

The running of a benchmark or load generator on a machine to assess the machine's performance in various situations is a commonly-used technique of system monitoring.

These programs need to be repeatable, and thus cannot rely on the actions of a single user. Instead, load generators and benchmark programs usually will attempt to simulate the average or peak usage of a machine. In the case of a benchmark, the time taken for the completion of a particular task helps in the assessment of the performance of the machine.

While popular, benchmarks have inherent problems that are not easily circumvented. The first major problem is that the typical user profile on one machine can differ from that on another machine. As a result, users of benchmarks must either design their own particular benchmarks for each system to be tested or use benchmarks that do not exactly match the researchers' requirements.

One way to overcome this problem partially is the development of standard benchmarks that test several different commonly-used aspects of the system. In this way an attempt can be made to exercise as many different aspects of operation as possible. Despite this, benchmarks often fail to test enough aspects of a machine's operation to be a totally reliable measure of performance.

Secondly, a single run of a benchmark does not provide reliable results. Random operations on a machine (tasks initiated by the operating system, other users, etc.), in addition to the unknown contents of the machine's caches, mean that the run time of a single test can vary considerably. This problem is overcome by running such benchmarks multiple times and averaging the various results.

The *Andrew Benchmark*, introduced in Howard et al. [42] tests five distinctly different parts of the operating system's operation. These five phases are:

- MakeDir - Construct a target subtree that is identical in structure to the source subtree.
- Copy - Copy every file from the source subtree to the target subtree.
- ScanDir - Recursively traverse the target subtree and examine the status of every file in it but do not access the contents of any file.
- ReadAll - Scan every byte of every file in the target subtree once.
- Make - Compile and link all the source-code program files in the target subtree.

This benchmark has been used by others [82] without any changes, although Ousterhout [71] produced an improved version of the Andrew Benchmark (the *Modified Andrew Benchmark* or *MAB*) which suits simultaneous testing in varied operating system environments. The major change is that, instead of compiling and linking code for the host machine, a C compiler is included that compiles for an experimental target machine called SPUR [41]. As a result, the same compiler is used on every machine tested. MAB has been used in the research of Macklem [56] and Ousterhout [71].

Endo et al. [34] note the problems of benchmarks, particularly in reference to file system testing. In Endo et al. problems of benchmarks are noted, including poor scalability and the benchmark failing adequately to measure the file system. They recommend a better benchmark, give outlines of the abilities such a benchmark should posses, and additionally detail two methods of achieving this end, although at this stage such a benchmark does not exist.

On a related issue, to address problems of benchmarks not being accurate simulations of the user workload, Ebling and Satyanarayanan [32] implemented a system that produces what they refer to as *micro-models*. A micro-model is a characterisation of a particular program. For example, the micro-model of a C compiler would be the reading of a .c file, the reading of several .h files and the writing of a .o file. Ebling and Satyanarayanan generated micro-models from short-term-trace data and, using these short-term traces, they were able to characterise various operations. Other micro-models were used to drive load generators which, as a result of the micro-models, gave a better approximation of the actual load. Such techniques seem certain to figure prominently in the future of benchmarks.

The *microscopic*-benchmarks of McGregor [59] were specifically intended to benchmark a single operation, such as a single `read` or `write` system call. The elapsed time for a single operation can then be used as a parameter for the queueing system throughout which that operation must pass.

Thekkath et al. [107] make use of a load generator to compare a new method of simulation. The use of a load generator means consistent testing of the simulated model can be compared with the original system on which the model is based. In this way, benchmarks are used to validate aspects of the simulator's usefulness and accuracy.

The creation and use of benchmarks is a popular technique for the comparison of systems and changes to systems. The results of Howard et al., Ousterhout and Ebling and Satyanarayanan among others [93, 113] lead to the introduction of new benchmarks and new methods of creating benchmarks. Researchers have also used benchmarks to compare the performance of different types of systems [69, 115, 31].

Benchmarks can also be used to show the change in performance on a system that result from changes in policy or procedure, that is, the method or procedure the system follows. Mogul [64] makes and measures the results of changes to the update policy of UNIX which affects when dirty disk blocks are written from the cache to disk. Cao et al. [19] use benchmarks to measure performance changes when the policy for file caching is altered. Baker and Sullivan [7] also used benchmarks to measure the performance of a system. In this case the measures were used to ensure that modifications to the system to improve other aspects of its operation did not degrade the systems performance too much. Workload generators have been used to load a system with *user-like* jobs or with tasks that stress-test the system [91, 46]. Macklem [56] used combinations of benchmarks and load generators to give information to assist in improving the NFS protocol. Additionally, benchmarks are common when comparing major changes in a system. The implementation of a new type of file system [84, 83, 82, 26] or a new type of object naming system [22].

3.2.2 System logs

System logs can give usable information about the operation of the system and activities of users. In particular system logs can provide the basis for useful research even though these logs are kept usually for other purposes (typically auditing).

Jensen and Reed [45] and Miller and Katz [62] investigate trends in file-migration patterns by analysing logs of files moving from one level of storage to another. These files are being typically moved into and out of a tertiary, mass-storage system (MSS).

Smith [97, 96] studied the movement of data sets associated with a particular editor to develop an algorithm for automatically moving data into and out of MSS.

The use of particular system logs, as with check-pointing, most commonly involves the analysis of long-term trends only. This was the case in the three cases cited above.

The use of system logs is usually highly restrictive and dependent solely on the information logged. A researcher may not have any control over the contents of the log. This situation is common where the log is designed for a particular purpose and the researcher has no desire to change the log, or perhaps because of access rights is unable to change the contents of the log.

However, there is one redeeming benefit: the logs themselves were generally not created for the research but for other reasons such as auditing, billing, cross-checking and error control. As a result, the researcher does not need to add significant workload to the system or have any significant overhead having the information contained in the logs made available.

3.2.3 Snap-shots

The ease with which benchmarks and system-log analysis can be done is comparable only to the ease with which system snap-shots can be taken.

Irlam [43] collected information from a once-only snap-shot operation of over 1,000 file systems. This collection was made by the running of several user programs with the results being collected and sent on for collation. These programs were able to be run on any UNIX-based operating system without any special access to operating system code. Irlam was able to obtain data easily from numerous sites in many, varied environments; the only common requirement was that they run an operating system with the appropriate user programs.

Snap-shots are suitable for use in long term research and, subsequently, the study of file migration. Both Smith [96] and Strange [101] have used snap-shots in such a study. A further study by Smith [98] uses a snap-shot with file-size and file-reference data to provide information on driving a cache simulator.

A system snap-shot makes an excellent supplement to other techniques. Full kernel instrumentation in association with a full snap-shot of the file system at the beginning and end of the trace period can, with appropriate post-processing, provide an accurate record of events on the system with knowledge of the state of the machine before and after the trace. Such a combination can mean the trace monitoring output does not need to be as comprehensive, reducing the size and potential impact of the full kernel instrumentation. Floyd [36, 37] used a combination of trace-driven analysis and snap-shots for a study on short-term file reference patterns.

Snap-shots do have one major drawback in that they can not be performed continuously. They give only a static picture of the file system at any one time. The result is that research such as that presented in Satyanarayanan [88] has filtered out all file trends that take place in the period between the times when the snap-shots of the file system are taken. This may not be a problem in research of longer-term trends but the use of snap-shots is not an appropriate technique *per se* in research where there is a need for shorter-term information.

3.2.4 Network monitoring

With the increasingly common use of networks for interconnecting computer systems, network monitoring, a relatively new technique, has the potential to be a commonly used monitoring method. Network monitoring takes two forms: firstly just the monitoring of the traffic of a communications network, useful for interpreting the make-up of communications-traffic in a particular network and secondly, interpreting the traffic of a communications network, useful in determining the operation of machines connected to a communications network.

While the interpreting of communications traffic is still a technique in relative infancy, there have been several research topics, particularly monitoring studies, performed already. Investigations into the overall capacity of networks investigated, such as that presented in Boggs et al. [17], is important when the efficiency of the computer network can play a significant role in the performance of an attached machine. Additionally, the work of Gusella [39] into the usage of these networks can assist in the planning for future networks. Gusella recorded the header information of every Ethernet network transaction (packet) over a given time period and later processed that data off-line. In this way, a partial trace of the operations on the network was made.

In 1990, Mogul [63] argued that workstations could make efficient monitors of local networks. They had the required combination of CPU power, memory size and network-interface-speed to enable them to efficiently collect and process network data in real time. Several years ago the monitoring of computer networks required complicated, expensive, specially-built equipment. Of course, such equipment still has a place in the analysis of the increasingly-faster computer networks becoming available.

Two facilities available on many common computers are Sun's Network Interface Tap (`NIT`) [103] and Digital's `packetfilter` [65, 27]. These facilities allow user programs to access and record data passing through the network directly. Before the availability of such facilities, packages such as that written by Barnett and Molloy [10] had to be written specifically for the Ethernet interface a particular machine possessed.

With the increased capacity of workstations, researchers now need not limit themselves to capturing all (or part) of the raw data that traverses the network and interpreting it at a later stage. Because of the capacity of networks (Ethernet has a raw capacity of 10Mb/s), the sheer quantity of data does not easily allow for a raw trace of the network contents (difficult to store). Programs such as `rpcspy` [11, 12], built on the Sun and Digital network monitoring facilities mentioned above, do some processing and interpretation of incoming data to reduce the quantity of information ultimately recorded.

Blaze, who constructed the `rpcspy` system of reference [11, 12], has used it to good effect for the monitoring of a network based around a large file server. His research results have also been used in several other publications [39, 10, 15, 16, 13]. Dahlin et al. [24] used the `rpcspy` tools to characterise file-system load in a distributed system. The results of this research have then been used to justify the building of a new style of distributed file system, xFS [120]. Anderson [1] used `rpcspy` to analyse the distribution of traffic across different file systems and to theorise on better use of local disks in a networked file system.

The non-intrusive nature of network monitoring and the ability to easily analyse the activity of a whole network simultaneously will mean an increase in the quantity and variety of research using this technique.

3.2.5 Kernel instrumentation

Much system-monitoring research has used kernel instrumentation but there is a broad delineation between those performing full instrumentation and those using kernel variables available at the user level.

Full kernel instrumentation

The accurate and comprehensive nature of kernel instrumentation makes it a popular choice when kernel instrumentation is possible. The accuracy of kernel tracing is an important benefit and the ability to get fine resolution on the timing of system calls was used in Griffioen and Appleton [38]. Zhou et al. [117] detail a full kernel instrumentation package called `snooper`, which is designed to be a low-overhead, system-call-only recording package. This paper also details interesting findings including information on data-transfer rates to and from file systems, the durations of various operations and process-lifetimes in the traced system.

A common use for full kernel instrumentation is to generate data that can then be used in trace-driven analysis of the operating system and file system. Ousterhout et al. [73] use full kernel tracing to give characteristics of the file system as well as data to drive a cache simulator. Smith [98] makes use of this method to drive file cache simulations. Bozman et al. [18] used kernel instrumentation to generate data used to characterise file reference behavior and drive a simulation of file reference behavior.

Several publications with instrumented kernels have been done on distributed systems. Several of these were done with the Sprite distributed system [72]. Baker et al. [8] presented not only the characteristics of the distributed file systems (and how its characteristics had changed when compared with a previous paper [73]), but also used the trace of the study to investigate how effective the caches were in a distributed file system. Welch [116] further performed a related study on the same distributed file system to analyse the effectiveness of cache consistency models in use. Additionally, in the same distributed file system, Welch [114] used a tracing system to look at the impact of changes in this system including the use of the number of client-server transactions per second as one of his comparison metrics. Makaroff and Eager [58] use kernel instrumentation to record physical-block information to show differences between systems performing different tasks such as the clients and servers of a distributed file system. Others have used kernel instrumentation to gain specific information about a machine. Ruemmler and Wilkes [86] were interested particularly in the disk's active data set. Li [53] instrumented MS-DOS machines to gain information about augmenting cache behavior in those systems.

Mummert and Satyanarayanan [67, 99] detail a distributed-system kernel tracing facility. In this facility, machines will forward their tracing results onto a central machine for storage. In this way some aspects of kernel instrumentation, such as the need to write large local trace files, are modified. This system exchanges the workload which would be incurred in the local recording of large trace-logs for the overhead of sending the data through the network to the logging host. This facility uses agents in the clients of a distributed system to periodically send trace records to a collection unit. Such a system has the effect of passing the problems of trace-data volume to the collector, a machine that would not necessarily be among those being traced. Kistler and Satyanarayanan [48] and Kistler [49] have used this technique to aid research into a new file system design.

Kernel instrumentation has been combined with other techniques to provide additional information. Floyd and Ellis [37] combined kernel instrumentation with the use of benchmarks in a study on file reference patterns. Floyd [36] combined this technique with the use of snap-shots to study shorter-term file references. In 1989, Cheriton and Mann [22] used kernel instrumentation in combination with benchmarks in a study on an improved naming service for distributed systems. Endo et al. [34] discusses a method of kernel tracing involving intercepting the communications between kernel components. For example, the tracing of the cache buffer would involve the interception of all information exchanged by this kernel component with the rest of the system. Such a method depends upon a highly modular kernel system, however, Endo et al. present this method as a facility in an already highly modular kernel design.

Kernel instrumentation is both popular and effective if its drawbacks can be overcome.

Kernel variables

The use of kernel variables often provides easy access to information in the kernel from the user level. UNIX system programs such as `ps` use such variables for the information they generate. In addition, the remote collection of kernel values is possible. A simple example of this is the `rwho` daemon service [28]. This daemon periodically broadcasts onto the network a packet of data containing information about the status of the machine and the users on that machine. While the `rwho` daemon is a simple example, Kupfer [51] shows the use of remote instrumentation for the collection of comprehensive data from kernel variables.

Spasojevic and Satyanarayanan [67, 99] mention the collection of an elaborate activity summary when a main trace system fails and then, at a suitable time, sending this summary to a collection agent. In this way their distributed data collection does not fail to collect any results even during a time when clients are unable to communicate with the collection agent. In 1988, Bach and Gomes [5] used kernel variables to show, among other things, that an operating system spends the most of its time dealing with file-system operations. Macklem [57] makes use of kernel variables to assess the performance of a new implementation of NFS. This is a good example of variables which have been placed into the kernel during development remaining available to anyone using derivatives of the implementation (unless, of course, the variables are removed deliberately).

Owing to the ease with which kernel-variable data can be collected, such information has been used in a variety of research, either as a primary or a secondary mechanism for supplying information.

3.2.6 Specialist hardware

Specialist hardware is often used during the development of other hardware. Emer and Clark [33] and Clark et al. [23] used specialist monitoring hardware to characterise the performance of particular CPUs. Such techniques would have common usage during the development cycle of such hardware but, because of the specific purpose for which such hardware must be built, wider applications are often not possible. Shand [92], however, used general-purpose equipment to design a hardware monitor. The result was that his hardware monitor could be adapted to monitor not just the system it was designed for but also other systems of similar architecture. Shand used hardware instrumentation to study the operation of a machine running UNIX and was able to give accurate, short-duration timing of events as well information on other aspects of the operating system such as task preemption and Direct Memory Access (DMA) handling.

The use of specialist hardware is uncommon for systems monitoring because of the investment that would have to be made in time and effort to get such a system operational. However, with the application of general-purpose equipment such as that in Shand's study, more reports done using this technique will become available.

3.3 Research using system monitoring results

In addition to publications that have incorporated their own system monitoring, a considerable number of researchers have used the results of others, commonly the traces of a full kernel instrumentation analysis, for use in their own work.

3.3.1 Trace-driven research

Considerable research has used the trace data collected by other authors for use in their own work. This is a major benefit of trace analysis. Once the original data set has been collected, dependent on its coverage, the trace may potentially be reused in other work.

The data used in Ousterhout et al. [73] is re-used in Thompson [108] in a follow-on study of the effects of file deletion.

The data from Baker et al. [8] has been used by Shirriff and Ousterhout [94] for a name and attribute caching study. The Baker data was again re-used in Baker et al. [6] and Baker [9] for a proof-of-concept study involving changes to hardware support for caches.

Blaze and Alonso [16, 15, 14] and Blaze [13, 11] all make use of common sets of data gained with the `rpcspy` tool [12].

While details are not given on exactly what traces are used, Ebling and Satyanarayanan [32] used traces to develop their micro-models for incorporation into high-accuracy load generators.

3.3.2 Characteristics and conclusions

In the tradition of citation, many papers also use information from previous papers to assist in the justification (or refutation) of ideas. As a result, research into operating systems can quickly gain momentum through the structured use of a results from several sources.

Baker et al. [8] uses results from Ousterhout et al. [73] for comparison as a follow on study covering similar ground. In this way Baker et al. have been able to show interesting trends in growth of systems between the two publications by comparing two related sets of results. Conclusions and the system characterisation documented in Ousterhout et al. [73] is used in Davies and Nicol [25], Floyd and Ellis [37] and Reddy and Banerjee [80]. This further work uses such values as the basis for workload, prospective, measured and simulated respectively. Ramakrishnan and Emer [79] use Zhou et al. [117] among others for examples of characteristics from which mathematical simulations are then built. Carson and Setia [20, 21] use the results of Ousterhout et al., Baker et al. and Smith [98], among others, to define and refine models they are developing. Thompson [110] uses the results of Ousterhout et al. and Smith [98, 96], among others, in an analysis of cache designs and a more general study of caching. Thompson also uses the work of Zhou et al. and results from his own previous studies [109, 108] in this work. Hartman and Ousterhout [40] use the conclusions of Ousterhout et al. and Baker et al. to justify the need for a new system and to assist in defining the characteristics the new system should possess.

As can be seen by this small snap-shot of workload, system-monitoring research has been extremely important both for the raw data it generates and the results and conclusions that are drawn from that data.

Chapter 4

Kernel Instrumentation

This chapter describes the kernel instrumentation performed in this study.

4.1 Objectives

As discussed in Chapter 3, a system-monitoring procedure can be required to provide information about the computer and operating system being monitored, in addition to the activities of the users of that computer. This information, as it relates to studies of the file-system, includes:

- the average or maximum amount of data an average user will require in a given time,
- the average or maximum time a file is used, or
- the average amount of data in a file.

Full kernel instrumentation gives us the ability to accurately and comprehensively record this information, directly from the kernel, as the events occur.

4.2 The design of full kernel instrumentation

As discussed in Section 2.4.5 there are a number of methods of kernel instrumentation. One of these methods, full kernel instrumentation, through the insertion of instrumentation code into an operating system, has the potential to generate comprehensive traces of information about a monitored system and its users.

However, full kernel instrumentation, as previously mentioned, does involve a sizable complexity in programming. A kernel for Ultrix 4.3a is made from approximately 1300 files, and has a total of approximately 720,000 lines of code. Implementing a full kernel instrumentation system, `snooper`, required changes or additions to 48 files involving approximately 1,600 lines and the addition to four new files which results in 2,750 extra lines of kernel code.

An additional complexity relates to the use of a kernel instrumentation system. `Snooper` is based on a set of modifications to the Ultrix 4.3a operating system. To make these modifications, the source-code is needed. To obtain operating system source-code it is necessary to obtain a licence from the vendor (Digital in this case) and also a licence from AT&T. The second licence is necessary because most commercial implementations of UNIX, such as Ultrix, are built around the original AT&T implementation, and incorporate some of its code. As a result source-code for the two operating systems must be sought, and because these are not common, their purchase can be time consuming and in some circumstances, expensive.

4.3 A kernel instrumentation implementation

The kernel instrumentation this thesis uses is the `snooper` package. The package was originally implemented by Siebenmann and Zhou [95] for Ultrix version 3.3. `Snooper` is a set of kernel instrumentation routines for the recording of information about logical file operations, physical-block operations, process execution and termination, etc. The `snooper` package also instruments parts of the virtual memory system, however as this does not have direct relevance to this thesis, it will not be discussed further here. The `snooper` package is based upon the package of the same name described in Zhou et al. [117] which, in turn, has its ancestry in the package used by Ousterhout et al. [73] to perform their study of the UNIX 4.2BSD file system.

It was required that the `snooper` package operate with Ultrix 4.3a, the latest version at the time. Due to differences in the various versions of the operating systems, the code can not be copied simply from one kernel system to another or from the Ultrix 3.3

implementation to Ultrix 4.3a. These code changes involved accommodating differences in the flow of the kernel code itself. Code changes were needed to be able to handle new data structures and accommodate changes in the usage of older data structures.

To ensure the correct operation of the new kernel, a suite of programs was written that would comprehensively test the modified kernel code.

The implementation of `snooper` under Ultrix 4.3a consists of four parts:

1. the mechanism for activating and deactivating the trace system, and the mechanism for changing tracing files during the course of a trace,
2. instrumentation of the various kernel components,
3. trace buffer management and synchronisation, and
4. off-line processing software.

The implementation of these parts in `snooper` is described in the following sections.

4.3.1 Trace system control

Modifications to the `snooper` system involved building two extra system calls to allow user-level control software to have access to the `snooper` code. The first of these system calls, `strace`, allows a user-program to activate the trace system. The arguments for `strace` include the name of the file to trace to and an indication of which kernel systems should be traced. By passing the `strace` system-call a NULL in place of the trace file, the trace system is signalled to shutdown.

The second of the system calls, `straceserver`, does not take any arguments or perform any direct function for the user program. It is a control point over the trace facility for emptying the trace buffers into the trace file. Typically, a part of the monitoring program will become a `daemon` program. This `daemon` program will make an `straceserver` call which will not complete unless the tracing system is deactivated. The daemon program can then be stopped, restarted or have its priority changed, just like any other process, thereby controlling directly the overhead introduced into the system by the transfer of trace buffers into the trace file. Such access to certain kernel functions is quite common and is used to control programs such as the NFS server programs [102].

In `snooper`, the trace buffers are managed as a ring. A routine (`getroom`) allocates and fills successive buffers in a buffer pool. As each buffer is filled, a flag is set as an indicator that the buffer should be flushed. If buffer space cannot be allocated, the trace record is dropped (and a counter is incremented tracking the number of dropped records). The `straceserver` call will look for and flush successive full buffers to disk continually and reset the appropriate flag when a buffer is free again.

The `getroom` routine uses buffers but cannot wait for `straceserver`. This is because `straceserver` may depend on kernel activities that are being traced and `getroom`, being critical in the trace code, may be called from one of those traced operations. An example is where `getroom` is being called to trace a record due to a buffer flush. The `straceserver` code can also cause a buffer to be flushed. As a result, this system as implemented is deadlock-free, but `getroom` will discard all trace records if `straceserver` should stop operating.

A mutual-exclusion problem was recognised in the porting of `snooper` to Ultrix 4.3a. This problem is related to the point in the trace system where the trace system output is changed from one file to another. The kernel tracing system may be required to change trace files at any time. When a particular trace file is growing too large, the output can be switched to a new file so that the old file can be dumped onto tape to free disk-space. However, without appropriate programming, there is a possibility that the `straceserver` process, unaware of the file changeover, can attempt to write data to the old file using an invalid or uninitialised file-reference. This situation exists partly because there are two different processes at work; one changing the output file and one dumping the current file. The UNIX system can cause the process changing the output file for the trace data to be suspended in the middle of its activity and signal the record-dumping process to continue. The result is `straceserver` (the record dumper) will attempt to use a file reference that is in an indeterminate state which will, typically, cause the operating system to crash.

A busy-wait lock is not acceptable in the kernel because, at this level, it would place an unnecessary overhead on the kernel and possibly change the system's behavior. Instead, to give a mutual-exclusion lock that was not overly-consuming of resources a Peterson's-solution-style lock [75] was implemented as part of the port of `snooper` to Ultrix 4.3a. The result of using this lock is that it is possible that the `straceserver` process may spend more of its time temporarily suspended and, thus, unable to dump the contents of buffers to the trace file. However, this risk of trace-data loss occurs only at trace-file changeover.

4.3.2 In-line instrumentation

Full kernel instrumentation involves the coding of *trace points* into the kernel. At these trace points, data are recorded about what is occurring. In Figure 11 we can see that kernel instrumentation can (potentially) capture information about every operation performed by user programs.

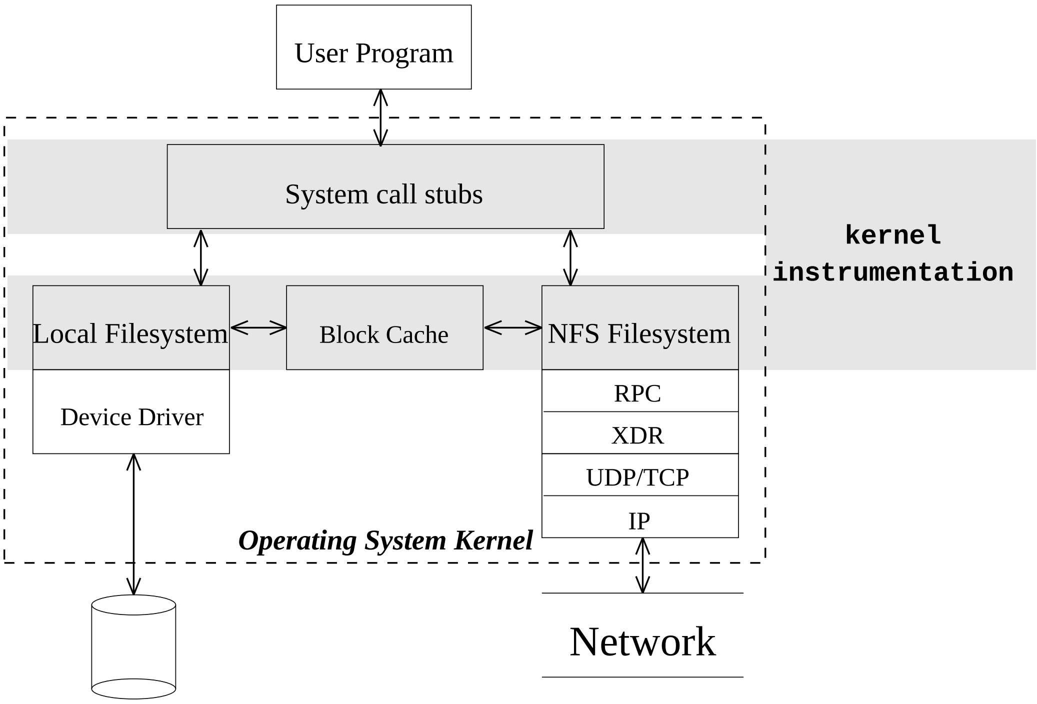

Figure 11: All levels of an operating system, showing the point at which kernel instrumentation information is extracted.

An example of instrumented system calls is the `seek` operation, which changes the point at which data are read from or written to a file. The following code fragment is from `lseek`, showing the `snooper` instrumentation in that system call. The additional code required for the tracing system is surrounded by `#if define(STRACE)` and `#endif  STRACE`. This tracing code will cause the `seek_trec` function to be activated each time a program successfully uses the `lseek` function call. The tracing function `seek_trec` could then record information about the `seek` operation.

```
....
        if ((where < 0) && ((vp->v_mode & VFMT) == VFREG)) {
                u.u_error = EINVAL;
                return;
        }
        ret = VSEEK(vp, where); /* actual Seek command */
        if (u.u_error) {
                u.u_error = EINVAL;
                where = -1;
         } else {
                if (ret) {
                        u.u_error = EOPNOTSUPP;
                        return;
                }
#if defined(STRACE)
        if ((trace_flags & TRACE_SYSCALL) && strace_vnode(vp))
            seek_trec(vp, (u_long) fp->f_offset, (u_long) where,
                  uap->sbase, start_time);
#endif  STRACE
                smp_lock(&fp->f_lk, LK_RETRY);
                fp->f_offset = where;
                smp_unlock(&fp->f_lk);
        }
....
```

A decision must be made when setting up an implementation of kernel instrumentation as to what is to be traced. In the `snooper` system only regular files are traced for logical operations. For the block-tracing component, `snooper` will trace all operations that involve the use of blocks.

4.3.3 Activities `snooper` traces

The system calls which `snooper` will trace include `creat`, `truncate`, `open`, `close`, `read`, `write` and `seek`. All of these operations are at the logical level and are carried out only on regular files. Additionally, `snooper` instruments the `delete` and `rename` system calls to allow the names of existing files to be traced as they are changed. The system calls `mkdir`, `rmdir`, `chdir`, `mount` and `umount` are traced to allow information on changes in the directory structure to be recorded. Completing the list of system-call operations which `snooper` traces are `exec`, `fork` and `exit`. These records can then be used to give information about which programs caused which particular `open` and `close` calls as well as chasing parent/child-process chains. It is also possible to reconstruct open-close sessions where the `open`ing of a file is in a parent process and the `close` operation occurs in a child process.

All `read` and `write` operations involving blocks are traced by `snooper`. Potentially these could include operations involving tape units or raw disk devices but, in practice, block activities are related exclusively to file data on a diskless client.

4.3.4 Additional information created by `snooper`

In order to give a form of unique file identification, `snooper` generates and assigns unique file-identification numbers (file IDs). The unique identification of files is important for the various operations on a particular file to be linked together and for any meaningful analysis to be conducted. An ideal approach to file identification is to use the pathname of the file. However, the pathname of a file is difficult or impossible to obtain during a trace without causing substantial overhead on the traced system. Additionally, a file can potentially be deleted and the file name reused.

Another alternative is to use the file's i-node number. However, while the i-node number is unique for a particular file system it is not unique across file systems. Furthermore, an i-node is potentially reused when a file is deleted and recreated.

The first time a file is processed by a file routine (`read`, `write`, `open`, `close`, and so on) a flag in the i-node is tested and set if it is not set already. Additionally, the file is given a file ID which is recorded in the i-node allowing it be identified by `snooper` the next time it is seen. The file IDs are consecutively-assigned integers and are used internally by `snooper` as the names of files.

Each `snooper` transaction is also identified by the process ID (PID) that caused it. This means that each of two different processes holding open the same file can be uniquely identified.

As a result of these two different forms of identification, every set of logical operations on a file by a process, (starting with an open and ending with a close) can be separated.

File IDs are associated only with logical operations such as read, open, write and not with block operations.

`Snooper` cannot easily identify when a process has the same file open a number of times simultaneously. In the off-line processing phase all the operations of the simultaneous open-close sessions are considered to be part of one open-close session. More details on how open-close sessions are determined is given in Section 4.3.7.

4.3.5 Data generated by `Snooper`

`Snooper` collects a copious amount of data from the various file system operations. Table 2 shows the fields for each record type. The trace file is binary to ensure compact data and quick transfer of records from each trace stub to the trace buffer.

Record Type	Data fields
open/creat	record type, open mode, reference count, pathname statistics, real time, CPU time, last-modify time, last-access time, g-node number, g-node generation number, file ID, file size at open, parents g-node number, parents device number, device number, process ID (PID), User ID (UID), file type, duration of operation, filename
close	record type, reference count, real time, CPU time, g-node number, g-node generation number, device number, file ID, file size at close, process ID (PID), User ID (UID), file type, duration of operation
read/write	record type, reference count, duration of operation, real time, CPU time, file ID, offset into file, bytes read/written, process ID (PID), User ID (UID), file type
seek	record type, seek base, duration of operation, real time, CPU time, file ID, old offset into file, new offset into file, process ID (PID), User ID (UID)
ftrunc/trunc	record type, duration of operation, real time, CPU time, g-node number, g-node generation number, device number, process ID (PID), User ID (UID)
delete	record type, pathname statistics, duration of operation, real time, CPU time, last modify time, last access time, g-node number, g-node generation number, file size at deletion, device number, parents g-node number, parents device number, process ID (PID), User ID (UID), filename

continued on next page

Record Type	Data fields
rename	record type, duration of operation, real time, CPU time, last modify time, last access time, g-node number, g-node generation number, device number, old parent g-node number, old parent device number, new parent g-node number, new parent device number, old filename, new filename, process ID (PID), User ID (UID)
exec	record type, pathname statistics, duration of operation, real time, CPU time, g-node number, g-node generation number, device number, file size, parent g-node number, parent device number, text size, data size, stack size, filename (last component), process ID (PID), User ID (UID)
vfork/fork	record type, child process ID, duration of operation, real time, g-node number, g-node generation number, device number, text size, data size, stack size, process ID (PID), User ID (UID)
exit	record type, duration of operation, real time, process ID (PID), User ID (UID), text size, data size, stack size, CPU time used in user mode, shared text size, shared memory size, unshared data size, unshared stack size, page reclaims, page faults, swaps, block input operations, block output operations, messages sent, messages received, signals received, voluntary context switches, involuntary context switches,
rmdir/mkdir	record type, pathname statistics, duration of the operation, real time, CPU time, g-node, g-node generation number, device number, directory size, parent g-node number, parent device number, process ID (PID), User ID (UID), filename
	continued on next page

Record Type	Data fields
chdir	record type, pathname statistics, duration of the operation, real time, CPU time, new directory g-node number, new directory g-node generation number, new directory's device number, old directory g-node number, old directory number g-node generation number, old directory device number, parent g-node number, parent device number, process ID (PID), User ID (UID), new directory's filename
mount/umount	record type, duration of the operation, real time, CPU time, process ID (PID), User ID (UID), mount flags, max transfer size, optimal transfer size, block size, file system type, total number of i-nodes, total number of free i-nodes, total number of 1Kbyte blocks, total number of free 1Kbyte blocks, total number of user consumable 1Kbyte blocks, minimum size in bytes before paging, major/minor devices, root mapping from exports, file system's root file path name, device's path name
block read/write	record type of block read/write, why the block read/write happened, cache hit, device block's file is on, size of I/O request, duration of the operation, real time, process ID (PID), g-node to which this block belongs, number of the block

Table 2: `Snooper` trace record types and data fields

4.3.6 Program execution

When a program is executed, the `exec` system-call will cause the program header to be read from the beginning of the executable file. The header of an executable file contains additional information about the size and position of the executable code, data and stack segments as well as checksums to ensure the integrity of the file. The `exec` system-call will then transfer control for that process to the executed program based on the contents of the header.

Some executable programs start with a special number which is used to indicate the type of the program. If no such special number is present, as is the case with most script files, the `exec` system-call will cause `/bin/sh` command interpreter or another program (if given) to be executed.

There are two different ways to treat executable programs:

Pure: pure executables are loaded into memory completely by the `exec` call. These programs are typically small, and it is expected that all parts of the program file could be referenced during execution. In most systems, pure executables are rare, most executables being of the demand-paged type.

Demand-paged: demand-paged executables are not fully loaded immediately into memory. Only a small amount of the program is loaded, the `exec` call will then transfer control to the program components loaded and will force the loading of pages of information from the program file as the information in the pages is needed. This method minimises the amount of memory needed to load a particular program.

The early implementations of NFS introduced a modification to the way some executable files were loaded to improve NFS performance [87]. Demand-paged executables did not benefit from the file system performing read-ahead because pages of the executable are frequently accessed from the file in a non-sequential fashion. The solution implemented was to cause small, demand-paged programs to be treated as if they were pure executables. This meant that all pages for the program were loaded into memory from disk at the start instead of being demand-paged during the course of execution. The result was improved performance because of the ability to take better advantage of read-ahead and the elimination of the demand-paging overheads for that particular class of executable. An additional improvement was the use of fill-on-demand clustering to group small page-in requests resulting from demand paging into one large one. Fill-on-demand clustering is a method used by the memory manager for the pre-paging of data. When a page fault occurs, the page-fault handler attempts to read in the desired page along with adjacent pages on either side of the page for which the fault occurred. These changes have now been incorporated into most modern UNIX operating system versions.

The loading of pages by a demand-paged executable is performed by the virtual memory manager, so there is no logical read operation associated with the loading. Instead, virtual-memory page-faults will cause calls to the paging routines to get data from the file as required. As a result for page-on-demand executables, `snooper` will give much lower values for logical bytes read than the reads actually required for the execution of the file. There is some difficulty in instrumentation of the virtual memory system because a number of virtual memory management modules are in use; including the swapping and paging sub-system. Because these routines are critical to the kernel's operation, instrumentation could potentially cause massive performance degradation. The virtual memory routines were considered to be outside the scope of this study.

As determined from the Ultrix `exec` source-code, the header (the first 76 bytes) will always have an associated logical-read operation. However, there will be no other logical traffic associated with the loading of the program file unless the executable is either a pure executable or smaller than a given size threshold (256 Kbytes by default). The reading of data from the program file is done via special routines in the virtual memory system. These routines access directly the vfs/v-node read routines bypassing the logical `read` operation all together. The result is the `snooper` system collects an incomplete record of file accesses for a program file that is being executed.

4.3.7 Off-line processing

The data which `snooper` creates are placed into data-files in a format that is both time and disk-space efficient but which is far from being user-friendly. Off-line processing is necessary to extract and summarise the required information and, because information about sessions of open-close events is particularly desirable, a majority of the off-line processing of `snooper` data involves the creation of open-close session records.

A trace-record formatting program (based on one present in the original `snooper` tools) was used to turn the binary data-file into a human-readable text file listing every transaction in the raw log. This was to allow the development of processing tools that can read the text files allowing faster tool development. The off-line processing tools have been written mostly in the Perl programming language [111] with occasional recourse to the C programming language [47] when this offered an easier alternative.

The trace record formatting program generates a line for each transaction. A sample output record may be:

```
B | 2118298 | 26742 | 0 | -10495 | 8301 | 0 | 8192 | FileF | Li | Rf | Hit |
```

In this first case we have a block-level operation with a block-read at time 2118298, by process ID 26742, owned by UID 0, of the file with inode 8301 on device -10495. The read was of the first 8 Kbytes of the file. It was file data from a remote disk and was in the system's local cache (Hit). Another sample entry may be:

```
rd | 2118302 | 26742 | 0 | 117 | 521 | 0 | 1271 | 4 |
```

In the second entry we have a logical-level read at time 2118302, by process ID 26742, owned by UID 0 and reading from the beginning of the file (0 offset), 1271 bytes. The `snooper`-allocated file ID is 521. The other numbers in this entry (117 and 4) are timing values and of no interest to us for this thesis.

As mentioned above, block entries do not have file IDs associated with them. This was a design decision taken by the original authors of `snooper` to reduce the amount of data the `snooper` system generates and the overhead of information in every block entry.

For the purposes of processing `snooper` data, open-close events need to be clearly defined. This is needed because there are numerous special cases for which an open-close record must be *created*. Open-close sessions can be considered to be summaries of the trace output, tabulating things such as duration of the open-close, the amount of data transferred, the size of the file when it was closed and so on.

A sample single line (shown over two lines for convenience) of the output from the off-line processing software follows:

```
2179130 | 4 | 26747 | 526 | 0 | -10495:8284 |
          791 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 791 | 791 | op-cl
```

For this example, the session occurred at a time of 2179130 and took 4 milliseconds. Process ID 26747 (a process of User ID 0) opened and closed file ID 526 where the file ID refers to i-node 8284 on device -10495. A total of 791 bytes was read from the file (logically), 2 blocks were read from a locally-cached copy of the file. Finally, the file was 791 bytes long when opened and 791 bytes long when closed. The *op-cl* indicates all parts of the open-close record were located. The other zeroes indicate no bytes were written, no blocks were read directly from the file (rather than the cache copy) and no blocks were written to the file either.

This record can then be used to give information about the average and maximum size of files accessed, the duration of open-closes, the amount of data transferred, the number of blocks in cache, the number of blocks written and so on.

Creating open-close sessions

There are several cases to be handled for the creation of an open-close record. We will go through each of the cases which the off-line processing software needed to be able to reconstruct.

Figure 12 shows various open-close sessions including sessions as they cross the beginning and end of the trace period. A regular open-close session where both open and close system calls occur during the trace period is illustrated by Case 1. Case 2 and Case 3 illustrate situations where either the open or the close operations were not recorded as they did not occur during the trace period. Case 4 illustrates the situation where neither the open or the close occurred during the trace period.

Figure 13 shows an additional complication to the process of creating open-close sessions from the trace records. This situation occurs when a parent process opens a particular file and its child process closes the file. The main complication is to which process are the file operations to be attributed. Should the off-line processing software consider this to be two sets of operations and subsequent sessions? For these situations where a file open-close is broken across two processes, a simplification was introduced to reduce the complications of off-line processing; all operations that the child and parent perform are added together to create a single open-close session record. This

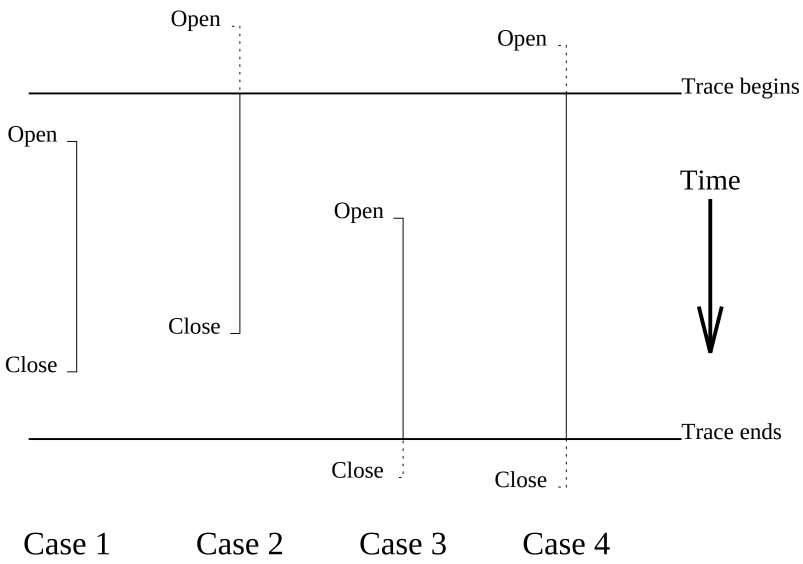

Figure 12: Various ordering of open-close operations that occur during the course of tracing.

simplification was made on the basis of several observations. Firstly, it was noted this occurrence of an open file descriptor being passed to a child did not occur often, 18 times over the whole trace period. Secondly, it was also noted that while the file was open for several processes, the majority of data, (all data in 16 cases), was transferred by one process only. Finally, when compared with passive network monitoring, the open-close sessions have no process related information in them. As the comparison between monitoring systems was on the basis of open-close sessions, independent of the processes responsible, this simplification seemed satisfactory.

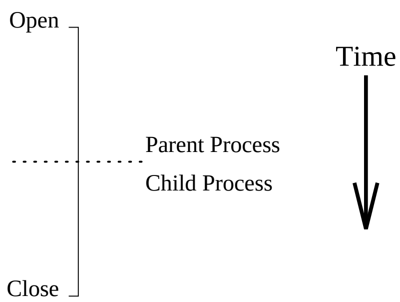

Figure 13: The *passing* of a file ID from a process to its child. This complicates the open and close sessions as the open occurred in the parent process and the close occurs in the child process.

Section 4.3.6 discussed inaccuracies that result from the method by which executables are loaded and inadequacies in the `snooper` instrumentation in this regard. In addition to those difficulties discussed, forming open-close sessions for the execution of programs adds additional complexity.

Figure 14 shows two situations where the initial loading of an executable from the file system in preparation for execution is considered to be an open-close session. Case 1 shows the case where a process is forked and then subsequently `exec`s a particular program. Case 2 shows the case where one process `exec`s another program over the top of itself. In each case the close is simulated for the end of that particular program's life, whether that program exited or another program was subsequently executed. An `open` is simulated at the time data is first read from the file. The kernel does not use the open system call to open files for execution but it does generate read system-calls to read data from the file. These `read`s are then logged by the tracing system. For the fork-exec and exec-exec situations illustrated in Figure 14, the size of the file can be determined from information in the `exec` system call.

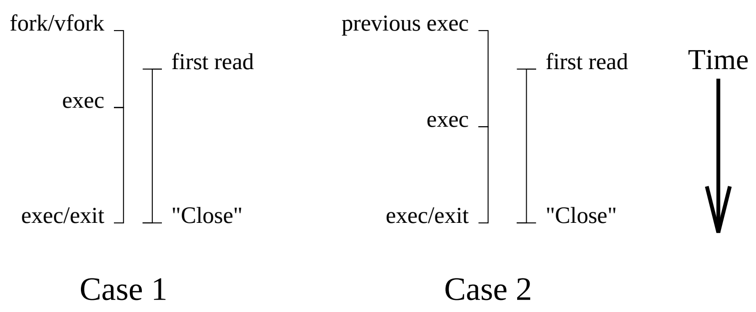

Figure 14: The two situations where the loading (paging) of files from disk for execution is considered an open-close session. Case 1 shows the case where a process is forked and then subsequently executes a particular program. Case 2 shows the case where one process will execute another program over the top of itself.

An objective of the off-line processing software was to sum the number of block reads and writes caused by each open-close session as well as the number of bytes logically read from or written to the file. However, because of the block cache implemented in UNIX, these logical operations may not have a one-to-one agreement with block operations nor will the block operations occur at the same time as the logical read or write, as the block cache is being used to minimise real disk activity. The result is that some logical activities might not cause physical activities until many seconds later. These characteristics further complicate the requirements of the off-line processing software.

An additional issue was that the `snooper` block records do not contain file IDs for the particular block on which an operation occurred, making the matching of logical and block level operations difficult. This situation was alleviated by there being certain records that allowed the matching of pairs of file IDs and device, g-node and g-node generation. Such operations which had both values present were open and close. Unfortunately, there were still a number of situations where this important mapping could not be established, such records were matched by hand.

The off-line processing software made use of a file containing mappings from file ID to device, g-node and g-node generation. This file was formed following a first pass by the processing software. The first pass would display all sets of open-close sessions for which there were no block records and open-close sessions for which there were no logical operations (open/close/read/write). By comparing these entries with the original trace dump, a mappings file was created to give the links between the unknown pairs of file IDs and device, g-node and g-node generation.

The off-line processing software bases the duration of sessions (the time from the open to the close) upon the time of the open and close when they were available. For the unusual situations shown in figures 12 and 14 above, the duration of sessions was based on the first and last operation (logical or block) recorded by the trace package on that particular file. The situation described in Figure 13 required the off-line processing software to keep track of which parents produced which children so that these *cross-generation* open-close sessions could be matched together to generate a single, open-close session record.

Off-line processing implementation

The off-line processing software took records from the trace file and generated open-close session records which were then used in the analysis described in Chapter 6 for the comparison of the full kernel instrumentation and passive network monitoring techniques.

The implementation, in the Perl language, of the off-line processing software is about 1000 lines. Each element (counter) of the output open-close session record is treated as an array item. These arrays were indexed by either file ID or a combination of device/g-node/g-node-generation. The data base [104] extensions to Perl were used to limit the in-memory sizes of arrays. Because some block activity occurs after the logical operation that caused it (blocks to be written caused by logical writes will sometimes not occur until the file is closed), there is a potential for records to cause data to be added to a particular open-close session record well after the logical operations had finished. This meant the trace records needed to have been parsed a considerable time after the last logical operation on the file in order to catch all block operations.

The off-line processing of the trace data, for a trace of 24 hours, took over 130 hours of CPU time, on a high-end workstation (Digital 2100 Server Model A500MP) to process 1.2 million records produced by the kernel trace system into 12 thousand open-close session records. It was determined that a great deal of time in processing the off-line processing phase was needing to search and match certain record types together.

4.3.8 Impact

The impact of the `snooper` system can be measured in a number of different ways such as the complexity of development, as mentioned in Section 4.2 or the speed changes to the machine we consider in this section.

A large concern in the development of kernel instrumentation is the effect on the performance of the system. If there is a degradation of system performance of a system, there is the possibility of altering the behavior of the system itself and that of the users. In order to assess the effect of the instrumentation on the kernel speed, the Modified Andrew Benchmark (MAB) of Ousterhout [71] was used. This benchmark was run under several different configurations of the machine's kernel and the mode of the test — these results are presented in Table 3. The results presented are an average of 10 consecutive tests. Before each test run, a single MAB was run to place the local block

	Regular kernel		Snooper kernel (inactive)		Snooper kernel (active)	
Phase I Creating directories	3.0	(0.0)	3.3	(0.8)	3.4	(0.5)
Phase II Copying each file	18.9	(0.3)	19.9	(0.6)	22.2	(0.7)
Phase III Recursive directory stats	17.3	(0.8)	17.7	(1.8)	18.9	(1.0)
Phase IV Scanning each file	15.3	(0.5)	16.0	(0.5)	17.1	(0.3)
Phase V Compiling and linking	146.8	(2.0)	148.9	(1.8)	160.1	(2.9)
Total	201.3		205.8		221.7	

Table 3: Results from running the Modified Andrew Benchmark for a non-instrumented kernel, for a `snooper` kernel without tracing enabled and for a `snooper` kernel performing tracing. All values are in seconds — values in parenthesis are standard deviations.

cache into a consistent state. For these tests the file systems of the machine are located on a central NFS server. However, for all the runs performed with `snooper` operating, the trace file was recorded on disks local to the test machine.

The results indicate that the difference between the regular kernel and the instrumented kernel with `snooper` inactive are relatively small, when compared to overall benchmark times, and are due to the extra code in each system call that tests if the `snooper` system is inactive. It is interesting to note that even such a small number of additional instructions to perform add so significantly to the overhead of the system, 4.5 seconds over the total benchmark.

The largest difference, though, is between the inactive and active instrumented kernel. The additional overhead of logging activities into trace buffers, logging trace results to disk and buffer management have added approximately 10% of overhead to the system. In real terms this is a difference of over 20 seconds between the regular kernel and the active `snooper` kernel.

Chapter 5

Network Monitoring

This chapter describes the network monitoring performed for this thesis. The network-monitoring system captures and processes NFS traffic between clients and server.

5.1 Objectives

Passive network monitoring can collect much of the information full kernel instrumentation is currently used to collect; such data as the number of active users on a system, the amount of data transferred to and from disks, which files are being accessed. However, a great deal of system monitoring information used by researchers is based on records of sessions delimited by the opening and closing of files. The kernel-instrumentation data was processed into such records. For network monitoring to be used as a replacement for full kernel instrumentation, the ability to generate such an open-close session record is required. This ability should give comparable information about the data transferred, the amount of data in a particular file and so on. With such open-close-session records we can then study the system in all aspects that would have been done using kernel instrumentation. Thus giving the ability to calculate such things as:

- the average or maximum amount of data transferred per file, in a given time,
- the average or maximum time a file is used, or
- the average amount of data in a file.

Using network monitoring much of this information must be inferred from the transactions which are observed. This chapter discusses the processes used for inferring open-close sessions from network traffic.

5.2 Network monitoring

Network monitoring, if done by a independent machine, can gain information about two communicating systems without impacting on their work or changing their behavior. Network monitoring does not require any changes to the system(s) being monitored and allows the simultaneous monitoring of multiple machines. However, network monitoring relies on there being suitable, useful information passing through the network. Without enough useful information, network monitoring may not be able to estimate satisfactorily what has occurred on a particular client.

With the network-based file systems in use or under development, such as NFS [87], Sprite [72] and Andrew File System (AFS) [66, 42], clients require many or all file-system operations to be done through the network. Because the network will be carrying all file-system traffic between clients and server, network monitoring can collect and interpret this data to give information on the file operations which clients are performing.

Network monitoring of distributed file systems gives access to information about physical blocks as they are read from and written to the server by clients. Additionally, there is a great deal of other information exchanged between client and server which is related to directory operations and cache consistency that enable users of this technique to estimate the operations clients have performed.

Network monitoring must rely exclusively on information present in the communications between client and server. It is generally not possible, nor desirable, to add additional information to this communication stream. As a result, network monitoring requires that the procedure for processing the incoming data incorporates a specific knowledge of the type of communications used and can make use only of what data are available in the communications channel.

In the case of NFS, the post-processing software must estimate operations on the client using not only the reading and writing of data from the server but also the other messages used to co-ordinate client caches and obtain directory information. Knowledge of the type of communications system (in this case, the details of NFS) can be considered to be a main requirement in effective network monitoring.

To perform network monitoring, a researcher requires a machine that can interface with the network and capture all data traversing it. Network monitoring can now be performed by today's faster workstations with the appropriate software. Typically, such a network-monitoring machine will be a workstation using software based around the Network Interface Tap (`NIT`) [103] packet capture mechanism from Sun Microsystems and the `packetfilter` [27] capture mechanism from Digital. The machine will record data from the network to a local disk. In a number of cases the workstation has sufficient power to perform some rudimentary processing of the data which can reduce the amount needing to be saved to disk.

Because the distributed file system used in this study is NFS, which uses RPC [55] for each transaction, the network monitor need look only for the two parts of the RPC exchange; the request and the reply. This is a considerable advantage over, for example, monitoring the stream of characters to a terminal. In a terminal's character stream, each character must be collected and the full stream reassembled to gain any understanding.

Additionally, RPC uses XDR [54] to allow communications between machines that do not share common hardware or operating-system software. Because of this, a network monitor can be any particular type of machine from any vendor, running any operating system. A Sun workstation can monitor a network of Digital machines because the type of data on the network is independent of the hardware or operating system of the machine that generated it.

In order to generate the desired open-close sessions from the monitoring of NFS traffic between client and server, the network monitor must process collected data following its capture.

First and foremost, the network monitor must filter the traffic of interest from the data captured from the network. In this situation only RPC transactions pertaining to NFS are of interest. Each individual RPC request or reply can be made-up of a number of Ethernet packets and a whole RPC request or reply will be retransmitted if it was partially lost in transit. Data to be discarded includes repeat copies of the same transaction as well as the parts of the RPC transaction that are not relevant to the monitoring (typically the data payload of NFS read and write transactions). The request and reply of each RPC transaction must be matched together and incomplete transactions (presumably retransmitted) must be discarded.

The network monitoring system must process the NFS transactions and estimate from them the open-close sessions that have occurred on the clients of the distributed file system.

5.3 A network monitoring implementation

The implementation carried out in the present study consists of four parts:

1. network monitoring and data extraction,
2. data translation, filtering and NFS/RPC call processing,
3. data check-pointing and compression, and
4. post-processing.

The network monitoring software implementation used in this study is a toolkit made up from two parts: `rpcspy` and `nfstrace`. This toolkit was implemented by Blaze [11] to enable network monitoring of a large distributed file system based upon NFS [11, 15, 16, 13]. Additionally, work of others [24, 1] has been based on measurements taken using `rpcspy` and `nfstrace`. The software is designed to operate on any machine that supports the Sun `NIT` capture mechanism or the Digital `packetfilter` capture mechanism. `rpcspy` collects network traffic, extracts NFS/RPC requests and replies, matches these requests and replies and constructs concise one-line-per-transaction records. `nfstrace` provides post-processing, creating open-close session records from the trace records generated by `rpcspy`.

It was required that the `rpcspy/nfstrace` operate with Ultrix 4.3a or SunOS 4.1. The software needed no modification to operate on the systems used, having been designed in a similar environment. `nfstrace` was modified to give duration information on the length of open-close operations. Both `rpcspy` and `nfstrace` were modified to give additional data in the trace, in particular some file-attribute information. `nfstrace` was augmented with a view to finding information about its method of operation and to be able to evaluate the open-close identification heuristics it uses.

Additionally, a time-stamping compression filter was written to reduce the quantity of data generated by `rpcspy`. This filter was also able to change output files, either after a given amount of time or a given quantity of data (number of lines).

5.3.1 Network monitoring and data extraction

As mentioned previously, the `rpcspy` utility is built upon either the Sun `NIT` capture mechanism or the Digital `packetfilter` capture mechanism. These facilities provide user-level software access to raw data packets traversing the network to which the system is attached.

In each of these mechanisms the user configures what type of data is to be extracted from the network and which hosts on the network this data could have come from (this could be set to all hosts). The device will return packets that satisfy this filter into a buffer.

It is assumed by the network capture mechanisms (`NIT` and `packetfilter`) that the buffer will be emptied continuously by the user's program. If the incoming queue of data overflows the buffer the extra data is discarded.

`rpcspy` configures the incoming network interfaces to accept any IP [76] packets from all hosts on the local network and then expects the network interface to pass these incoming packets to it.

5.3.2 Data filtering, data translation and NFS/RPC call processing

After receiving data from the network interface, `rpcspy` uses a series of filters, translators and algorithms to piece the NFS transactions together. `rpcspy` filters the incoming packets, accepting only those packets that are from the internet user datagram protocol (UDP) [78]. It then filters the packets again, selecting only those destined for the NFS service port on the assumption that these are NFS packets. This NFS data is then translated from XDR [54] format into a data-format suitable for the local machine to process.

Once the data are in the required format, `rpcspy` checks the RPC header and if the packet is a reply to a previously recorded request the pair are matched together and processed. If the packet is a new request it is queued in a list of requests pending replies.

Once the RPC request and reply have been paired they are processed according to the type of NFS transaction the RPC request/reply is carrying. Each NFS transaction has data of interest extracted and a transaction record is recorded along with a time stamp of when the transaction was complete.

An example pair of RPC transactions is printed below. These are hexadecimal dumps of the two Ethernet packets that make up a particular RPC transaction. In this case, the transaction is to get the attributes of a particular file.

An RPC request (transmitted using UDP/IP on Ethernet):

```
0000  08 00 2b 24 34 2b 08 00 2b 1c 26 9d 08 00 45 00   | ..+$4+..+.&...E.
0010  00 94 a6 38 00 00 ff 11 79 fb 82 c2 4a d7 82 c2   | ...8....y...J...
0020  4a c9 03 ff 08 01 00 80 0b b4 0f 90 e8 6b 00 00   | J...........k..
0030  00 00 00 00 00 02 00 01 86 a3 00 00 00 02 00 00   | ................
0040  00 01 00 00 00 01 00 00 00 30 2f 4b ac e9 00 00   | .........0/K....
0050  00 18 62 75 73 6d 61 6e 2e 72 64 74 2e 6d 6f 6e   | ..busman.rdt.mon
0060  61 73 68 2e 65 64 75 2e 61 75 00 00 00 00 00 00   | ash.edu.au......
0070  00 00 00 00 00 01 00 00 00 00 00 00 00 00 00 00   | ................
0080  00 00 04 15 00 00 02 00 00 00 02 3d 3b 52 02 00   | ...........=;R..
```

```
0090   00 00 02 3d 3b 52 00 00 00 00 00 00 00 00 00 00   | ...=;R..........
00a0   00 00                                              |  ..
```

and the matching RPC reply:

```
0000   08 00 2b 1c 26 9d 08 00 2b 24 34 2b 08 00 45 00   | ..+.&...+$4+..E.
0010   00 7c 48 7f 00 00 ff 11 d7 cc 82 c2 4a c9 82 c2   | .|H.........J...
0020   4a d7 08 01 03 ff 00 68 36 79 0f 90 e8 6b 00 00   | J......h6y...k..
0030   00 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00   | ................
0040   00 00 00 00 00 00 00 00 00 02 00 00 41 ed 00 00   | ............A...
0050   00 07 00 00 00 00 00 00 00 00 00 00 02 00 00 00   | ................
0060   20 00 ff ff ff ff 00 00 00 02 00 00 15 04 00 00   | ...............
0070   00 02 2e a7 11 ae 00 01 be 35 2f 4b 83 d6 00 03   | .........5/K....
0080   a5 d7 2f 4b 83 d6 00 03 a5 d7                     |  ../K......
```

After the network capture mechanism and `rpcspy` have filtered the transaction and `rpcspy` has reconstructed the particular NFS operation, `rpcspy` would record this transaction as (shown here on three lines for clarity):

```
793488611.244329 | daneel.rdt.monash.edu.au | busman.rdt.monash.edu.au.0 |
    getattr | "131500000200000036b514020200"|
           ok, {040755, 0, 1024, 0x2, 785561935.347853 }
```

Table 4 shows the NFS transactions and data fields recorded by `rpcspy`. Several transactions such as **root** and **writecache** are typically not used in NFS implementations although `rpcspy` can process them because they are part of the NFS standard.

Transaction Type	Data fields
null	real time, server, client, transaction type, user ID
getattr	real time, server, client, transaction type, user ID, NFS file handle, call status, file attributes
setattr	real time, server, client, transaction type, user ID, NFS file handle, requested attributes of file, call status, new attributes of file (at the end of the call)
root	real time, server, client, transaction type, user ID
lookup	real time, server, client, transaction type, user ID, NFS file handle of directory, file name (to be looked up), call status, NFS file handle for file name, attributes of the file name
readlink	real time, server, client, transaction type, user ID, NFS file handle of link, call status, contents of symbolic link
read	real time, server, client, transaction type, user ID, NFS file handle, offset, number of bytes to read, call status, number of bytes actually read from the file at the given offset
writecache	real time, server, client, transaction type, user ID
write	real time, server, client, transaction type, user ID, NFS file handle, number of bytes to write, call status, attributes of the file at the completion of the write
create	real time, server, client, transaction type, user ID, NFS file handle of directory, file name, call status, NFS file handle of created file, attributes of created file
remove	real time, server, client, transaction type, user ID, NFS file handle of directory, NFS file handle of file to remove, call status

continued on next page

Transaction Type	Data fields
rename	real time, server, client, transaction type, user ID, NFS file handle of source directory, NFS file handle of file to be renamed, NFS file handle of destination directory, NFS file handle of file destination, call status
link	real time, server, client, transaction type, user ID, NFS file handle of source file, NFS file handle of destination directory, file name of destination file, call status
symlink	real time, server, client, transaction type, user ID, NFS file handle of destination directory, file name of the symlink to be created, target of symlink, attributes for symlink, call status
mkdir	real time, server, client, transaction type, user ID, NFS file handle for destination directory, file name of new directory, file attributes for new directory, call status
rmdir	real time, server, client, transaction type, user ID, NFS file handle of target directory, file name of directory to remove, call status
readdir	real time, server, client, transaction type, user ID, NFS file handle of directory, call status
statfs	real time, server, client, transaction type, user ID

Table 4: `rpcspy` transaction types and data fields

5.3.3 Data check-pointing and compression

Monitoring a network can generate a considerable amount of data. A program like `rpcspy`, even condensing the data to the extent that it does, can generate several megabytes of data in less than ten minutes of monitoring a moderately-busy network. Consequently there is a potential for running out of storage space on the monitoring machine. A solution is to checkpoint and compress the output and, if possible, to change file names to allow previous records to be copied to tape.

A compression and check-pointing filter was written, which was option-driven to change to a new file either after a given amount of time or a given number of lines of input. This filter was able to incorporate the current time into the name of the file for easy identification of the sequence of log files. Also the filter could, if required, generate compressed output using the standard `compress` utility in UNIX which routinely obtained a 4:1 compression ratio. This compression and check-pointing filter makes the data output from `rpcspy` more manageable.

5.3.4 Post-processing

The post-processing of the `rpcspy` data is done by the utility `nfstrace`.

NFS has no explicit open or close transaction, so this piece of software must piece together open-close sessions from the NFS transaction log of `rpcspy` using a heuristic based on the operation of NFS. `nfstrace` makes an estimation of open-close sessions that have caused the NFS transactions to occur. Partially, this relies on consistency in NFS implementations, that is, for every `open` system call (independent of whether the file is to be read to and/or written from or just accessed) an NFS `getattr` transaction is generated.

However, open-close sessions to `read` or `write` data handle the data itself in significantly different ways, although it should be noted the `write` transaction case is easier to handle, because the cache does not have as dominant an effect on the `write` operations. As a result, much of the special case handling `nfstrace` must do applies only to NFS `read` transactions.

NFS write

When a user on an NFS client writes data to a file, it will either be written directly to the server or be written to the server at the close of the file on the client. This means that during an open-close session on a client, writes are synchronous with the server during that session. By the time the open-close session has ended, all data written by the client into the file will have been carried over the network.

Because of this, a program such as `rpcspy` will see these writes as one or more NFS write transactions. The result is the write system-call will be written to the file as an NFS transaction. However, if only a partial block was written, the write will not occur until the block is full, a periodic operation occurs, causing modified data blocks to be written to disk (commonly each 30 seconds) or the file is closed.

NFS read

When a user on an NFS client requests data from a file, this data will either be available locally, in the cache of the client or it will need to be read from the server.

Figure 15 shows the flow of a read request on an NFS client. The shaded boxes indicate operations that can be recorded by the `rpcspy` program.

This flow diagram shows how the cache-consistency model of NFS works.

- If a particular block is not present in the cache of the local client, it is retrieved from the server. In this case `rpcspy` will see a read transaction occur.

- If a particular block is present in the cache of the client but the cache copy has not been checked recently, the client will perform a `getattr` transaction. The `getattr` transaction will return the time and date of the file on the server. The client can now check the cache copy of the file; if it is older than copy of the file on the server, the cache copy is out of date.

- If the cached copy on the client is out of date, the client will remove it from memory and force a new copy to be read from the server. In this case `rpcspy` will see a `getattr` transaction followed by a read transaction.

- If the cached copy on the client is not out of date, the client will not need to obtain a copy from the server, and returns data from the cached block on the server to the user. In this case `rpcspy` will see a `getattr` transaction used by the client to check and confirm the validity of the cached copy of this file.

- If the cached copy on the client has been checked recently the client will not check with the server in any way. In this case `rpcspy` will not see any transactions

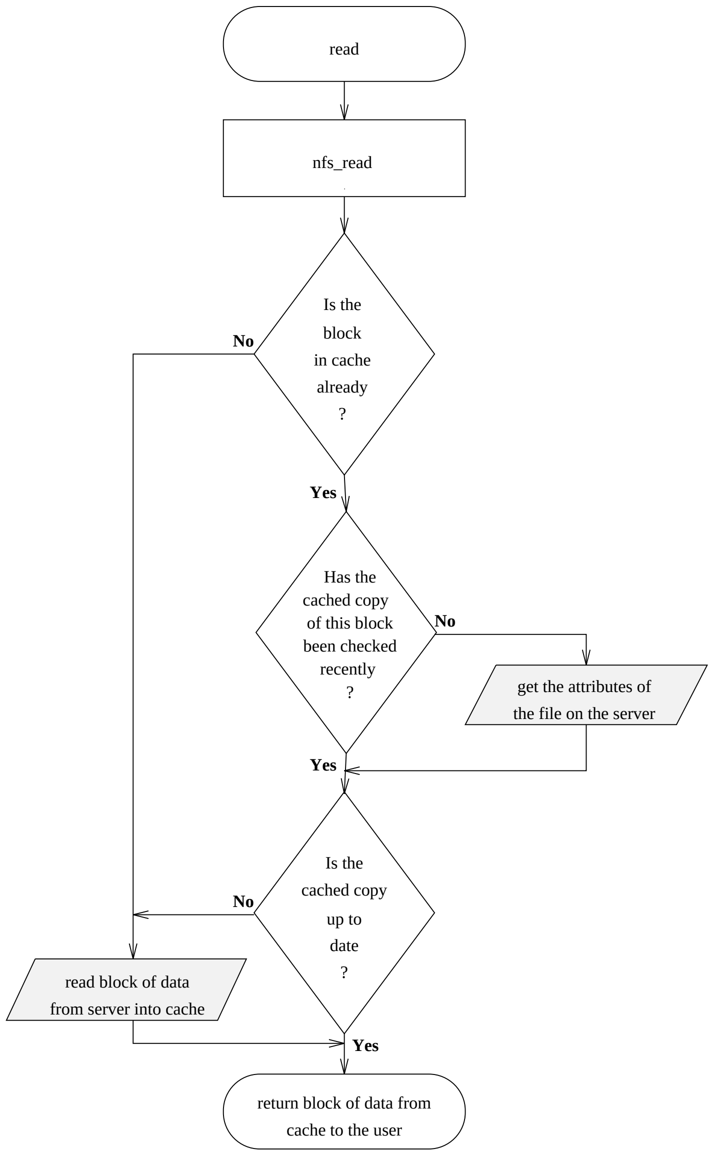

Figure 15: The flow of a read request on an NFS client. The shaded boxes indicate operations that can be recorded by the `rpcspy` program.

between client and server.

It is important to note that `rpcspy` will not be able to detect the read operation if the block to be read is already in memory, the cached copy has been checked recently and it was up to date when last checked.

`nfstrace` open-close sessions

Although `rpcspy` generates a large amount of data (an entry corresponding to each transaction), `nfstrace` only uses a small number of transactions in its analysis: the NFS `read`, `write`, `setattr`, `getattr` and `lookup` transactions.

The heuristic used by `nfstrace` is important because `nfstrace` must identify the open-close sessions which have occurred. Blaze designed `nfstrace` using several pre-miss; the primary one is that NFS is consistent in its operations when a file is opened. A `getattr` transaction on the file is performed for every open system-call and then the read or write system-calls will cause combinations of NFS `read`, `write`, `setattr` and `getattr` transactions.

The heuristic `nfstrace` uses is best described by the conditions under which it will record an open-close session record. These conditions are used by `nfstrace` to conclude that particular client has closed the file; as shown in the following list.

1. If the previous operation on a particular file was more than a given time ago, this will cause an open-close session record to be generated for the previous operation(s). An access to the file with an NFS `getattr`, `setattr`, `read` or `write` transaction all reset this timer.

2. If the current transaction is a `setattr` and there were previous data transfers from read or write transactions, or the previous transaction occurred more than two seconds previously.

3. The current transaction is a `setattr` and size of the file is being set to zero, yet some data were previously written to the file.

4. The current transaction is a `read` and there was previously a `write` on the file.

5. The current transaction is a `read` to the first byte of the file and that byte has been accessed previously.

6. The current transaction is a `write` and there were previous data transfers from `read` or `write` transactions or the previous transaction occurred more than two seconds previously.

7. The current transaction is a `write` to the first byte of the file and that byte has been accessed previously.

The recording of an open-close session record will reset all counters and flags, such as the amount of data transferred and whether the first byte has been accessed.

From these rules it is apparent that `nfstrace` will be unable to correctly interpret several situations. For example, `nfstrace` will be unable to detect open-close sessions where the file is open for both read and write operations, such as a database file. It will not correctly interpret the situation where data is written to the first block of a file a number of times; each time the first block is rewritten, `nfstrace` will consider this to be the start of a new open-close session on the same file. These cases are more extensively discussed with the presentation of results in Section 6.9.1 and in Chapter 7 in which a discussion of ways these rules can be refined to improve the results `nfstrace` is given.

Table 5 indicates which system calls in Ultrix 4.3a cause NFS `getattr`, `setattr`, `read` and `write` transactions to occur.

NFS Transaction	System call
write	In unusual circumstances (an open-close session on the client where the cache is disabled) the `write` system-call will directly cause NFS `write` transactions, otherwise the NFS transaction will not occur until an 8 Kbyte file block is filled (writing to the file in 8 Kbyte blocks).
	continued on next page

NFS Transaction	System call
	Either the `fsync` or `sync` system-calls will cause any outstanding blocks to be written to the file server which results in NFS `write` transactions for those blocks.
read	the `read` system-call will cause an NFS `read` transaction in some circumstances (as described above in Section 5.3.4) the `write` system call (counter-intuitively) can cause an NFS `read` transaction if the amount of data to be written for a particular block is less than the size of a block. This is because the NFS `write` operation can `write` to the server only in units that are integral multiples of the block size.
setattr	The `fchmod`, `chmod`, `fchown`, `chown`, `utimes`, `sync`, `fsync`, `truncate` and `ftruncate` system calls will all cause NFS `setattr` transactions.
getattr	The NFS getattr transaction is used for getting directory information as well as cache consistency, so it may be caused by the `unlink`, `creat`, `close`, `fsync`, `access`, `stat` and `lstat` system calls in addition to being caused by the `read` system call.

Table 5: A table of the NFS transactions and the system calls that can cause their occurrence.

A comparison of the heuristic for `nfstrace` and the causes of certain NFS transactions reveals the behavior of `nfstrace` when processing records from the Ultrix NFS implementation. For the rule-base used in `nfstrace` it can differentiate only between read and write open-close sessions. `nfstrace` will be unable to determine if a file was both read from and written to in the same session; instead it will infer that two separate sessions have taken place.

Some of the rules used by `nfstrace` do not seem intuitive. However, rules such as those surrounding the `setattr` transaction mean that `nfstrace` can, in this example, interpret correctly a truncate system call on a pre-existing file. If a pre-existing file is truncated when it is opened, any previous transactions involving that file will be recorded.

During the development of the rules used in the heuristic for `nfstrace`, Blaze did not have access to kernel level tracing of the systems he wished to monitor. He used a simple benchmark incorporating the `ls`, `cp`, `touch` and `wc` programs to develop and evaluate new rules. The `ls` is particularly important because it will cause numerous NFS `getattr` transactions which can potentially be interpreted as read operations.

Blaze found that after several hours of operation, `nfstrace` was able to detect 100% of the writes, 100% of the uncached reads and 99.4% of the cached reads. Blaze then concludes that cached read operations were over-reported by 11%, even though the `ls` command was 50% of the benchmark activity. Blaze reflected that, while it was encouraging to obtain this level of accuracy from the system, it was not conclusive. He suggested that the particular workload of the tests cases may have been misleading `nfstrace` in unanticipated ways.

`nfstrace` output

`nfstrace` will take the transaction log of `rpcspy` and generate an open-close session log. A sample entry from this log is show below (shown on two lines for clarity):

```
787451296.297919 | 3.299941 | read |
        daneel:15150000bd100100ef35ff6005f8 | alquist.2015 | 0 | 864
```

For this entry at time `787451296.297919`, user ID `2015` on the machine `alquist` read file `15150000bd100100ef35ff6005f8` from the server `daneel`. The read transaction was of 864 bytes and `nfstrace` considers the whole file was read from the contents of the client cache. This session lasted `3.299941` seconds.

Such open-close session records enable us to give information such as estimations of the durations of open-close sessions and the amount of data potentially accessed and transferred in that time. By generating this information researchers are able to use `rpcspy` and `nfstrace` as tools for system monitoring. The tools are suited particularly to the simultaneous monitoring of many machines in a distributed file system.

5.4 Implementation restrictions

While the `rpcspy/nfstrace` system was able to give results comparable to kernel instrumentation (Chapter 6), there were, sometimes significant, differences in the results of the two systems.

`rpcspy` has three major drawbacks. The first relates to restrictions in the network interface supplied for a particular machine. The second drawback is related to the restricted information available to `rpcspy` at the time of operation. Finally, `rpcspy` depends on the insecure, unencrypted data of the RPC transactions to be carried over the network to which the `rpcspy` machine is attached. As a result, `rpcspy` cannot be as easily used to extract passive network monitoring information from networks using secure RPC implementations such as transmitting RPC using Kerberos [100]. This limitation is not easily overcome.

Additionally, `nfstrace` has limitations in its implementation. Primarily these are able to be improved, as discussed in Chapter 7, however several, such as the amount of information available to `nfstrace` are not easily altered. The limitations of the `rpcspy/nfstrace` system is discussed in the following sections.

5.4.1 Network packet capture mechanism drawbacks

The `rpcspy/nfstrace` tools depend heavily on the ability of the network interface of the machine on which they are being run to capture all traffic passing through the network. Packet-loss by the network interface does not have a linear relationship with network utilisation. The network interface will not lose data when utilisation is low. However, data loss will increase as utilisation increases to a point beyond which it will be unable to accept any increase in the data-transfer rate and the amount of data it can process will flatten out no matter what the utilisation beyond that point.

Protocol type		Sub-protocol		Types of packet	Packet size	%
Internet Protocol (IP)	67	UDP	36.9	NFS	155	24.7
					1500	12.2
		TCP	30.1	(all)	80	15.1
					192	9.0
					1272	6.0
Novell Netware (IPX)	33	-		-	155	19.8
					768	13.2

Table 6: A breakdown of the traffic mixture used for testing `rpcspy` response to Ethernet utilisation

A study was performed to quantify the potential data loss of `rpcspy` and to calibrate the network interface `rpcspy` uses. To perform these tests satisfactorily, a network analyser capable of full utilisation measurements on Ethernet was required. A Hewlett Packard Internet Advisor Model J2522A was used both to perform measurements and to generate artificial loads on the network. The `packetfilter` mechanism used was in a DECstation 3100 running Ultrix 4.3a, the `NIT` mechanism used was in a Solbourne SC2000 (a machine compatible with the Sun Sparcstation 2) running SunOS 4.1.3.

Tests of `rpcspy`, where the network was loaded artificially, used the traffic breakdown in Table 6 which was based on an analysis of the network over several 24-hour periods.

The `packetfilter` facility of Ultrix offers some configuration options. In particular, the size of the packet buffer, where packets processed by `packetfilter` are placed for collection by the user process, can be set. The `NIT` mechanism in SunOS does not offer this configurability. The default configuration and an optimum (largest configurable buffer size) for `packetfilter` in addition to the `NIT` mechanisms are compared in Figure 16. This figure shows the percentage of unprocessed Ethernet packets versus Ethernet network utilisation. It is apparent that not only are the characteristics of the `NIT` mechanism poor beyond 10% utilisation but that the `packetfilter` mechanism did not demonstrate the same level of loss until utilization was close to 50%. The `packetfilter` mechanism showed no loss until over 15% utilization, a stage by which `NIT` mechanism loss was close to 25%.

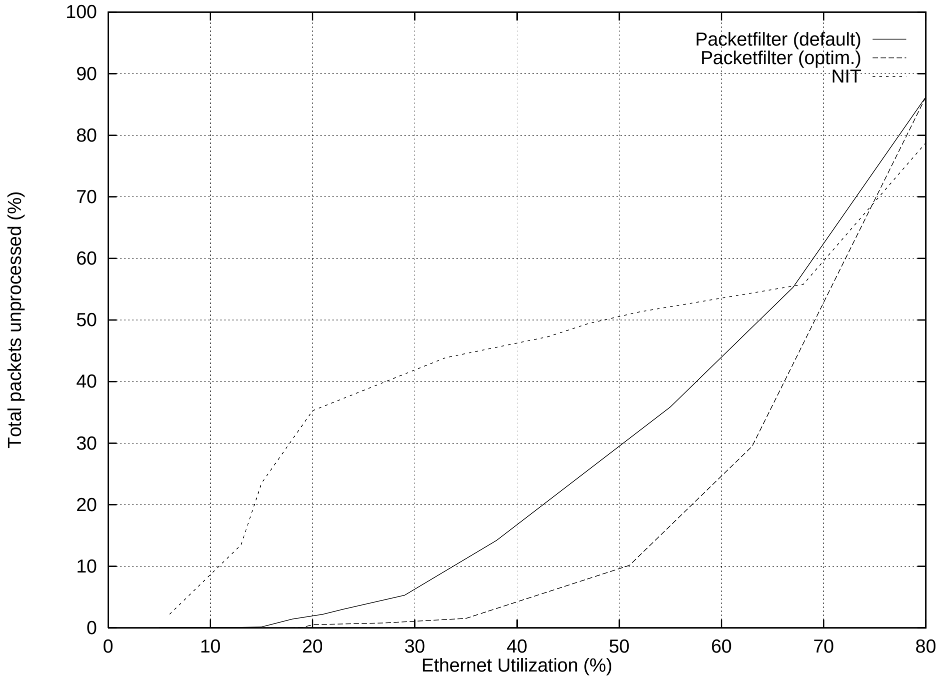

Figure 16: A comparison of Ethernet utilisation versus packet loss for various workstation Ethernet interfaces. `packetfilter` default and optim(um) are two configurations of the Ethernet packet capture facility of the Digital DECstation, `NIT` is the Ethernet capture facility in Sun Microsystem's SunOS.

A significant issue in `rpcspy` is the combination of processing overhead on the client, which is imposed by the need of `rpcspy` to match RPC transactions, and the packet-loss characteristics of the Ethernet interface which `rpcspy` is using. Figure 17 shows the number of NFS transactions versus Ethernet utilisation. The Ethernet utilisation in these tests is almost purely NFS traffic. By using NFS traffic exclusively we are able to establish the maximum number of NFS transactions each `rpcspy` system is able to process in a given time period. The Hewlett Packard test equipment recorded the actual number of NFS transactions that occurred over this time. For this test the `packetfilter` was left in the default configuration.

The test shows that each system has a maximum number of packets it can process. The `NIT`-SunOS system is limited to processing about 175 NFS transactions per second. The default configuration `packetfilter`-Ultrix combination appears to be limited to processing approximately 260 NFS transactions per second. It is important to note this was a stress-testing of the `rpcspy` and that such NFS loads were not a characteristic of the network to which these machines were connected. From the figures in Table 6 we can see that 36% of the total Ethernet traffic is from NFS. It would not be true to say of this 36% that half of the number of NFS Ethernet packets would be the count of NFS transactions. Such a simplification would not allow for there being incomplete NFS transactions (the loss of the request or reply in a transaction) nor would it allow for NFS transactions that required more than one pair of network packets (transactions where the data payload required two or more Ethernet packets). In each of these cases `rpcspy` does not need as much processor time as if it had had a complete NFS transaction. As a result, the test network operating at 12% utilisation could mean less than 72 transactions per second in a mixed load with a variety of NFS traffic rather than the 200 transactions per second that Figure 17 stress-test indicates.

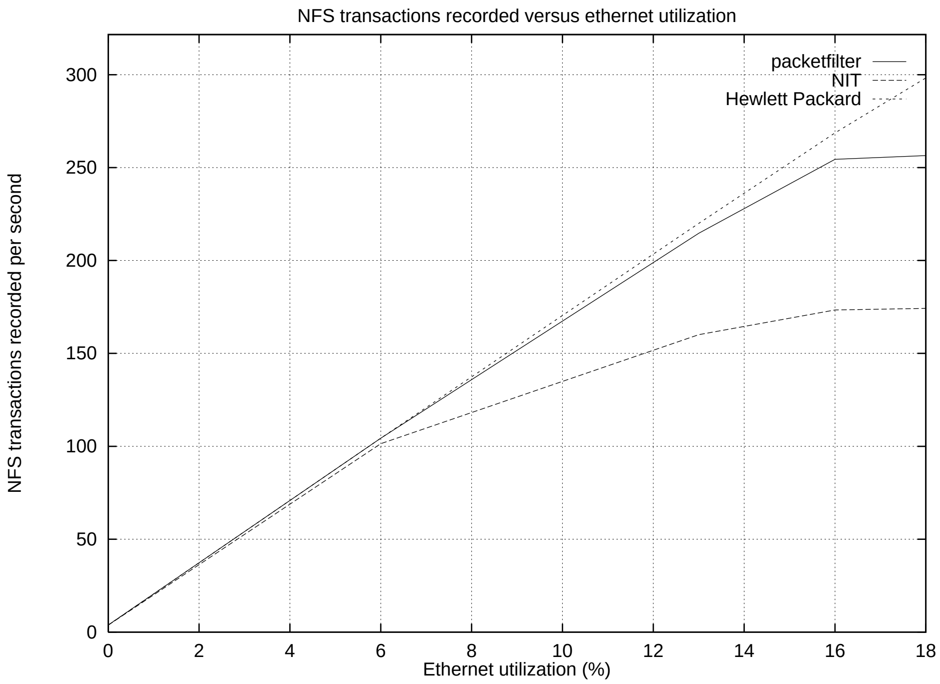

Figure 17: The number of NFS transactions versus Ethernet utilisation for the `NIT` and `packetfilter` capture mechanisms. Results from a network analyser recording no packet loss is also given (Hewlett-Packard).

The exact cause of such data loss is not certain but it could result from limitations in the hardware of the network interface and/or limitations in the software of the packet collection and filtering mechanism.

This characteristic is unfortunate. It is during the time when the network is busiest that utilisation across a distributed file system will be potentially highest. Because there is potential for `rpcspy` based tools to lose data about transactions at busy times, studies such as file sharing, a situation that would be more likely to occur at busier times, would be affected adversely.

Such drawbacks could be potentially overcome by the use of faster workstations with faster hardware network interfaces. However, this may not be as easily solved if the problem is due principally to a poor software implementation in either the network packet capture mechanism (`NIT`/`packetfilter`) or `rpcspy`.

While this characteristic loss does exist, it is significant only above about 10% utilisation for the `packetfilter` mechanism. Boggs et al. [17] comment that most Ethernet loads are well below 50%, close to 5% of the network capacity and the network on which measurements were taken supports this observation with a maximum load over 24 hours of no greater than 18%.

5.4.2 Restrictions in available data

`rpcspy` is restricted in the information to which it has access; it must work with only the information which is carried within the network. Because of this, `rpcspy` cannot determine such information as which process is responsible for a particular transaction. Additionally, `rpcspy` cannot differentiate between two different process accessing the same file. The only differentiation `rpcspy` can make are those of file, user and machine. The result of this is that if a particular user is simultaneously accessing a file with two different programs `rpcspy` might misinterpret the transactions that result. An example of where this might occur is when a user is simultaneously editing a source file and compiling that source file.

The reason for this alteration of behavior is that the cache will make the two different sets of logical operations seem to be, typically, the same set of block operations. What `rpcspy` estimates has happened depends on the actual transactions that occur across the network. For the same set of operations, slight changes in timing will cause different transactions to occur across the network. As a result, it would be difficult to deduce what `rpcspy` would report in such instances.

5.4.3 `nfstrace` restrictions

`nfstrace` interprets `rpcspy` data and estimates the open-close sessions that caused the NFS transactions `rpcspy` has traced. As discussed above, this means `nfstrace` depends heavily on `rpcspy` collecting all available data from the network.

`nfstrace` can suffer errors as a result of misinterpreting NFS transactions or because information has been omitted from the NFS transactions. Also `nfstrace` has implementation-imposed restrictions. For example, it is unable to identify open-close sessions were data is both read from and written to the file. Baker et al. [8] found open-close sessions that involved data read from and written to the file accounted for 1% of all transferred bytes and 1% of the number of open-close sessions. Such an estimation would indicate misinterpretation but it would not cause a dramatic impact on results. It is, however, worth noting.

Errors of misinterpretation

`nfstrace` will interpret some transactions incorrectly. As can be seen in Table 5 a number of different operations can be responsible for an NFS `getattr` transaction occurring. `nfstrace` assumes that a `getattr` transaction has occurred in relation to a block being read from the file cache. However, there is a potential for the `getattr` to have been caused by a program such as `ls` getting information about the files or even by the deletion of a file.

`nfstrace` has the potential to interpret the write system-call incorrectly. This is because of the nature of write which performs all block transactions in the modulus of the block size. For an existing file, the remainder of a partial block write must be read first. There is a potential for `nfstrace` to incorrectly misinterpret this as a separate read-session on the file.

`nfstrace` uses an heuristic based on accesses of the first byte of a file. Although most file accesses (90% according to Ousterhout et al. [73]) will be read sequentially from the file system, there is a potential for `nfstrace` to misinterpret up to 10% of file accesses. Additionally, because `nfstrace` considers that access to the first byte is a good indicator of a new open-close session, there is potential for `nfstrace` to incorrectly assess an open-close session where the data are read or written a multiple number of times.

Errors of omission

`nfstrace` depends heavily on the information supplied to it through the NFS transactions. Because of the lack of an open or close operation in NFS, `nfstrace` must attempt to derive open and close operations from the other NFS transactions that occur.

Figure 18 shows a number of open-close sessions with their associated read/write operations. The same figure shows a set of typical, related block activities.

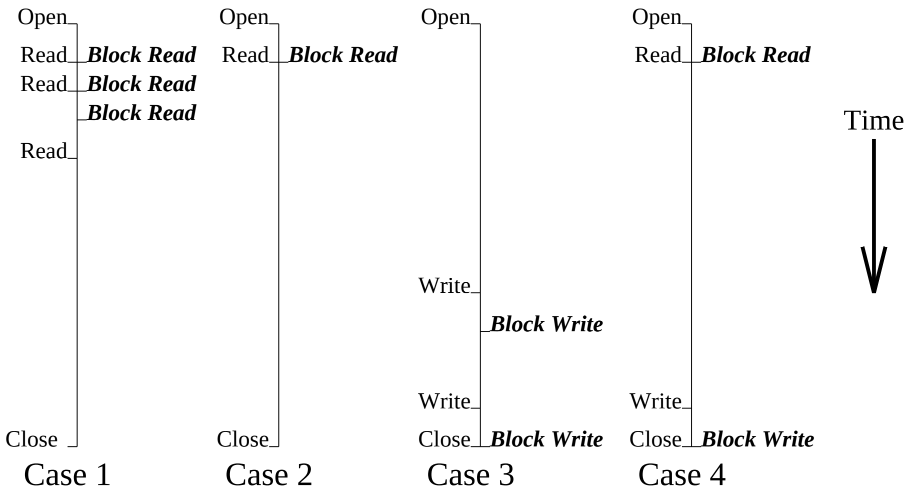

Figure 18: A variety of open-close sessions with read, write or read/write activity, also shown is a typical set of related block activity.

Case 1 shows a typical open-close session consisting of a number of read operations. The corresponding block operations could be caused directly by the read or could be the result of a kernel read-ahead operation. As a result `nfstrace` might consider this session to have taken less time than it actually did because all the block operations were finished very early in the session even though another read occurred much later in the session.

Case 2 shows a simpler open-close and read although the same problem exists. `nfstrace` might consider this session to have taken a very small amount of time when compared with the length of the total open-close session operation.

Case 3 shows an open-close session where data are written to a file. Delays in the writes themselves occurring because of write-behind policies mean that `nfstrace` could easily misinterpret the length of time taken by this open-close session.

Case 4 shows an open-close session where both a read and a write have occurred. `nfstrace` will see the corresponding transactions and interpret this is an open-close session of reading and another open-close session of writing.

These errors make a very broad assumption that in all four cases the data of the blocks read from or written to the file caused exactly one read or write transaction to occur over NFS. If this simplification is removed, the number of different possible misinterpretations increases many times. Exactly what `nfstrace` will interpret depends on what transactions (if any) occur as a result of each operation. An example might be Case 3: for the initial open system-call the client will often cause a `getattr` transaction to occur. `nfstrace` may misinterpret this as being another operation, in this case most likely a read that was a successful cache-hit of a file that will towards the end of the open-close session have data written to it.

Another example is Case 1 If this file had been accessed recently by a program on the client, the file will be in the cache of the client. If the access was recent enough, no transactions, not even a `getattr` for cache consistency, will be caused. As a result `nfstrace` would not even see this session occur.

Figure 19 shows that the block operations upon which `nfstrace`'s record will be based do not necessarily correspond with the logical open and close operations in an open-close session.

5.4.4 Local versus remote file system performance

The whole operation of `rpcspy` and `nfstrace` depends on file-system traffic of interest passing through the network. In quite recent times there has been a tendency for machines to have large disks supplied with them [24], but before that, there had been a trend towards the centralisation of disk resources. This, after all, is one reason that distributed file systems have become so popular. In the test environment, machines

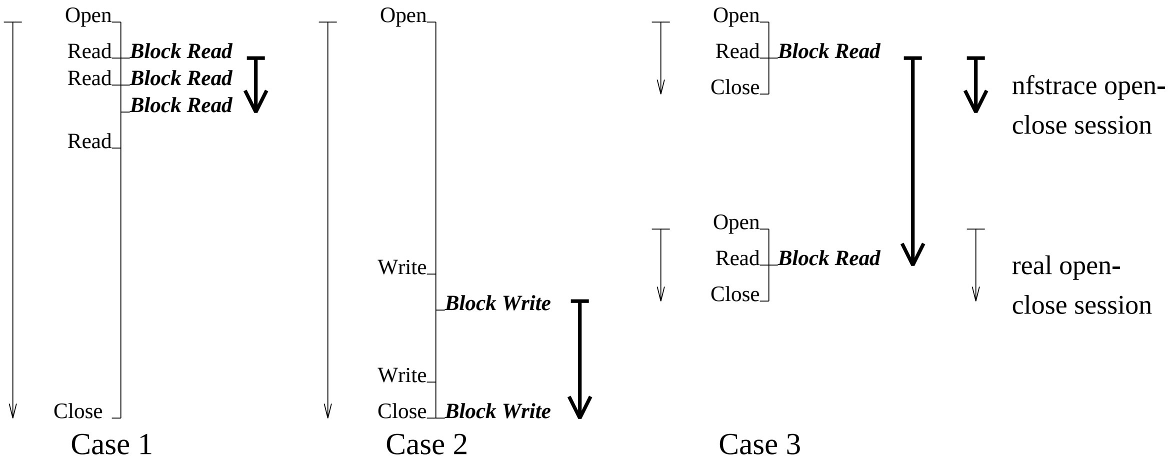

Figure 19: Several open-close sessions as generated by `nfstrace`are compared with the actual open-close session that occurred. The open-close session generated by `nfstrace` depends heavily on the type of NFS transaction each block access will invoke.

have small local disks principally for swap space and for the `/tmp` and `/var/tmp` file systems. However, with `/tmp` and `/var/tmp` transactions going to the local file system, no NFS transactions are generated and `rpcspy` cannot capture the transactions so `nfstrace` cannot interpret them.

This seemed unsatisfactory because these open-close sessions were potentially of great importance. The `/tmp` and `/var/tmp` file systems generally contain short-term temporary files so it could be expected that these file systems contain a significant percentage of the small-duration open-close sessions. This seemed a reasonable assumption given that the traffic to these file systems is commonly temporary files from editors and compilers.

A solution used for this study is to move the `/tmp` and `/var/tmp` file systems to a remote disk. This means the transactions can be captured and `nfstrace` can interpret this data. Because this move was desirable, it was important to establish the impact on performance of moving the file system from local to remote disk would have.

Table 7 shows variations in the performance of the Modified Andrew Benchmark (MAB) with various disk configurations. A typical user would find their work closest to the second test with the test done on a remote disk and with the machine having a local `/tmp` and `/var/tmp` file systems. The difference between this and the machine with remote `/tmp` and `/var/tmp` file systems was less than 7% of time over all the tests, which seemed an acceptable impact.

	Local test		Remote test		Remote test	
`/tmp`/`/var/tmp` location	Local		Local		Remote	
Phase I Creating directories	2.9	(0.5)	4.4	(0.8)	3.0	(0.0)
Phase II Copying each file	14.5	(0.7)	15.0	(0.5)	18.9	(0.3)
Phase III Recursive directory stats	15.2	(0.6)	15.9	(0.8)	17.3	(0.8)
Phase IV Scanning each file	14.5	(0.7)	15.3	(0.8)	15.3	(0.5)
Phase V Compiling and linking	128.8	(0.4)	138.1	(1.0)	146.8	(2.0)
Total	175.9		188.7		201.3	

Table 7: Results from running the Modified Andrew Benchmark on a system with combinations of a remote and local disk. Firstly with the test to the local disk, with local `/tmp` and `/var/tmp` file systems, then with a test to remote disk with local `/tmp` and `/var/tmp` file systems and lastly a test to remote disk with remote `/tmp` and `/var/tmp` file systems. All values are in seconds, values in parenthesis are standard deviations.

Chapter 6

Comparison of Monitoring Techniques

6.1 Introduction

This chapter compares and contrasts two system-monitoring techniques. The `rpcspy/nfstrace` passive network-monitoring tools are compared with the `snooper` kernel instrumentation package. In particular, the results from `snooper` are used as a baseline against which the accuracy of the results of `rpcspy/nfstrace` can be compared.

The comparisons of `snooper` and `rpcspy/nfstrace` have been done by using simultaneous traces of a single machine over a 24-hour period. The trace of this machine was performed from 11:00 a.m. Monday, 12th of December, 1994, until 11:00 a.m. the following day. The machine traced was a Digital DECstation 3100 configured with 20Mbytes of memory, running Ultrix 4.3a. This machine was configured with a local disk for virtual memory swap activities. The `rpcspy` trace was recorded to local disk so as not to perturb the results with extraneous network activity. During the 24-hour period, a loss of 1.5% of total Ethernet traffic was recorded giving a loss of 0.6% of NFS transactions from the total recorded trace.

The `nfstrace` post-processing tool uses a heuristic (described in Section 5.3.4) that incorporates a timeout used to determine how long an open-close session will last. The value is user-selectable but the default value of 135 seconds was used throughout the analysis described in this chapter.

Figure 20 shows the relative instrumentation points of `snooper` and `rpcspy`. The differences in the points of instrumentation lends to differences in the types of information available to each system. Of particular note is the fact that `snooper` can record information about file operations between the user program and the kernel, whereas the file-operations `rpcspy` can record are those between the kernel and the remote file system. A major difference between these sets of communications traffic is that `rpcspy` can record only operations that were not able to be serviced by the client's block cache, as the cache will prevent many data requests from ever going to the NFS level. In particular, the cache will prevent most consecutive accesses to the same file from becoming duplicate NFS requests, and many short-duration file accesses may never have their associated data transferred at the NFS level.

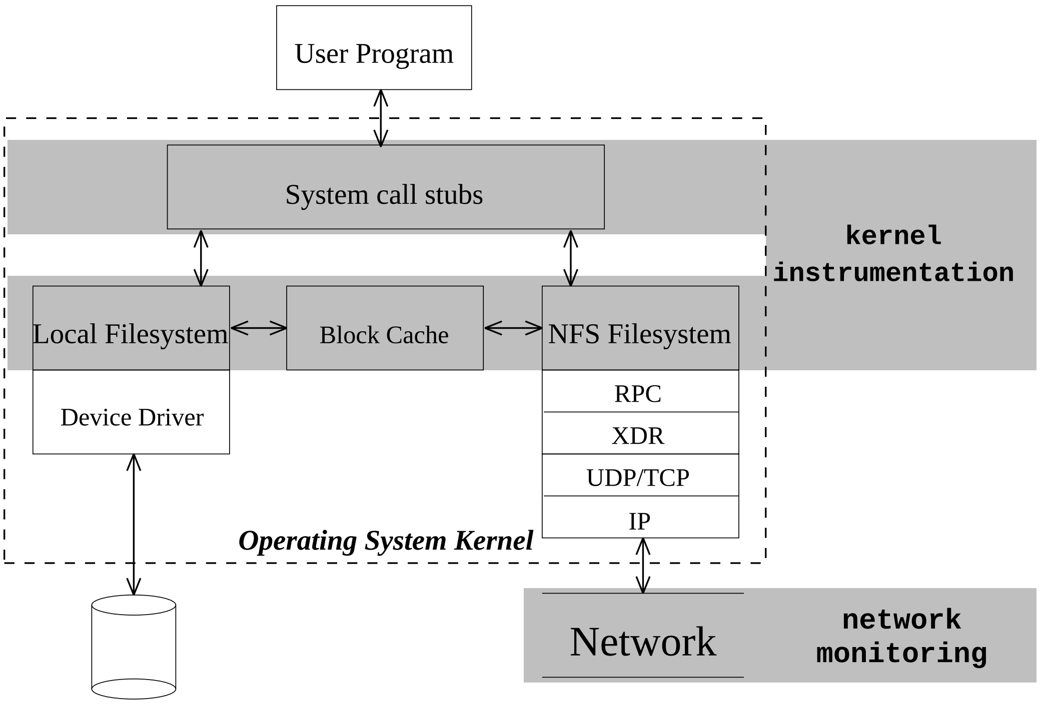

Figure 20: The data flow between a user program and an NFS file system. Instrumentation points for kernel instrumentation (`snooper`) and network monitoring (`rpcspy`) are indicated. This diagram compares the difference in the information available to each system. In particular, one instrumentation point, `snooper`, is before the cache and the other, `rpcspy` is after the cache.

This ability to filter transactions associated with duplicated access to the same file-data and those associated with short-duration files is one of the design objectives of caches [73, 98, 8]. The cache is **intended** to filter file-system transactions from needing to be sent to the file system. The cache performance of a system is related directly to the system's overall performance because so much of a system's operation is tied directly to file tasks and thus to the performance of the cache [71].

6.1.1 Excluded data

Due to the problems noted in Section 4.3.6, all file transactions associated with the reading of executable files by either the `snooper` or `rpcspy` systems were removed from the trace data before processing. It should be noted the problems associated with executable files stem from a shortcoming in the `snooper` instrumentation, not the `rpcspy/nfstrace` system. Records pertaining to the `snooper` trace file itself were removed from the output records during the processing stage.

While the removal of all execution transactions may seem to change the results presented, the remaining data still permitted a satisfactory comparison of the two monitoring systems and that the amount of potential comparison-error which would be introduced due to the inclusion of incomplete execution records by `snooper` was not justified. Additionally, file system traffic resulting from the loading of executable files was excluded from previous studies such as Ousterhout et al. [73] and Baker et al. [8] for the same reasons.

6.2 System traffic

The characteristics of the total file-system communications traffic are commonly-used measurements. In the case of diskless workstations, the measurements are important for insuring that the networks have adequate transport capacity and that the servers of diskless workstations have adequate service capacity. In any sort of workstation such values define the required capacity for disk interfaces, as well as being used in cache and bus design [117, 80, 73, 8, 85].

A comparison of communications traffic to and from the file system at the logical level, and of the communications traffic at the `rpcspy` network level, are not strictly comparable, as each set of measurements was made on a different side of the cache. However, one of the objectives of `nfstrace` was to estimate operations that occurred

Particular measurement	interval length	`snooper` (bytes)	`nfstrace` (bytes)
Total data transferred		86,644,530	46,967,724
Average data transferred	10 seconds	10,028	5,436
Peak data transferred		5,120,000	5,048,320
Average data read		7,468	2,590
Peak data read		5,120,000	3,914,935
Average data written		2,560	2,846
Peak data written		5,120,000	5,048,320
Average data transferred	10 minutes	601,698	326,165
Peak data transferred		19,028,550	17,015,414
Average data read		448,103	155,387
Peak data read		10,427,845	7,144,164
Average data written		153,595	170,777
Peak data written		8,600,705	9,289,091

Table 8: The total data transferred for the system. Peak and average values for 10 second and 10 minute intervals are also given.

at the user level by analysing the data communications traffic between client and server and the transactions used by the client to ensure the contents of the cache are up to date. As a result, while `rpcspy/nfstrace` cannot generate information on exactly what data were transferred between the user programs and the file system (including the NFS file-system routines and the block cache), it can calculate the exact amount of data transferred between the NFS file system and NFS server.

Table 8 gives a summary of results for the comparison period. It is immediately apparent that for total data-transfer values, there is a major difference in the value `nfstrace` estimates for the total data transferred when compared with `snooper`. They differ by a factor of 1.7. From these results it is equally apparent that over the course of a long term analysis (24 hours) results for peak values and `write` data are comparable for the two systems.

Peak values display this characteristic because they typically involve data that is not suitable for long term storage in the cache because it is too large or too volatile [73, 8, 98] and that this characteristic is independent of the particular load a machine is under [71]. As a result, the similarity between transferred data, particularly peak values, would remain across any sample taken. In comparison, values for the total quantity of data transferred over time is not similar. The difference between `snooper` read averages and `nfstrace` values is not surprising. The client cache will eliminate successive NFS transactions for reading data from the NFS server and as a result, `nfstrace` cannot record the data transfer that had occurred at the logical level.

Figure 21 shows plots of data transferred over time as recorded by `snooper` and `rpcspy/nfstrace`. Heavy data transfer, particularly heavy writing activity, between 7am and 11am is due to the testing of image encoding algorithms (by another researcher) on this machine requiring the reading and writing of large image files.

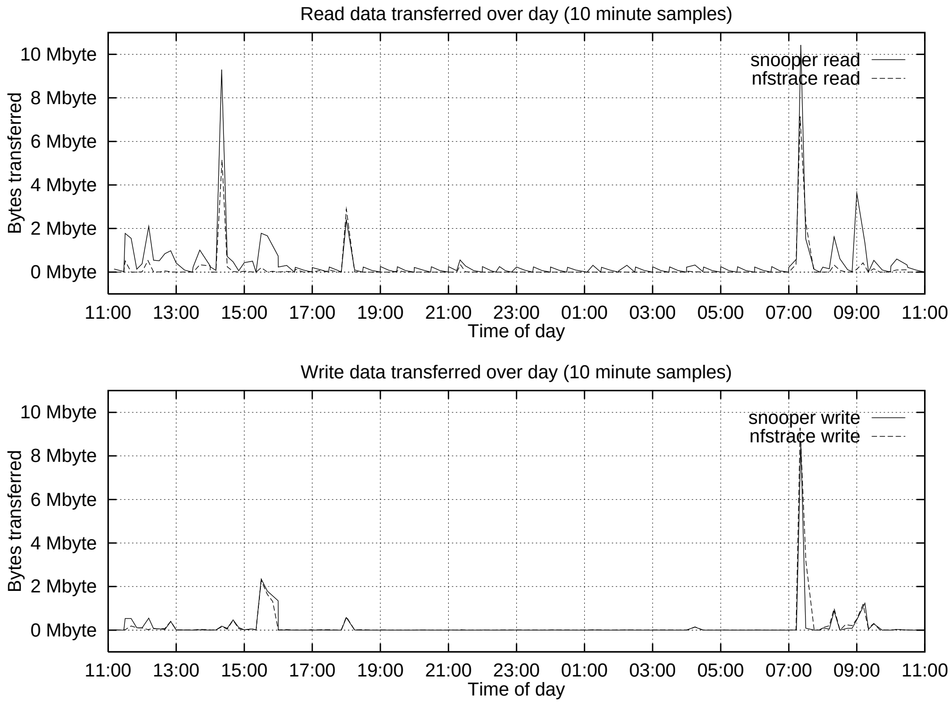

Figure 21: Read and Write transfers as recorded by kernel instrumentation (`snooper`) and network monitoring (`nfstrace`). A quiescent system from 19:00 until 7:00, the machine is busy during the daylight hours.

The graph of read-data shows an example of the difference between data gained from `snooper` instrumentation and that available to `nfstrace`. Periodic accesses by automatic jobs account for the regular communications traffic logged during the 19:00 to 07:00 period. Because this communications traffic involves the regular execution of programs, commonly with little other file-system activity, the cache of the client contains all the software and associated data files to be used on these regular occasions. The result is that approximately 300 Kbytes of logical data are read each 30 minutes

File System	Function and Contents
`/`	`root` file system, also includes `/var` and `/tmp`. Top-level file system containing temporary directories and logging directories.
`/usr`	contains *standard* software distribution, in addition to libraries and include files for the current system.
`/var/spool/mail`	contains each users' mail file.
`/usr/local`	contains locally installed software.
`/usr2`	home directories for a group of users.
`/packages`	contains commercial software packages and collections of project specific data (in this case image data).

Table 9: A breakdown of the file systems of the study and their respective functions.

at the `snooper` level, but `rpcspy` records negligible read-activity between client and server over the same period.

The reason `nfstrace` is not as accurate for records of raw data transfer is because NFS transactions do not contain significant information about blocks read from the cache of the client. The only specific `read` data available to `rpcspy` about data transfers that occur is when data are read by the client from the server's disks.

6.3 File system transactions

As discussed in Section 2.1.2, each file system is used typically for a particular purpose. For example, one file system contains the user's directories, another file system contains executable files for the system, etc. The DECstation analysed in this study did not have any local file systems, apart from that used to store trace data locally, and a local swap disk. Table 9 lists the different file systems the client accessed over the trace period and the tasks each file system served.

A breakdown of the type of data transferred to and from each file system can be used to assist file system configuration decisions. Such decisions can include: which file systems generates so much server traffic it should be locally attached to this machine and how widely a particular file system is used. A breakdown of each file system's communications traffic is given in Table 10.

It is important to note that at the system-call level, as recorded by `snooper`, there

File System	snooper		rpcspy/nfstrace	
total				
/	31,736,478	(36.63)	5,863,351	(11.59)
/usr	2,941,480	(03.39)	1,446,003	(02.86)
/var/spool/mail	4,385,788	(05.06)	3,142,239	(06.21)
/usr/local	1,455,692	(01.68)	965,364	(01.91)
/usr2	38,660,513	(44.62)	35,251,413	(69.66)
/packages	7,464,579	(08.62)	3,934,663	(07.78)
read				
/	27,267,823	(42.26)	2,853,302	(12.73)
/usr	2,941,480	(04.56)	1,446,003	(06.45)
/var/spool/mail	3,836,074	(05.94)	2,311,208	(10.32)
/usr/local	1,455,692	(02.26)	965,364	(04.31)
/usr2	21,561,247	(33.41)	10,895,621	(48.63)
/packages	7,464,579	(11.57)	3,934,663	(17.56)
write				
/	4,468,655	(20.20)	3,010,049	(10.68)
/var/spool/mail	549,714	(02.49)	831,031	(02.95)
/usr2	17,099,266	(77.31)	24,355,792	(86.38)

Table 10: Total data, read data and write data transferred per file system as measured by `snooper` and `rpcspy/nfstrace`.

is a characteristic breakdown of these transactions. Of particular note is a very large percentage of operations associated with the `/` partition. The large number of transactions on this partition will have been potentially compounded because the `/tmp` and `/var/tmp` directories resided on the `/` file systems. `/tmp` and `/var/tmp` can potentially carry a large percentage of operations because temporary files are traditionally created in this directory structure [108, 81].

Table 10 shows a moderate similarity between between the result from the two monitoring methods. Notable exceptions are traffic involving the `/` partition, and read-traffic in general. While differences between values for read between `snooper` measurements and those of `rpcspy` can be explained as resulting from the cache mechanism filtering read requests between client and server, the read traffic for the `/` partition is particularly pronounced. This difference is likely to result from a high usage of system files located in the `/etc` directory being accessed, resulting in the corresponding cache entries always being valid. Such examples are `/etc/passwd`: the list of users able to use a system, `/etc/hosts`: a static table of the systems known to this client and for this particular version of UNIX, /etc/fstab: a file listing the file systems that should be mounted on this client.

The notable difference in the recorded quantities of read and write data for /usr2 is a reflection of the volatile nature of files on this file system. In particular, software for image encoding was being developed and a cycle of

1. edit program
2. compile program
3. run program

will exist. This development cycle, during stage 1, results in source-code files being written to the NFS server (and seen by rpcspy) but not necessarily read from the NFS server. During stage 2, in addition to the source-code files, libraries will be read only once from the server and then may remain in the local cache while being used repeatedly. Finally, during stage 3, while file transactions relating to the loading of the executable file itself have been removed, this program takes as input a raw image stream and outputs an encoded image stream. On consecutive runs the raw image stream could have remained in local cache.

It should be noted that the ratio of read-to-write traffic already greatly favours the write-traffic for /usr2 as measured with the snooper system but the cache activities, filtering traffic, increase this ratio.

Significant differences between the amount of write traffic recorded by each monitoring system for both the /usr2 and /var/spool/mail file systems can be attributed to the block cache needing to transfer data to and from the file system in block-sized pieces. The result of this is that a modification of one byte in a file will result in the writing of a whole block (8 Kbytes for these file systems) to the file system.

From this breakdown it is clear that, while activities on the / file system are responsible for a large percentage of logical file traffic, block caching seems to reduce that quantity of data transfer by a factor of up to 6. By comparison, the /usr2 file system is responsible for a higher quantity of data transfer and, in the development and

	interval length	`snooper`	`rpcspy`
Number of active users			
Maximum	10 minute	4	4
Average		1.6	2.2
Maximum	10 second	3	1
Average		1.0	1.0
Total bytes transferred per active user			
Maximum	10 minute	6,342,850	5,477,752
Average		263,535	109,820
Maximum	10 second	5,120,000	5,048,320
Average		11,422	18,404

Table 11: The maximum and average number of active users over given intervals and the total quantity of data transferred per active user in those intervals.

balancing of file systems, it would be important to establish whether this is a transient condition or a regular trend for communications traffic for that particular file system.

6.4 System users

Table 11 presents several values related to the number of active users on the system and the amount of traffic generated by them. Such tabulations have been made in a number of previous studies and are useful in the estimation of the load a user may impose on a system as well as the worst-case scenarios for this load.

The differences in Table 11 for the number of users are most likely the result of `snooper` recording the real User ID (UID) associated with each logical operation and `rpcspy` recording the effective User ID associated each NFS transaction. This difference comes about because programs such as `inetd` (the internet service daemon) [29] perform operations as one user and spawn programs that will run as another user. The result is that counts of active users made through `rpcspy/nfstrace` usually differ by a value of one when compared with the active user count from `snooper`.

Average-data-utilised per user indicates that cache-hit rates are, once again, absorbing a substantial quantity of communications that would have occurred between each user and the file system. It is interesting to note that the maximum values recorded by each system are almost identical. This is most likely due to the transfer of large amounts of data, causing the client's cache to be quickly overrun with new data. As a result, only a minimal amount of data is cached at all during this time.

6.5 Files

As files are the common unit of data accessed on a file system, information about the range of files accessed, as well as the working size of those files, enables developers to determine the necessary size of file caches, to establish common working-set sizes and to quantify other related measurements.

As had been mentioned earlier, the difference in the average file size for the `/` file system was predictable. This will principally be a result of a large number of small, system-related files not requiring access from the NFS system. The differences in other values will have resulted from the caching of, and repeated accesses to, active files (even if these files were active only for a short period of time). In this context, an active file is one which is accessed one or more times.

Table 13 lists the number of different files recorded at the `snooper`, `rpcspy` and `nfstrace` levels. At the `rpcspy` level, this is a count of every file that had a read or write NFS operation performed on it. The filtering characteristic of the cache is obvious when comparing the number of files that had logical operations performed on them at the `snooper` level with the number of files for which data was read from or written to at the `rpcspy` level. Larger differences for the `/` file system will have been as a result of accesses to the large number of regularly-accessed system files located there. These files would be accessed often and be modified infrequently, as a result, having a long cache life.

The results in this table show an area where the estimation method used by `nfstrace` can generate discrepancies. `nfstrace` must estimate traffic to and from files that have not caused any `rpcspy` `read` or `write` transactions. With the exception of `/var/spool/mail`, `nfstrace` must estimate additional operations for files on each of the five file systems. `nfstrace` has estimated extraneous operations on files of `/usr2` and underestimated these operations for the other file systems, `/` in particular.

File system	snooper	rpcspy/nfstrace
/	43,378	227,880
/usr	437,123	287,006
/var/spool/mail	267,887	201,417
/usr/local	10,226	12,310
/usr2	42,713	46,067
/packages	1,316,180	440,371

Table 12: A comparison of the average size for files accessed on each particular file system.

As discussed in Section 5.3.4, `nfstrace` estimates operations on files from a combination of NFS `read`, `write`, `setattr` and `getattr` transactions. The estimates of files which did not involve NFS `read` or `write` transactions would have resulted from `setattr` or `getattr` operations. By using `getattr` transactions alone, there is potential for `nfstrace` to confuse `getattr` transactions caused by such operations as getting a directory listing with those transactions being used to validate the contents of the client cache.

In comparison, the graph of Figure 22, a normalised cumulative distribution of the number of files of each size, shows that the estimation calculated by `nfstrace` compares well with the results of `snooper`. The two significant differences between the results of `nfstrace` and those of `snooper` which lead to disparities in the graph are for the number of zero-length files and the number of files which were approximately 700 bytes in length.

In the first case, `nfstrace` is not able to generate accurate estimations of accesses to various zero-length files and creates many more than actually existed. This may most likely be due to `nfstrace` being unable to differentiate `getattr` transactions for directories and `getattr` transactions being as a result of the opening of a zero-length file. Files with a short life-span can present a problem to `nfstrace`. This is because given a short enough life-span between file creation, the writing and reading of data, and file deletion, no NFS `read` or `write` transactions may occur during the open-close session. As a result `nfstrace` is not easily able to record data transfer operations on files with a short life-span. In the second case, related case, `nfstrace` has underestimated the number of accesses to various files 700 bytes in length. In addition to the reasons above, it is possible that `nfstrace` evaluates many of the 700-byte file accesses as being zero-byte files accesses because of the block cache absorbing the small-file transactions.

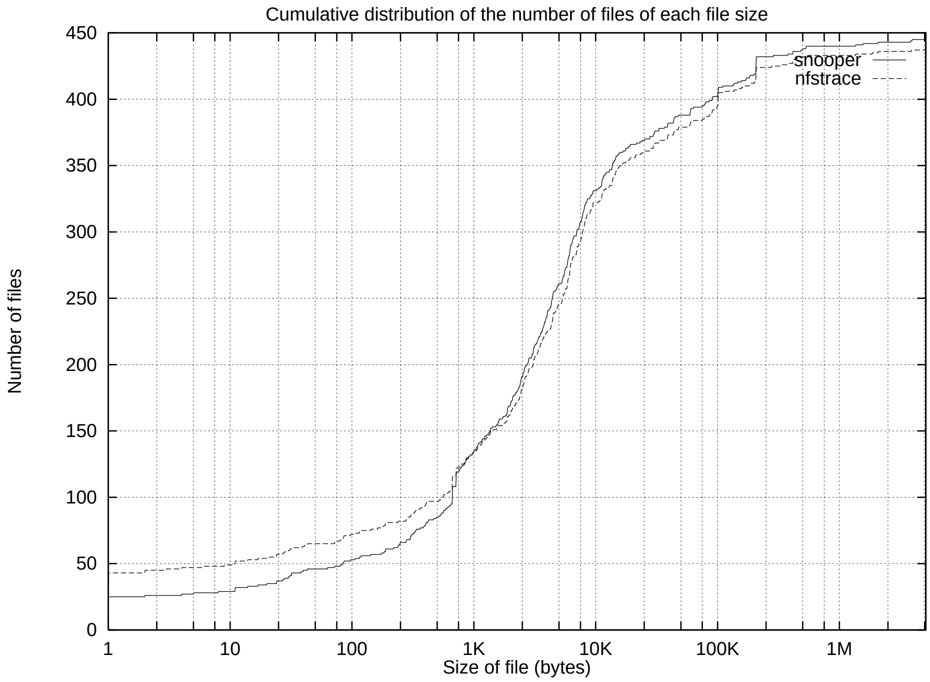

Figure 22: Cumulative distribution of number of different files accessed versus file size. From this graph we can deduce the number of times different files less than a given size have been accessed. For example both techniques suggest that over 150 of the files accessed are 1 kbytes in size or smaller. *Note:* the file size axis is logarithmic.

The following table, 13, gives a breakdown of the number of different files accessed by the system during the measurement period. These values are consistent with the hypothesis that `nfstrace` was unable to evaluate correctly accesses to zero length files. The average file size for `/` would strongly confirm this, although the `/packages` results run counter to this. This strong counter-example could be due to the unusual nature of files on that particular file system: we note also that `nfstrace` results count one less file for that file system; a single large file would have modified this average considerably.

While there are notable differences in each of Tables 13 and 12, the results from them, in addition to those of Figure 22 show that `nfstrace` was able to give results broadly comparable with those of `snooper`.

File System	snooper		rpcspy		nfstrace	
/	111	(24.89)	68	(17.13)	98	(22.37)
/usr	10	(02.24)	8	(02.02)	8	(01.83)
/var/spool/mail	3	(00.67)	3	(00.76)	3	(00.68)
/usr/local	49	(10.99)	46	(11.59)	48	(10.96)
/usr2	269	(60.31)	269	(67,76)	278	(63.47)
/packages	4	(00.90)	3	(00.76)	3	(00.68)
Total	446		397		438	

Table 13: A breakdown per file-system of the total number of different files accessed during the trace period. The values in parentheses are each count as a percentage of the total number of files.

6.6 File open-close sessions

The open-close session of a particular file is a concept around which a number of measurements are based. A number of studies have used such measurements; examples include file sharing, file utilisation and various cache studies [73, 8, 42, 48, 89, 6].

Such open-close session measurements include the length of time a particular file is open, the amount of data accessed in that time, the amount of data *potentially* accessed (the size of the file opened), what sort of open-close session was involved, was the file opened to read and/or write operations, etc.

The number of open-close sessions as well as a breakdown of the relative types, are tabulated in Table 14. The implementation of NFS under Ultrix includes the synchronous writing of modified data blocks to the file system at the close of a file. This means that `nfstrace` can only potentially miss write operations on files that ultimately leave the file with zero length, for example, some sort of temporary file. Read-only open-close operations have no such certainty. As a result, `nfstrace` will not be able to generate results for reads on files that occur in close succession (where the cache contents are still valid). Additionally, `nfstrace` may not correctly interpret `getattr` NFS transactions used to validate the cache. The result is `nfstrace` will either miss some open-close sessions altogether, incorrectly interpret NFS transactions as not being an open-close session, or incorrectly consider that the NFS transactions from two or more separate open-close session are from the same open-close session.

	snooper		nfstrace	
read entries	7442	(88.07)	1749	(68.51)
write entries	557	(06.59)	804	(31.49)
read-write entries	35	(00.41)	-	-
null entries	416	(04.92)	-	-
Total	8450		2553	

Table 14: The count of open-close sessions each monitoring system interprets. Additionally, a breakdown of these open-close sessions into read-only, write-only, read-write and null open-close sessions is shown. A null session is where no data are read from or written to the file (although the file was opened). Values in parentheses are the percentage of the total number of files each type represents.

The larger number of writes recorded by `nfstrace` will certainly include the read-write operations `snooper` recorded. `nfstrace` is unable to differentiate read-write sessions and would consider each of such operations as a separate read and write session. Null open-close sessions, where no data are transferred and the file is simply closed, would not be able to be detected by `nfstrace`. Instead, `nfstrace` interprets any file open, were that the only operation on a particular file, to be a reading of an unknown amount of data from the client cache.

Because the borders between read and write operations cannot be determined accurately, `nfstrace` will tend to collect successive open-close sessions together, interpreting them as one longer open-close session. As a result of this, the average duration of the open-close sessions reported by `nfstrace` may be higher than the durations reported by `snooper`.

Tables 15 and 16, record the open-close sessions broken down by type of open-close operation per file system basis, and by file system per operation. Firstly, Table 15 shows the full effect of the cache filtering, combined with `nfstrace` incorrectly interpreting information available, causing open-close sessions to be removed. This is especially the case for the `/` file system. The results for `/var/spool/mail` are a good example of where `nfstrace` has misinterpreted the NFS `getattr` transactions as open-close sessions because mail files are often checked for new mail resulting in `getattr` transactions. By way of comparison, a better result is given for the `/packages` file system. Files from this file system are unlikely to be able to be kept in cache for

File System	session type	snooper		nfstrace	
/	read	6415	(90.33)	818	(63.21)
	write	354	(04.98)	476	(36.79)
	read-write	35	(00.49)	-	-
	null	298	(04.20)	-	-
/usr	read	123	(73.21)	61	(100.00)
	null	45	(26.79)	-	-
/var/spool/mail	read	18	(40.91)	75	(91.46)
	write	4	(09.09)	7	(08.54)
	null	22	(50.00)	-	-
/usr/local	read	146	(100.00)	100	(100.00)
/usr2	read	731	(74.52)	686	(68.12)
	write	199	(20.29)	321	(31.88)
	null	51	(05.20)	-	-
/packages	read	9	(100.00)	9	(100.00)

Table 15: A breakdown of the open-close sessions on each file system by type of open-close session. Values in parentheses are each type of operation as a percentage of the open-close sessions on that file system.

long periods. The result is that `nfstrace` is able to give a better result for open-close sessions because the NFS transactions for this file system were more complete.

Because the cache is removing the need for a large number of the read operations to result in NFS transactions, the read:write ratio is closer to unity for the results of `nfstrace` than the results of `snooper`. While this ratio is expected, even desirable, for the measurements of data transferred, these values are incorrect for open-close sessions, resulting in higher average data transferred per session and incorrect information about the characteristics of the sessions.

However, while the ratios of the various types of open-close sessions produced by `nfstrace` are not particularly close to those of recorded by `snooper`, adding the figures for null sessions to the read open-close sessions improves the comparison for all file systems except for `/`.

For Table 16, all write values are increased by `nfstrace`, particularly in the case of `/usr2`. This error will partly be because `nfstrace` interprets the creation of any file and any subsequent writing to that file as two separate write events. Additionally, `nfstrace` can incorrectly interpret multiple writes to the same file as consecutive open-close sessions. Because `nfstrace` interprets an access to the first byte of a file as the start of a new open-close session, `nfstrace` can interpret multiple writes into the same location in a file as multiple open-close sessions on that file. As an example, this situation can arise with the `vi` editor [30]. `vi` uses log files that check-point the edit operations as they occur on the file, so `vi` can be continually writing small changes to the log file. These collections of small writes will result in blocks being written to the server and if there are a number of writes made to the first block, the first block may be written to the server several times. Each time the first block is written to the server `nfstrace` could potentially misinterpret the writing of data as separate open-close sessions on the log file. It is worth noting that the actual number of extra sessions is quite small and in comparison with values for all open-close sessions, will be overwhelmed by the quantity of other open-close sessions (read sessions in particular). However, for open-close sessions writing to a file, these extra open-close sessions can be significant.

Some of these problems are as a result of the algorithms used by `nfstrace`. While some assumptions have been made by `nfstrace` so as to produce an open-close session record, this particular situation may be resolved with a more sophisticated `nfstrace` algorithm.

The duration of an open-close session is important in determining the amount of time a particular file is in use. This, in turn, is important in calculating the amount of time files are shared between users and, in a distributed file system, between systems. Figure 23 shows that duration of open-close sessions recorded by `rpcspy` will be longer than those recorded by `snooper`. The longer open-close sessions that cause the differences in average durations are likely to be a result of transactions that are part of separate open-close sessions being interpreted as part of the same open-close session.

Additionally, the calculation of duration from NFS traffic means that lead and lag times (times in which the file is open but no operation occurs) will be different from the average length of the open-close session. These situations are represented graphically in Figure 19, these figures show that the block operations upon which `nfstrace`'s record will be based may not correspond with the logical open and close operations in

File System	session type	snooper		nfstrace	
read	/	6415	(86.20)	818	(46.77)
	/usr	123	(01.65)	61	(03.49)
	/var/spool/mail	18	(00.24)	75	(04.29)
	/usr/local	146	(01.96)	100	(05.72)
	/usr2	731	(09.82)	686	(39.22)
	/packages	9	(00.12)	9	(00.51)
write	/	354	(63.55)	476	(59.20)
	/var/spool/mail	4	(00.72)	7	(00.87)
	/usr2	199	(35.73)	321	(39.93)
read-write	/	35	(100.00)	-	-
null	/	298	(71.63)	-	-
	/usr	45	(10.82)	-	-
	/var/spool/mail	22	(05.29)	-	-
	/usr2	51	(12.26)	-	-

Table 16: A breakdown of the open-close sessions of each type, breakdown is by the file system of the file. Values in parentheses are each file system's operations as a percentage of the open-close sessions of that type.

an open-close session.

Figure 24 graphs a comparison of the data-transfer rate as measured by `snooper`, as per Figure 21, with the amount of data `nfstrace` estimates was potentially available to the system, (an accumulation of the sizes of files accessed). While not directly comparable, it is worth noting that the accumulation of the sizes of files is able to give enough information to estimate with fair accuracy the trends of data transfer between client and server.

Figure 25 shows a cumulative distribution of open-close sessions versus the amount of data transferred. It is important to note that one reason that `nfstrace` differs so significantly with `snooper` is that `nfstrace` was unable to detect the large percentage of open-close sessions during which approximately 1 Kbyte was transferred. Additionally, `snooper` results estimate that fewer than 500 of the open-close sessions transferred one or zero bytes, whereas `nfstrace` results estimate those circumstances existed for more than 1,000 of the sessions it recorded.

A primary reason `nfstrace` does not record the large number of sessions transferring approximately 80, 750, 900 and 1,100 bytes is because those files are in the cache and no data is transferred between server and client. This reason is strengthened by the

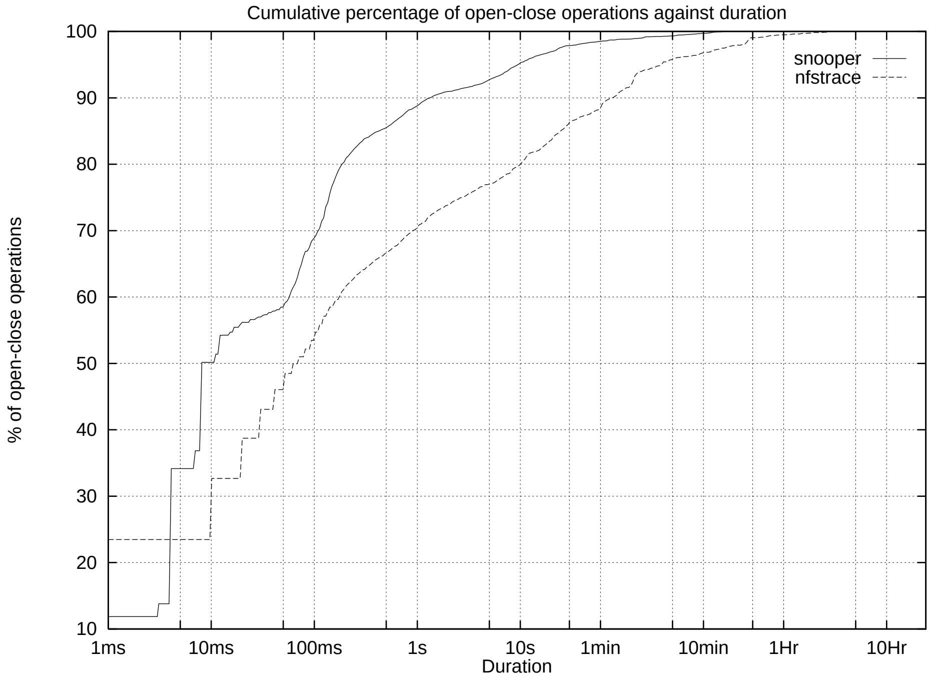

Figure 23: Normalised cumulative distribution of the number of open-close sessions versus the duration. From this graph we can deduce the longest of the open-close sessions for a given number of those sessions. For example, the `snooper` technique records that 70% of the sessions have a duration of about 100 milliseconds or less. *Note:* the duration axis is logarithmic.

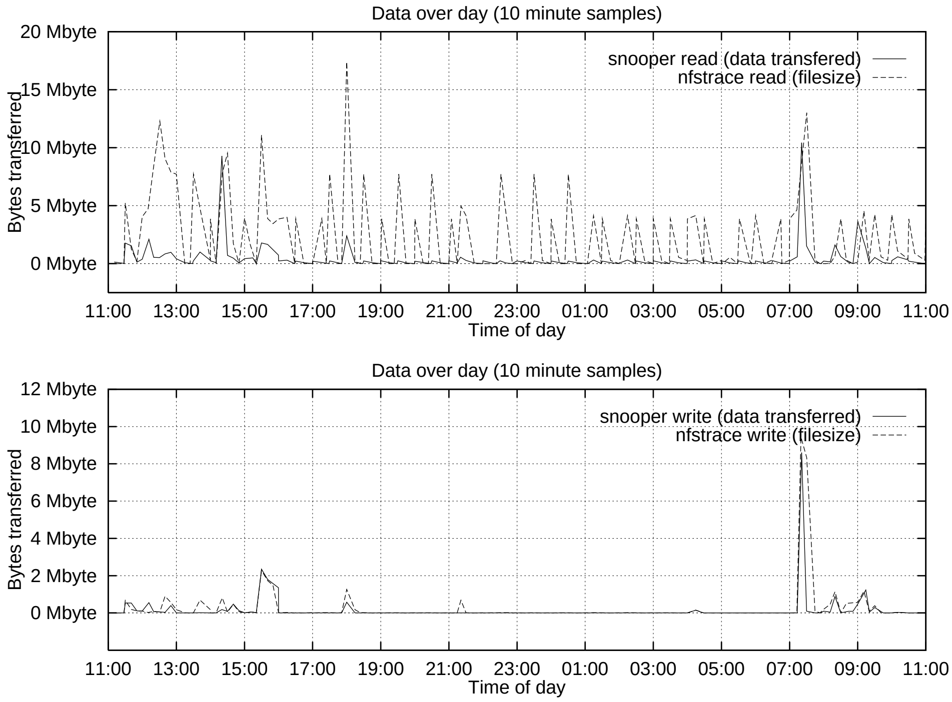

Figure 24: These graphs compare the transfer rate measured with `snooper`, to the total amount of data `nfstrace` has calculated the client has had access to in each file from which it has read data. As a low-order approximation, these values are comparable giving the same characteristics for data utilisation over time of the trace.

files being of small size and stay in the cache, and that `nfstrace` gives trends similar to those of `snooper` for other transfer values (even if the actual number of sessions is greatly reduced).

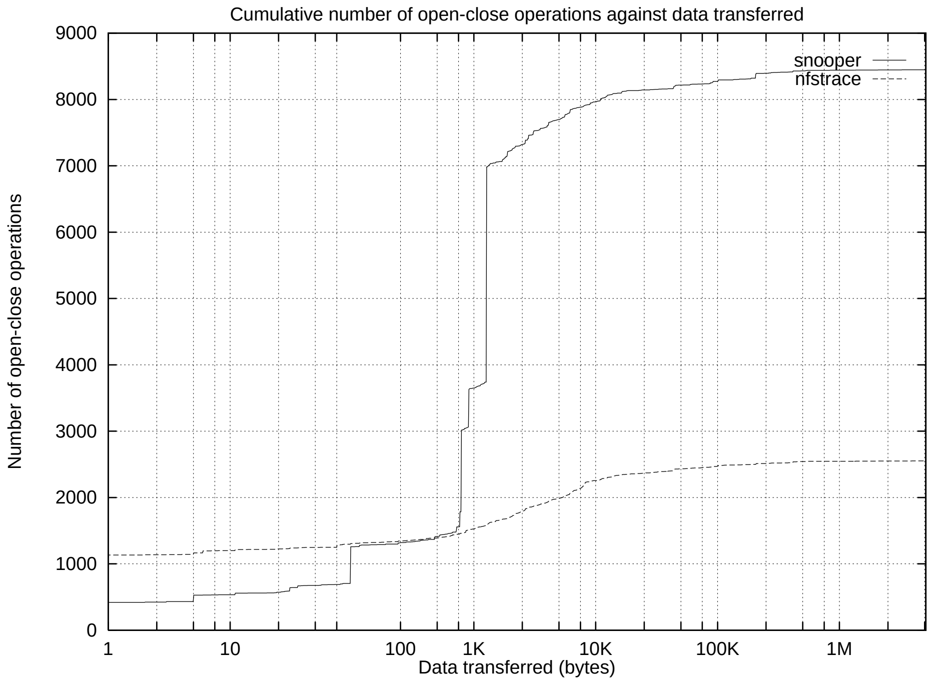

Figure 25: Cumulative distribution of the number of open-close sessions versus the data transferred for each open-close session. From this graph we can deduce the amount of data transferred per open-close session for a given number of those sessions. For example, the `snooper` technique records that over 7,000 sessions transfer about 1,100 bytes of data. *Note:* the data transferred axis is logarithmic.

The differences between `snooper` and `nfstrace` in Figure 26 have resulted from `nfstrace` being unable to interpret frequent accesses to files of a certain length, in particular, files which are 80, 750, 900 and 1,100 bytes in size. Accesses of such files account for a large percentage of the overall open-close sessions for regularly-accessed files but `nfstrace` is not recording an open-close session for them. This results in an exaggeration in the graphs for the number of open-close sessions for common data-transfer and file-size values. This situation is probably exacerbated by the inability of `nfstrace` to record many of the open-close sessions in which no data transfer is made.

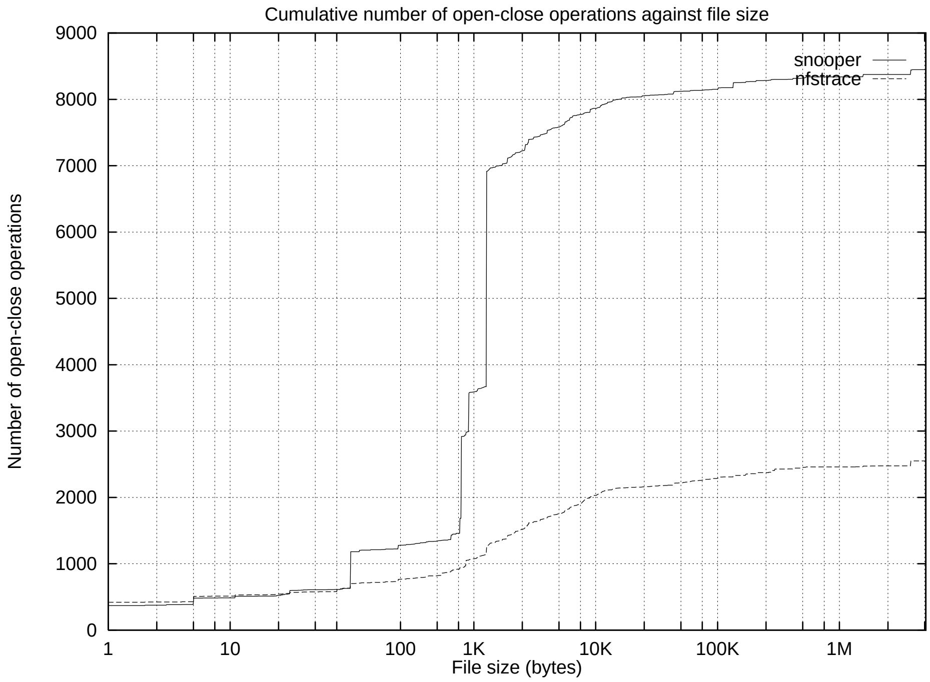

Figure 26: Cumulative distribution of the number of open-close sessions versus the size of the file accessed in each open-close session. From this graph we can deduce the maximum size of files opened for each open-close session for a given number of those sessions. For example, the `snooper` technique records that over 7,000 of the sessions access files containing less than 1,100 bytes of data. *Note:* the file size axis is logarithmic.

	NFS transactions (%)		payload data (bytes) (%)		Average payload data NFS transaction
Reads	10,387	(61.73)	26,502,065	(51.87)	2,681
Writes	6,129	(38.27)	24,591,581	(48.13)	4,012

Table 17: The above values are a count of all NFS read or write transactions as well as all the data transferred in those transactions to or from the file server.

6.7 Results from `rpcspy/nfstrace` only

The combination of `rpcspy` and `nfstrace` may not currently be able to give accurate information on open-close sessions and other logical levels of operations. However, both these tools, particularly `rpcspy`, are able to give useful information about the NFS protocol, the utilisation of the network by a particular client, and server loads.

While a small amount of this information may be available from a particular machine, the most-accurate results are recorded from the network itself.

6.7.1 Network

In Table 17 a breakdown of the number of NFS transactions, data carried by those transactions and the average amount of data per transaction are given. This set of results is, like those presented previously, without executable file traffic. The very high figure for the average amount of data per write transaction should be noted. It results from write operations being performed in block-size unit or as close to block-size units as possible.

In these results the amount of data read and written for a single client is almost the same. This would lead to a conclusion that the performance of the client would be as dependent on the speed of write operations as on read operations. This is not the case in logical operations where the amount of data read commonly outstrips the amount of data written to a file system by a factor of about 2.5 [73, 8].

A breakdown of the NFS transactions, as is given in Table 18, shows the type of utilisation of the network by the client as well as highlighting potential problems. From the table it is interesting to note that over 30% of the NFS transactions performed by the client were `getattr`. While these transactions are not used exclusively for cache consistency, the quantity of the transactions far outweighs any other. This would imply that in comparison to `read` transactions, which account for only 6.8% of the total number of transactions, there are over four times as many operations to check on cache consistency than transactions that cause data to be placed into the cache.

In this particular breakdown a potential problem is highlighted. The number of `readlink` transactions which are used to resolve symbolic links, is over 15% of the total. According to an NFS performance guide produced by Sun Microsystems [105], such a high figure may be adversely affecting performance, and reducing the number of symbolic links will potentially improve performance.

In this way, information about the functioning of the NFS protocol can give information both about areas where any development would have best impact. In this example, with a large utilisation of `getattr`, small improvements in its performance will give potentially dramatic overall performance increases. Additionally, such intrinsic information can indicate potentially troublesome operational areas, in this case the high percentage of `readlink` transactions.

With `rpcspy` there is potential for this technique to be used to collect information about all machines communicating on a particular computer network. For example, complete information about a file-server's NFS transactions could be collected from the communications traffic of that particular server. Table 19 gives results similar to those of Table 11 (results for a single machine), with the exception that these results have been collected from all communications traffic with a particular server. This table has two parts. The first part is based on the data exchanged between client and server. While the second part is based on the NFS transactions exchanged with the server. This table also provides an additional breakdown between both clients and users. The results in this table have not had executable traffic removed, and thus it is not directly comparable with results previously given. It is given as an indication of the unique abilities of network-based monitoring.

Another difference between NFS transactions that carry data, `read` and `write` and those that do not, such as `getattr`, is also given in Table 19. The number of client

NFS transaction	Total number	(% of total)
getattr	45,902	(31.55)
lookup	27,214	(18.71)
readlink	22,159	(15.23)
statfs	12,861	(08.84)
readdir	10,109	(06.95)
read	9,886	(06.80)
write	6,129	(04.21)
remove	4,924	(03.38)
setattr	3,787	(02.60)
mkdir	693	(00.48)
rmdir	687	(00.47)
rename	434	(00.30)
create	352	(00.24)
null	339	(00.23)
link	11	(00.01)
root	0	(00.00)
writecache	0	(00.00)
Total	147618	-

Table 18: A breakdown of all NFS transactions to the file server.

machines and the number of users, transferring data to and from the server, differs significantly from the number of different clients and users performing one or more NFS transactions. This difference could result from a number of sources. Firstly, clients may not need to collect data from the server if a local cache copy is available, as a result the NFS transaction will be a `getattr`, transferring no data, the result is that users on these machines do not need to retrieve data from the server that is available in the client cache. The other reason for the difference could be that NFS transactions will be triggered from clients when users do operations on the disk that does not involve retrieving data. An example of this might be getting a listing of a directory on the NFS server.

6.8 Results from `snooper` only

By virtue of `snooper` being a set of modifications to the kernel, there is potential for `snooper` to collect any information about the operation of the kernel. This can include information about processes, an aspect of machines that `nfstrace` will not be able to

From data transferred	
Total number of users	41
Total number of clients	44
Maximum amount of data transferred by users	339,668,438
Maximum amount of data transferred by clients	379,519,630
Average amount of data transferred by users	19,626,049
Average amount of data transferred by clients	18,287,910
From NFS transactions	
Total number of users	101
Total number of clients	137
Maximum number of transactions by users	618,891
Maximum number of transactions by clients	317,547
Average number of transactions by users	10394.0
Average number of transactions by clients	7662.7

Table 19: A comparison of active users, maximum and average data transfers determined using both data transferred and NFS transactions. Results are further broken down on the basis of per-user and per-client.

access as information describing the source process does not become part of the NFS transactions.

6.8.1 Process information

The execution of a process goes through two distinct phases. In the first phase, the process is created, using the `fork` system call, and the program to be executed is loaded into memory. In the second phase, control is transferred to the new program using the `exec` system call and finally the process will terminate with an `exit` system call.

Table 20 gives average times for each of these phases as well as the average time taken between `fork` and `exit`, that is, the total lifetime of a particular process. It has been noted that UNIX processes are, on average, short-lived [35, 52]. This is partially because process creation has a low overhead and as a result processes have become a resource that is highly used by operating systems developers.

Figure 27 shows a cumulative distribution of the time taken in each phase of the process's life. It is noteworthy that while more than 95% of processes have finished the `fork`-`exec` cycle in less than half a second, 3% of processes take 10 seconds or longer to have finished this phase. Also, 90% of processes take greater than 200 milliseconds

	`fork` to `exec`	`exec` to `exit`	`fork` to `exit`
Average time in seconds	0.37	19.80	20.18
Maximum time in seconds	53.86	5,810.40	5,810.51

Table 20: The average time for phases of a process's lifetime. The `fork` to `exec` time covers the loading of a program. The `exec` to `exit` time covers the execution of the program. The `fork` to `exit` times given are the total time taken for a process.

from `fork` to `exit`, yet 90% of processes take less than 10 seconds over this same time.

If we assume that open-close sessions of files are not restricted to processes with any particular characteristics in regard to process lifetime, and that open-close sessions that extend across two or more processes are relatively uncommon, then this process life-limit would indicate the open-close session of a file was unlikely to be longer than 10 seconds in duration in 90% of cases. Such a result confirms that `nfstrace` may be estimating incorrectly the occurrence and length of open-close sessions and, as a result, will have the potential to incorrectly estimate the amount of data transferred for each open-close session.

While the file traffic for programs executed was not included in these results, it would indicate, from this result, that if the average open-close session (including file traffic for executables) was included, the average duration would be closer to 10 seconds.

6.9 Summary

The preceding results show that, while the two set of results are not directly comparable, `nfstrace` is able to make a first order approximation of number of values traditionally measured by systems such as `snooper`, such as the total I/O transferred by a machine or the quantity of data written. Additionally, other estimated values, while imprecisely estimated by `nfstrace` in the current version, could potentially give accurate enough results to be able to replace systems such as `snooper` outright in a number of circumstances, such as measuring the number of active users per machine or the distribution of file size compared with files accessed. Most discrepancies in the interpretation by `nfstrace` when compared with results from `snooper` relate to the identification of open-close sessions. Minimisation of these errors would improve the

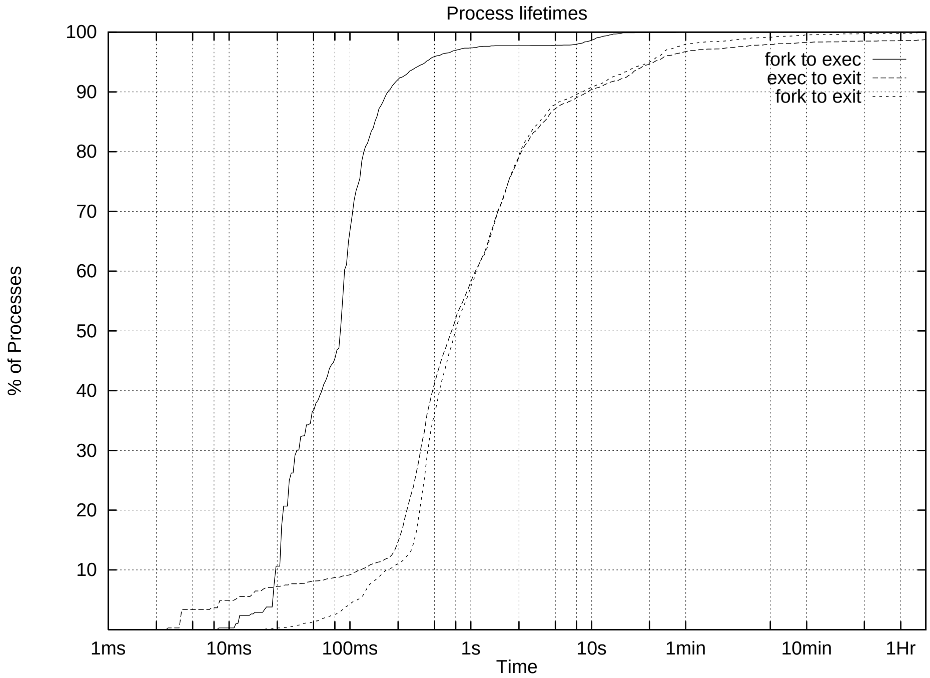

Figure 27: Cumulative distribution of the stages of process-life versus the time taken in each stage. Most processes have two stages, the time taken between the `fork` and `exec` system call when the executable is loaded, and the time taken between the `exec` and `exit` system call when the executable is run. The time given as `fork` to `exit` is the total amount of time taken for each process. From this graph we can deduce the maximum time taken for an execution stage for a given percentage of processes. For example these graphs show that about 65% of the fork-exec times of processes take less than 100 milliseconds. *Note:* the time axis is logarithmic.

estimation of both open-close session duration and data-size results.

A number of the results collected by `nfstrace` are not comparable with those collected by `snooper`, for example the amount of data transferred. While values for the maximum data transferred and write operations can be compared, values affected by significant caching (e.g. reading of data, particularly small amounts of data repeatedly from the same file), will differ significantly.

In addition to measures which can be compared, the unique nature of both `snooper` and `rpcspy` means that each has access to different types of information. `Snooper`, as a piece of kernel instrumentation, is ideally suited to record information about processes, an area from which network monitors are unable to retrieve information. In comparison `nfstrace` is ideally suited to collecting information about all machines on a particular network, including for example, all the traffic to a particular for server. These differences mean each technique has a role to fulfil, but there is certainly potential for network monitoring to be able to make measurements for which kernel instrumentation has traditionally been used in the past.

Additionally, it is worth pointing out that the information `rpcspy` generates and that `nfstrace` in turn uses, is not in error. The differences between `nfstrace` output and that of `snooper` occur because `nfstrace` attempts to estimate the operations on the user side of the cache, from the operations that occur on the file system side of the cache. Improvements in the performance of `nfstrace` would come about from improvements in this estimation process.

6.9.1 `rpcspy/nfstrace` problems

For `nfstrace` to be a more useful tool, the accuracy of its estimations needs to be improved. There are a number of areas where `nfstrace` either makes errors or does not have enough information with which to work.

Areas in which `nfstrace` can potentially be improved:

1. `nfstrace` incorrectly interprets the creation of a file to be two separate open-close sessions.

2. `nfstrace` does not always correctly interpret `getattr` transactions. The result of this is an overestimation of cached reads and an underestimation of open-close sessions with no data transfer.

3. the method `nfstrace` uses for summing transfers together can result in spuriously missing `read` or `write` transactions.

To a large extent these problems, particularly 2 and 3, are also a result of NFS not making enough information available for `nfstrace` to be able to estimate the operations that are occurring. The lack of data supplied by NFS also means `nfstrace` acts as a filter removing short, consecutive, open-close read sessions. Such operations are absorbed by the cache and as a result fine-grain sporadic operations are missed.

During this study, the recording of all Ethernet traffic by the `rpcspy` machine was not possible (a loss of 1.5% was recorded). This implies a loss of 0.6% of the total NFS transactions from the recorded trace, if we assume a ratio of NFS to non-NFS traffic at the same ratio as was recorded during the testing of `rpcspy` network packet capture mechanism (Section 5.4.1). While a source of potential error, this data loss is overshadowed by the error introducing aspects of the operation of `nfstrace`. As a result, while this error should not be discounted, it can be considered to have low overall significance in the results.

Chapter 7

Improving passive network monitoring

In this chapter the discrepancies between results gained with `rpcspy` and `nfstrace` and those gained by `snooper` (as shown in Chapter 6) are discussed. This chapter also includes a discussion of methods by which the discrepancies can be reduced.

There are two distinct levels at which this suite of software operates: `rpcspy` collects NFS transactions from the network and `nfstrace` interprets the NFS-transaction trace of `rpcspy` and generates a trace of open-close session records. Each of these two levels is discussed separately.

7.1 Improving `rpcspy`

7.1.1 Limitations of `rpcspy`

`rpcspy` must be able to collect and pair enough NFS-transaction requests and replies to enable accurate interpretation. Ideally, `rpcspy` should be able to record every NFS transaction, but its ability to do so depends also on the Ethernet packet capture mechanism of the machine on which it is operating. As was shown in Section 5.4.1, Figure 16, neither of the Ethernet packet capture facilities `rpcspy` can use (the `NIT` mechanism of SunOS and the `packetfilter` mechanism of Ultrix) are able to capture every Ethernet packet beyond a particular level of Ethernet utilisation. In addition to this, `rpcspy` has limits on the number of NFS transactions it can handle in a given time because it has a significant amount of processing to perform in the matching of each transaction.

The tapering effect of this processing is shown in Table 17.

As a result of these limitations, the performance of `rpcspy` is bound by the ability of the Ethernet capture mechanism to collect all packets on the Ethernet network at a given utilisation-level and on the performance of the `rpcspy` machine to process the NFS transactions.

7.1.2 Improvements to `rpcspy`

In Section 5.4.1, a comparison revealed that the `NIT` Ethernet facility of SunOS offers limited configurability as well as a higher packet loss, while the `packetfilter` facility of Ultrix offers better configurability and higher performance (fewer lost packets). The configurability of `packetfilter`, in particular the ability to increase the size of filter buffers, gave a much better loss-characteristic for this system than the `NIT` based implementation of `rpcspy`. Because `rpcspy` depends so critically on the characteristics of the Ethernet packet capture mechanism facility for the equipment compared, the selection of `packetfilter` is almost unavoidable.

Obtaining optimum performance from `rpcspy` also involves using a machine that does not have other significant duties that would detract from its ability to process NFS transaction information. For example, running `rpcspy` on the NFS server or a workstation with a heavy workload would not give optimum performance results. Ideally, the workstation used should be one dedicated to the task of collecting `rpcspy` data, if only for the period of the trace. Additionally, it is a reasonable assumption that the more powerful the workstation, the greater its ability to process NFS transactions.

While an `rpcspy` configuration was not attempted on the newer workstations from Digital or Sun Microsystems, both the improved performance of the workstations and Ethernet capture mechanism would suggest potential improvement in the operation of `rpcspy` on such machines.

7.2 Limitations of `nfstrace`

Chapter 6 showed discrepancies between results of `nfstrace` and those collected by the `snooper` kernel-instrumentation system. These discrepancies, listed in Section 6.9.1, are a result of the lack of information available to `nfstrace` and of sometimes incorrect interpretation of this information by `nfstrace`. The misinterpretation is caused by simplifications in the rule-base used by `nfstrace`, listed in Section 5.3.4, and inappropriate `nfstrace` parameters.

In `nfstrace`, a complicated relationship exists between each rule in the rule-base it uses. As a result, the solution to a number of problems with `nfstrace` would simultaneously solve or, in certain cases, complicate other observed discrepancies. The listing below shows each major discrepancy or problem with `nfstrace`.

1. `nfstrace` treats the creation of a file as two separate open-close sessions.
2. Underestimation of the number of open-close sessions. This also means `nfstrace` can overestimate the data transferred per open-close session, particularly in the case of writes.
3. `nfstrace` is unable to observe logical data transfer.
4. `nfstrace` has no record of open-close sessions that transfer no data at the logical level.
5. `nfstrace` has no record of open-close sessions that both read and write data.
6. The `nfstrace` method used for summation of read operations and write operations can result in transferred data not being counted.
7. The method used for estimating the purpose of an NFS `getattr` transaction is simplistic.
8. `nfstrace` does not estimate the contents of a client cache. As a result `nfstrace` will assume files in cache are being accessed when this is not the case.
9. `nfstrace` is unable to detect short open-close sessions.

7.3 Improvements to nfstrace

In this section we will outline methods for improvements in each of the areas listed in the previous section are outlined.

7.3.1 nfstrace treats the creation of a file as two separate open-close sessions

This misinterpretation arises because `nfstrace` does not interpret NFS `create` transactions at all. With the current algorithm, `nfstrace` processes the NFS `getattr` transaction (a by-product of file creation used to get the attributes of the new file into the client) and assumes that an open-close session involving a cache copy of the file is taking place. `nfstrace` then interprets the NFS `write` transactions that typically follow the creation of a file as a part of a different open-close session on that file.

While a modification to interpret the NFS `create` transaction would marginally increase the complexity of `nfstrace` algorithm, the change would mean the correct interpretation of the file-creation event. Such a change would also involve modifying the interpretation of the `getattr` transaction caused by the creation of the file and insuring, when appropriate, that operations writing data to the new file were also treated as part of the same file-creation open-close session.

7.3.2 Underestimation of the number of open-close sessions

While the central reason for this underestimation, a lack of information from the NFS transactions, can not be solved easily, `nfstrace` does not always correctly interpret the information it does have available.

Figure 28 illustrates the operation of the timeout in `rpcspy`. The timeout is used by `nfstrace` to determine when a record for an open-close session should be generated. If a new transaction occurs after a period greater than the length of the timeout since the previous transaction it will be treated as the first transaction of a new open-close session. In this way, the timeout-period represents the time between one open-close session and the next session, on the same file, on the same client, by the same user.

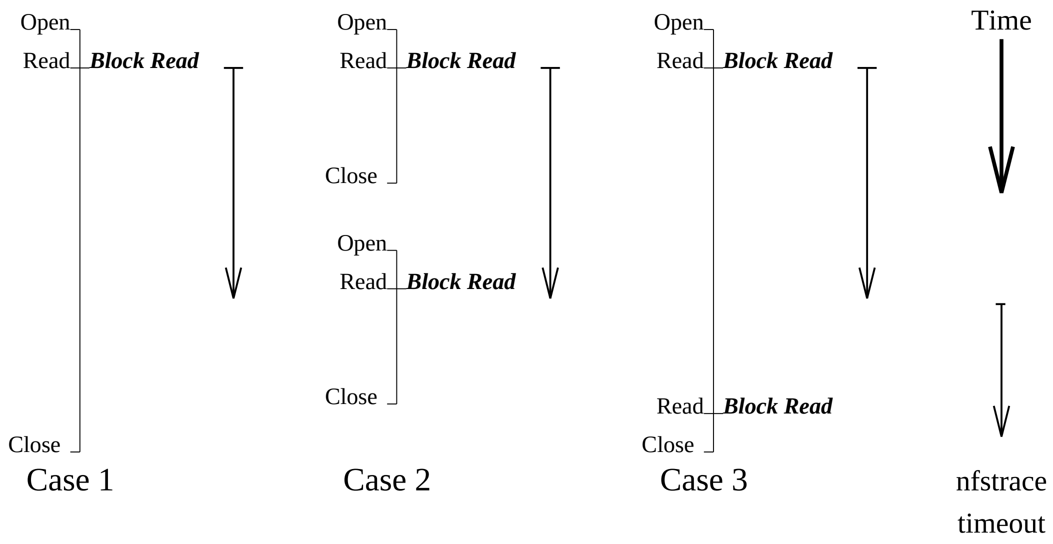

Figure 28: The operation of the `nfstrace` timeout in an open-close session. Case 1 shows the normal operation of the timeout, where an open-close session involves no additional NFS transactions. Case 2 shows the situation where too-long a timeout will cause the operations of two separate open-close sessions to be considered part of the same open-close session. Case 3 illustrates the situation where a timeout is not long enough and will typically cause the later NFS transactions to be considered as part of a new open-close session.

Because this value is adjustable and the behavior of `nfstrace` depends critically on the value of this timeout, the selection of this value is important. Unfortunately, the appropriate value for the timeout is not easily calculated. The `nfstrace` software uses a default value of 135 seconds (Blaze selected this value on the basis of his own experimentation), but it is not difficult to conceive of circumstances where the timeout might be inappropriate (in a network of significantly faster or slower machines for example).

Figure 29 is a plot of the number of open-close sessions recorded by `nfstrace` versus the timeout value. From this graph we can see that, under the current heuristic, no timeout value would enable `nfstrace` to match the total number of open-close session records generated by `snooper`. This graph also illustrates how characteristics of the underlying NFS system manifest themselves in the trace data. The significant steps at around 60 and 120 seconds are due to significant amounts of traffic, 60 and 120 seconds after blocks of a file have been read from the server. This traffic is most usually `getattr` transactions being sent to validate the contents of the cache. Additionally, the large step from zero seconds results from `nfstrace` assuming each transaction (such as a single NFS `read`, `write` or `getattr` transactions) as a single open-close session.

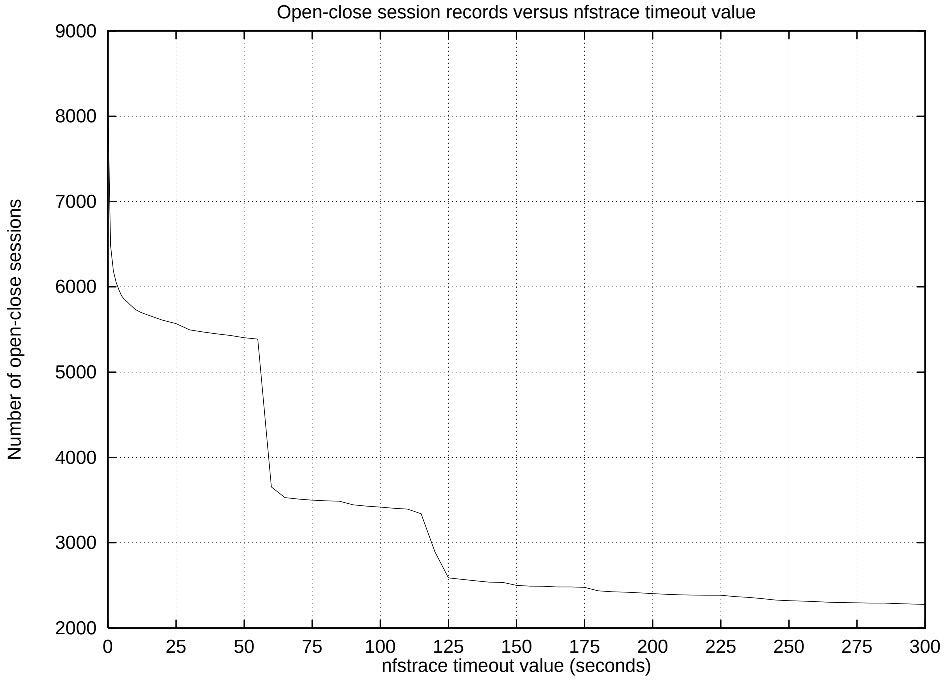

Figure 29: The number of open-close sessions recorded by `nfstrace` versus the value of the timeout `nfstrace` uses as part of its heuristic. `nfstrace` uses a default value of 135 seconds.

The number of open-close sessions is important because it affects values such the average size of files and the amount of data transferred over a given time. However, because data written to a file will almost always be seen as an NFS `read` transaction and is independent of number of open-close sessions, simply changing this timeout value to increase the number of open-close sessions could correct one of the values at the expense of another.

7.3.3 `nfstrace` is unable to observe logical data transfer

`nfstrace` cannot collect actual data about logical data transfers because NFS transactions do not contain this information. In the case of write operations, data are committed to disk when a file is closed and the amount of data logically written is nearly identical to the amount of data written to the file at the block level. However, the use of the client cache means that the amount of logical-read traffic does not have a one-to-one relationship with the data read at the block level. The difference is desirable because it improves the performance of the client machine and minimises the workload placed on the communications network and server machine. However, to get an accurate figure for logical file operations, a simple approach might be to modify the amount of data transferred for `read` operations by some multiplier. The results of Section 6.3 indicate that such a ratio would be around 2.8 : 1. This is broadly in agreement with other file system cache study results [73, 8, 98, 108, 14, 58], most of which consider the ratio 2.5 : 1 to be typical.

However, on closer inspection, the values in Table 10 show that while this average might be true for traffic taken as a whole, it is not so on a file-system-by-file-system basis. Using such a ratio as a multiplier is open to error because, as the results in Table 10 indicate, the different requirements of a file system influence the data transferred at the logical and block level for this file system as well as the ratio of the transferred amount. As a result, the use of a multiplier derived from this sort of ratio depends heavily on a similarity between the workload of the machine on which the ratio was derived and the machine on which this value is to be used.

Such a technique has the advantage of simplicity and, depending on the accuracy required, the use of a multiplier may be sufficient. However, it is worth noting that in addition to changes in workload, changes in the amount of memory available for caching can have a dramatic effect on the cache-hit ratio ([8, 73, 98]) and, because of this, such modified results could easily, if accidently, be of little worth.

Another approach is to estimate the amount of data in the cache. This would, at least, allow a user of `nfstrace` to estimate the amount of data that could have potentially been accessed. `nfstrace` makes such an estimation which, currently, is based on the size of the file thought to have been accessed. However, this is misleading because in many cases, such as those involving executable or large data files, the whole file might not be in the cache of the client computer. Furthermore, only part of the file might be accessed. Using the file size will give at least a rough estimate of the amount of data potentially available during an open-close session but will cause an overestimation of the amount of data accessed.

An improvement in the estimation of the amount of data in the cache of the client would be for `nfstrace` to incorporate a simulator of the cached data. In this way, `nfstrace` could make a more-accurate estimation of the amount of data potentially transferred in an open-close session. The incorporation of a block cache simulator into `nfstrace` would also assist in other areas where `nfstrace` reports results incorrectly or where the results show a great discrepancy with those of `snooper`, this concept is mentioned in more detail below.

7.3.4 `nfstrace` has no record of open-close sessions that transfer no data at the logical level

Because `nfstrace` has no access to logical data, open-close sessions that transfer no data will either appear as read open-close sessions with no data transferred or not appear at all. This problem has no easy solution. However, it would be solved partly by a better differentiation between programs such as `ls` causing the `stat` system call to invoke NFS `getattr` transactions and the `getattr` transactions being used to check the validity of the contents of cache or at the start of the `open` system call.

Blaze [11] found that the main cause of NFS `getattr` transactions was not related to cache-consistency or to the opening of files, but was the `ls` program itself. With this information `nfstrace` may be able to handle, as a special case, NFS `getattr` transactions that have occurred immediately following a request for the `ls` or `ls`-type programs rather than immediately assuming the NFS `getattr` transaction was as a result of a cache-consistency check. This method has a number of potential flaws and would involve pre-loading information about which programs were prone to generate `stat` system calls (and thus NFS `getattr` transactions) so that `nfstrace` could recognise them.

There is no easy way for `nfstrace` to detect with certainty any open-close sessions that have had no logical data transfer. With the use of a block cache simulator, `nfstrace` would at least be able to predict NFS `getattr` transactions that did not refer to blocks a client had in its cache. Additionally, combined with a better method of detecting spurious `stat` system calls, `nfstrace` may be able to better identify NFS operations as a result of open-close sessions and, thus, open-close sessions that have, at least potentially, no data transfer at the logical level.

7.3.5 `nfstrace` has no record of open-close sessions that transfer both read and write data

Because `nfstrace` has no access to logical data and a simple heuristic is used, open-close sessions that cause data to be both read from and written to a file appear either as a pair of open-close sessions (one for read operations and one for write operations) or an open-close session is generated only for the write operations. The reason for this second behavior, i.e. only one write open-close session for a file that has had data both read from and written to it, is that the heuristic used by `nfstrace` attempts to cope with the fact described in Section 6.6 that data transfers must be in block-sized units. Consequently, even if only part of a block is being written, the block to be modified must be read by the client before the modification can take place and the modified block must then be written back.

An approach of the version of `nfstrace` Dahlin et al. modified for their study ([24]) would be to record when a file is truncated (the file is explicitly set to zero length or the file has had no data transferred and the first operation is to write to the first byte in the file) and then, regardless of what data was read from the file, the open-close session would be treated as a write-only open-close session. In this way, the complex special handling of blocks of a file read before they are written would be more accurate and doubts about blocks read from the server being part of a write operation would be removed. In all other cases where data is both read from and written to a given file, `nfstrace` would consider the open-close session to be a read-write session. This solution appears to be a good method by which read-write sessions can be differentiated.

7.3.6 The `nfstrace` method used for summation of read operations and write operations can result in transferred data not being counted

As mentioned above, `nfstrace` does not have a concept of a file being both read from and written to during an open-close session. As a result, the heuristic `nfstrace` used to calculate when a file was written to could cause previous NFS `read` transactions, resulting from logical `reads`, to be included as part of the data transferred during the write-only open-close session.

In combination with the suggestion to differentiate read-write open-close sessions from write-only open-close sessions, `nfstrace` could be modified to record separately the data read from and the data written to the server. In this way, post-processing could enable the actual amount of data read or written to be determined instead of the current system where the amount of transferred data is summed.

7.3.7 The method used for estimating the purpose of an NFS `getattr` transaction is simplistic

As mentioned above, `nfstrace` uses a relatively-unsophisticated method to determine whether an NFS `getattr` transaction was used for cache validation or as a result of another operation such as a `stat` system call. `nfstrace` does not keep track of the contents of cache so it can make a prediction of whether an NFS `getattr` transaction would be the result of cache validation only by assuming that previously-read cache contents were being accessed. By incorporating a cache simulator, `nfstrace` would be able to predict with more certainty whether an NFS `getattr` transaction was part of a cache validation or as a result of another operation such as a `stat` system call.

`nfstrace` does combine the tracking of NFS `lookup` transactions, where a directory entry is translated into a particular NFS file handle, to assist in eliminating spurious NFS `getattr` transactions being handled as cache validation. This process is made possible because the NFS `lookup` transaction is commonly part of the sequence of calls when programs that display information about files (such as `ls`) are executed. However, the name-to-NFS-filehandle translation process, (causing the NFS `lookup` transaction) can result from many other causes. Theoretically, `nfstrace` could be excluding some NFS `getattr` transactions from being considered as having been caused by cache validation (such as at the opening of a file) because, as part of the opening of a file, the name of the file had to be resolved into an NFS file handle.

The incorporation of a cache simulator into `nfstrace` has the potential to help minimise the misidentification of NFS `getattr` transactions. This is a complicated change and would also involve `nfstrace` collecting information about other operations such as `readdir`, an operation which, like `lookup`, is used in the translation of name to NFS-filehandle.

7.3.8 `nfstrace` does not estimate the contents of a client cache.

As mentioned above, `nfstrace` assumes that whenever a cache access is made the whole of the file may have been transferred. Additionally, `nfstrace` may misinterpret NFS `getattr` transactions as being for files in cache when that file was never cached or when the cache entries had expired.

7.3.9 `nfstrace` is unable to detect short open-close sessions

Case 2 in Figure 28 illustrates a problem `nfstrace` may have when the operations of two separate open-close sessions are close enough together so that `nfstrace` interprets the operations to be part of a single open-close session. In the case where no NFS transactions are generated for the second open-close session (perhaps due to caching) there is no easy method by which `nfstrace` can be alerted to the second open-close session and there is no way that `nfstrace` could generate an open-close session record for it.

The other situation, where `nfstrace` interprets operations of the second open-close session as part of the first open-close session, may be avoided easily. Currently, `nfstrace` does attempt to do this. If the first byte of a file is (re)accessed, `nfstrace` considers that to be the start of another open-close session. In this way, if data is transferred as part of the second open-close session then `nfstrace` will potentially record these transfers as a second open-close session. An improvement to this method would be for `nfstrace` to include information on which blocks of a file have been accessed so that it could differentiate blocks accessed once from those accessed a second time. In this way `nfstrace` would interpret a second set of accesses to any blocks in a particular file as the beginning of a new open-close session on that file. The complication this method may introduce is where a file is not closed but the user starts to re-read previously read data (although previous studies show this is relatively uncommon) and instead a new open-close session will be started [73, 8].

Another case for `nfstrace` to handle is where the second open-close session causes a single NFS `getattr` transaction. `nfstrace` is not configured to interpret NFS `getattr` transactions as a special case. If the transaction occurs within the timeout period, as in case 3, it will be interpreted as being part of the first open-close session. If it occurs outside the timeout period, `nfstrace` will consider it the start of a new open-close session. It is unclear what the effect would be of changing `nfstrace` to interpret all such single NFS `getattr` transactions as the start of a new open-close session (independently of whether the timeout has ended). Such an assumption may potentially improve the interpretation of transactions by `nfstrace`.

These two modifications, combined with the cache simulator mentioned above, have the potential to improve the ability of `nfstrace` to record all open-close sessions, except the case where no NFS transactions occur over the duration of the whole open-close session. In the case of the Ultrix implementation of NFS, this is not as serious a limitation as it would first appear. The `open` system call in Ultrix NFS will always cause an NFS `getattr` transaction. This means that all open-close sessions will cause NFS transactions in the Ultrix implementation.

7.4 A block cache simulator for `nfstrace`

The prospect of a cache simulator improving the performance of `nfstrace` lead to the development of a proof-of-concept simulator. While not fully integrated with `nfstrace`, this software served to prove that such a concept was feasible and did not add overly to the complexity of the `nfstrace` system, while still giving the potential to improve the accuracy of `nfstrace` results, reducing the divergence between those and the results obtained by the kernel instrumentation system.

The cache simulator would be designed to supplement the `nfstrace` system and the existing `nfstrace` rule-base (Section 5.3.4). A cache simulator would enable `nfstrace` to be able to give accurate estimations of the data that may have been accessed in a particular open-close operation, as well as assisting `nfstrace` to be able to estimate better when open-close operations do and do not occur. In this way, the simulator would be driven with the NFS transactions that `nfstrace` uses, potentially by `nfstrace` itself.

Such a simulator could operate without explicit information about clients, using various preprogrammed sizing parameters. However, for accurate operation, the simulator should be preprogrammed with the size of caches of the clients and the number of blocks each client cache can contain. The simulator will not be completely accurate in its assumptions on what is in the client cache contents. This is because of a number of factors:

- the initial state of a client's cache is unknown,
- some file operations may not be visible as NFS transactions.

The first of these problems is addressed by starting the simulator in a known state. Using the trace of NFS records from the time a client is powered up would be a solution to this problem. The second problem is less easily solved and `nfstrace` suffers similar interpretation errors with small files (Section 5.4.4). However, a cache simulator has a greater ability to track the occurrence of read-only operations on any file, because the cache simulator can track blocks that are still valid in the client cache, thus reducing the magnitude of this error.

7.4.1 Block cache operation

The contents of a client cache are never explicitly removed. The removal process occurs because the client requires other data to be in the cache, with blocks being removed on the basis that these blocks have been the least-recently-used (LRU). From the perspective of the simulator, blocks would only be explicitly replaced following the occurrence of an NFS `read` `write` transaction. The decision on which block should be expired in the simulator can be based on which block has been least recently accessed (expired) and which block is oldest. Exact rules involving issues such as the replacement of cache block entries with data from NFS `write` transactions require further investigation, however the NFS `setattr` and `create` transactions could potentially assist in more accurate expiration of the cache for these block types. The NFS `getattr` and `setattr` transactions (in combination with other NFS transactions such as `create`) would cause the simulator to (re)validate the contents of the cache at any time.

7.4.2 A block cache simulator design

The proof-of-concept design was developed both as a supplement to increase the accuracy of `nfstrace` and as a trial to show the simulator is able to calculate such figures as file and block sharing in the distributed file system without substantial post-processing. This design was implemented in six files, about 3,000 lines, of C program code. This design was implemented as follows:

- A list of NFS filehandle references was maintained which in turn, referenced client entries. The client entries, in turn referenced all blocks of that particular file that was currently contained in each cache. This design made the revalidation of cache entries (on receipt of NFS `getattr` and `setattr` transactions) uncomplicated. Also, in this way the number of copies of each file in each client cache could be easily determined.

- Another list of clients was maintained, this in turn indexed the cached blocks, in order of least recent usage and age. This list enabled simplified location of the oldest cache blocks of a client so that these could then be replaced.

- An additional list of files, in turn referencing lists of the blocks that make up each file, enabled counts to be maintained for the occurrence of blocks of a file in each client's cache. In this way block sharing across clients could be easily established.

The trial simulator, while complete, requires extended testing against the kernel instrumentation system. In particular, such testing could validate the simulated contents of a client cache against the real contents of a client cache.

7.5 Summary

Improvements of `rpcspy` will be achieved by using a high speed machine with a high-speed, low-loss network interface to be dedicated to the task of data collection. The improvements to `nfstrace` can not be stated quite as concisely. Smaller changes to `nfstrace` include:

- adding the ability to interpret other significant NFS transactions such as `create`,
- using a simple ratio multiplier to obtain an estimate of data transfers at the logical level,
- modification of `nfstrace` to keep information about file truncation giving the ability to interpret file re-write events
- separately recording data read from and written to the server for all open-close sessions,
- recording information on which blocks of a file have been accessed, and
- interpreting NFS `getattr` transactions that immediately follow a file being read or written as another open-close session.

While some of these changes, such as the last item listed, would need to be tested to ensure the resulting extra records were correct, others in the list would give immediate improvement in the abilities of `nfstrace`.

More significant changes to `nfstrace` include

- pre-loading information about programs that cause `stat` system calls such as `ls`,
- build a block cache simulator into `nfstrace`.

In order to pre-load information about commonly-used programs that cause `stat` system calls, it may be necessary to profile the system prior to any significant tracing activity. In most systems, commonly-used programs such as `ls` could be expected to generate potential problems and could be added by default. However, the need to do a profiling operation would not only increase the complexity of passive network monitoring but might also negate any advantage of network monitoring by potentially requiring access to the machine being monitored. Another alternative, or addition, to pre-loaded configuration information is for `nfstrace` to characterise programs such as `ls` as it processes the NFS-transaction data. `nfstrace` would locate `ls` type programs by noting programs which, once executed, caused clusters of NFS `lookup` and `getattr` transactions, typically for files sharing the same sub-directory. In this way, `nfstrace` would be simultaneously processing the data and gaining enough information to locate programs causing extraneous NFS `getattr` transactions thus improving the prediction of `ls` type programs during the course of the run.

The incorporation of a block-cache simulator into `nfstrace` offers the best potential for increasing the accuracy of `nfstrace`. Unfortunately, several significant items of information would be needed to recreate accurately the block cache of a client. These would include the cache size on the client, the number of cache entries and the size of the data blocks being transferred between client and server. Additionally, the programming and testing of a cache simulator is not a simple task and because of resources used (memory, etc.) would potentially mean `nfstrace` could not be run simultaneously with `rpcspy` which is the recommended operating mode (in order to reduce output data).

The addition of the simulator would mean that `nfstrace` would be attempting to model a particular type of block cache. While there is a common ancestry for the method used by block caches in UNIX and its derivatives, there are notable differences. The introduction of such facilities as the demand-paging of executables, a facility noted in Section 4.3.6, means the behavior of the caches of systems being monitored will differ, sometimes dramatically. The result is that `nfstrace` may be required to incorporate models for several different block-cache systems. While this would add to the complexity of `nfstrace`, the common ancestry of block caches means much of the code used in each simulator would be common to all. It is conceivable that such an `nfstrace` could read a configuration file containing information on which cache method each client was using. Without appropriate configuration information, `nfstrace` could assume a particular model, perhaps the most common cache method used or the worst-case simulator model.

Such a pre-loaded configuration file would also contain information about NFS parameters such as cache and attribute timeouts, thereby assisting the accuracy of the simulator. This information, on a file-system by file-system basis could also give information about the characteristics of access to a file system, e.g. mail file systems can potentially cause open-close sessions to be generated when none was, and so on.

A block-cache simulator would increase the accuracy of the open-close session predictions `nfstrace` makes and allow `nfstrace` to be used for other purposes. `nfstrace` has the potential to simultaneously simulate the caches of all the machines on a network so it could be used to study interactions between the caches of different machines. For example, such a facility would enable a comprehensive study of block sharing among NFS clients.

An extension to `nfstrace` would enable it to keep track of information about the directory systems in a distributed file system. Modifications to directory information are written synchronously back to the server as the modifications take place, but the directory information itself is cached on the clients. Because changes to the directory information are written to the server synchronously, it is possible for `nfstrace` to maintain an accurate simulation of the state of the file system. Additionally, `nfstrace` could incorporate a directory-name cache simulator in the same style as a block-cache simulator and be able to simulate the contents of this cache among many clients. As in the case of a block-cache simulator, a directory-name cache would enable `nfstrace` to be used to study interactions between the caches of the clients and track the history of changes to the file system. The use of such a modification may enable a follow-up study to Shirriff and Ousterhout's work on name and attribute caching ([94]).

Many of the limitations in `nfstrace`, indeed, the very need for `nfstrace` to have to *estimate* open-close sessions, are caused by the fact that this information about open or close is not transmitted in the NFS protocol. Other distributed-system protocols, such as Sprite [72] and the Andrew File System [42], transmit information related to the state of files in the distributed file system. If `nfstrace` was modified to work with such a state-oriented distributed system, the accuracy of `nfstrace` output could potentially be as high as a full kernel instrumentation trace. The potential for accurate `rpcspy/nfstrace` analysis of distributed systems should also hold true for any distributed file system that transmits enough state information across the network. This method even has the potential to work on theoretical distributed file systems, such as xFS [120, 24], which depart from a central file server model completely. It is conceivable that during the development of such monitoring systems, methods based on the passive monitoring of network traffic would become a primary tool for assisting in the development and ultimately the management of such systems.

Another technique for increasing the accuracy of `nfstrace` is to add *simulated* state operations to NFS. This would involve modifying the kernel of each client to output extra NFS transactions for system calls such as `open`, `close` and `seek`. It would not be necessary for the server to act on or even acknowledge these calls, however the transmission of the extra information through the network would potentially give `nfstrace` enough information to be able to establish when files were opened and closed. Of course, such modifications are contrary to many of the concepts of passive network monitoring, requiring modifications to perhaps many client machines. However, this technique would maintain the benefit that the collection of the trace data would be independent of the server and clients. It would impose no extra workload directly upon them. This method of adding additional information to the communications traffic between client and server, for the purposes of monitoring, was used in Baker et al. [8] as one of a number of modifications they made to collect data for their work.

Distributed computer systems do not consist solely of distributed file systems. Systems such as Sprite [72], NOW [74, 118] and Amoeba [106] enable the migration of processes among CPU elements (typically a CPU element is a computer workstation). A monitoring method for such a system might involve monitoring the network's interconnecting processing elements and tracking the movement of the processes in the same way that `nfstrace` monitors the movement of file data among workstations. In this way, passive network monitoring has possible applications in areas other than just the monitoring of distributed file systems. Any system with significant amounts of information passing through an easily monitored communications network would lend itself to this technique. Other distributed systems that may lend themselves to monitoring in this way are the information services of the World Wide Web [119] or the network based windowing system, X [90].

Chapter 8

Conclusion

System monitoring is important in the development, refinement and operation of computer systems in general and of operating systems in particular. Chapter 3 described a number of methods of system monitoring and Section 3.3 illustrated how the results of system-monitoring studies were commonly used in other studies on topics such as cache simulation and user profiling. The results of such studies are used in the design and implementation of new computer systems as well as in the refinement of computer systems already in operation.

Chapter 4 discussed `snooper`, an implementation of full kernel instrumentation able to give detailed, exact, comprehensive trace information about a system being monitored. This exhaustive trace information is then processed into a record of file open-close sessions. Such a system is able to detail any aspect of the operation of the kernel such as logical or block-level file operations and details of processes. Because of the ability of full kernel instrumentation to present such a wide overview of the system, it has been the preferred method for system monitoring among system developers.

Chapter 5 described the `rpcspy/nfstrace` system. `rpcspy/nfstrace` is an implementation of a passive network-monitoring system which is able to generate an estimation of open-close sessions on files from NFS transactions exchanged between client and server as observed by a trace machine. While passive network monitoring is not able to replace full kernel instrumentation in every role, it can give useful first-order approximations and has a potential for its accuracy to be increased. Additionally, passive network monitoring offers independence of the monitored systems and the ability to monitor many machines simultaneously.

A comparison of these two system-monitoring methods was the theme of Chapter 6. In that chapter, results show that the passive network-monitoring implementation, while unable to give results comparable with full kernel instrumentation in all cases, was able to give good predictions of values derived from the full kernel instrumentation in certain areas. This was particularly true of those areas related to the writing of data. During this chapter we also established several areas of discrepancy between the results of `snooper` and `nfstrace`. The chapter then covered areas where each monitoring system was able to report information which was outside the capabilities of the other. Finally, the areas of discrepancy between `snooper` and `nfstrace` were summarised.

Chapter 7 covered comprehensively those areas where `rpcspy` and `nfstrace` had either errors or significant discrepancies when compared with the results of `snooper`. Solutions to each of the areas of discrepancy were then discussed. Additionally, it was noted that passive network monitoring had the potential to give more accurate information in a distributed file system with a greater amount of state information transmitted through the monitored network and to be able to monitor other network based systems.

8.1 Summary comments

In this thesis, system monitoring has been discussed as a significant part of the development of computer systems. A common method of monitoring systems is to use full kernel instrumentation, involving the modification of the source-code for the operating system of the machine. Passive network monitoring can be a preferred choice over kernel instrumentation for certain system monitoring work, particularly where the source-code of the operating system is not available. Additionally, other advantages of passive network monitoring can make it a preferred choice. These include:

- an independence of the collection of results from the machines being monitored on the network,

- the ability to simultaneously monitor multiple machines on a network, the passive network monitoring system requires no modifications to the operation of the monitored systems, and

- the collection of data with passive network monitoring does not impact on the machines being monitored.

Through the comparison of these two techniques, it is shown that passive network monitoring is satisfactory, as a partial replacement for full kernel instrumentation.

In addition to this, passive network monitoring is non-invasive, platform independent and has the ability to simultaneously monitor many network users. This gives it the potential for use in many systems studies using a broader cross-section of machines. Only through such a broad analysis can new systems be built based on information gained from more than just test systems and theories.

8.2 Future work

Ideally, future work would broaden the base over which the comparison of the two systems (Chapter 6) was made. The improvements could encompass both the inclusion of all traffic types, instead of the restriction to only non-executable file traffic, and the performing of the comparison on machines in a variety of operating circumstance. By comparing over a variety of systems, any peculiarities of the load the test system was placed under would be highlighted or, at least, minimised.

The logical extension of this work is the implementation of the suggestions in Section 7.3. These improvements would also need to be tested in a manner similar to the comparison of Chapter 6. For suggested improvements to `nfstrace`, it would be important to ensure changes did not alter the algorithm in unexpected ways.

Using a more accurate `nfstrace`, a comprehensive analysis in the style of Ousterhout et al. [73], Baker et al. [8], Howard et al. [42] and Spasojevic and Satyanarayanan [99] could be possible. Such an analysis would not only form an interesting comparison and contrast with those studies but also enable data to be collected from a variety of systems, rather than the traditional limitation to academic or research installations.

A comparison of `nfstrace` with a similarly-designed RPC transaction processor analysing other distributed file systems based upon RPC communications would give an interesting point of comparison between NFS and those systems.

The incorporation of a cache simulator into `nfstrace` would offer the potential for an increase in the accuracy of `nfstrace` estimations and the possibility for `nfstrace` to be used to perform other system studies directly without the need for any extensive results processing. Such a study could cover performance issues, while another study could be made into the sharing of files and blocks among clients. In the case of a performance study, the cache simulator could be used to establish relationships between block lifetimes and cache effectiveness with the size of caches and timeout characteristics of the NFS system.

As discussed in Section 7.5, a study into the utilisation of files and sub-directories, including lifetimes, usage distribution, etc., would also be possible with a suitably-enhanced `nfstrace` system. By combining such a modified `nfstrace` system with data about the file system before and after the trace period, it would be possible for `nfstrace` to accurately simulate and track operations on the directories of the file system. Such a facility would allow studies into file-naming structures and the caching of those structures in the style of Shirriff and Ousterhout [94].

Appendix A

Glossary

block In a file system, a block is a unit of allocation. The file system allocates space in block-size units, or in fragments of block-size units.

block cache/cache In operating systems, a block cache is an area of memory where commonly used disk blocks are stored. The objective of a block cache is to minimize a machine's need to access disk drives. In exchange, block caches will use real memory, making it unavailable to the operating system or other programs.

child process A process that is the direct descendent of another process which created it using the `fork` system call.

client A machine that requests services from a server. A client is usually unrelated to a server: the only association with the server is through a communications channel.

daemon A long-lived process that commonly provides a system-related service. A characteristic of such a process is that it has no controlling user terminal; i.e. it is a "background" process.

DMA/Direct Memory Access A hardware technique whereby the memory of a machine may be accessed without involving any activities on the part of the Central Processor Unit (CPU). This technique is commonly used for disk drives, so that disk drive controllers can place data directly in memory, enabling the CPU to continue doing other tasks.

dirty buffer blocks When a process changes the contents of a file, the cache will contain these modified versions of disk blocks, known as called dirty buffer blocks. The operating system must keep track of these blocks ensuring this data is ultimately written to the disk.

disk block A disk block is a unit of allocation for the disk. The file system is allocated in disk block-sized pieces or fragments of disk block size pieces. Commonly a disk block is the same size as the sector size of the physical disk media in use.

Ethernet A 10Mb/s baseband communication technology for the interconnection of computers in a local area network. Ethernet involves the use of a single broadcast cable connecting all machines in a local area network. To communicate with each other, machines send consecutive broadcasts onto the network for the receiving station to record.

exec A system call in the UNIX operating system allowing a process to execute a particular program. The exec system call has the effect of causing a process to replace its current code, stack and data memory with those of a new program; i.e. a new process is not created.

exit 1. The termination of a process in the UNIX operating system. 2. A system call in the UNIX operating system allowing a process to terminate itself.

file A file is a basic construction in the UNIX operating system. It is a linear array of bytes, it has at least one name (link) and it exists until all its names are deleted explicitly.

file system A collection of files. In the UNIX operating system, a file system is, in most cases, restricted to a single, physical, hardware device such as a disk drive.

filehandle A filehandle is an NFS construct by which an NFS server can uniquely identify each file it makes available to NFS clients. An NFS client does not need to decode the contents of a filehandle, a client only depends on it being a unique reference to the file on the NFS server.

fork A system call in the UNIX operating system allowing a process to create other processes.

full kernel instrumentation Obtaining data about a computer system through the placing of instrumentation code into the operating system (kernel) of the computer system.

kernel A kernel is the central controlling program that provides basic operating-system facilities. The UNIX kernel provides functions to access the file system, creates and manages processes, and supplies communications facilities.

ls A UNIX command to list and generate statistics for files including access times, ownership, etc.

mount 1. A UNIX command to make available a the data of file system from a particular point in a pre-existing file system. 2. The action of splicing together two separate file systems so that from the users perspective there appears to be one larger file system.

mount point A mount point is the position (directory) in a file system under which a mounted file system will be placed. For example, the `/usr` file system is mounted upon the directory `/usr`.

Network File System/NFS Sun's Network File System, a distributed file system which is a *de facto* standard in the UNIX computer community for the sharing of files between computers. Statelessness is central to the NFS mode of operation; i.e. servers do not store state information about clients such as which files a client has open. Additionally, all NFS transactions (between client and server) are self-contained and repeatable (idempotent).

NFS transaction An NFS transaction is a self-contained pair of RPC operations; a request and a reply.

NIT SunOS Ethernet packet filter. A software packet capture mechanism giving direct access to Ethernet traffic.

packetfilter Digital's Ethernet packet filter. A software packet capture mechanism giving direct access to Ethernet traffic.

parent process A process that is the direct ancestor of another process as a result of creating the second process with the `fork` system call.

passive network monitoring Obtaining data about one or more computer systems attached to a network by monitoring communications traffic exchanged by that computer with others through the network.

PID/process ID In the UNIX operating system, a process identifier. An identification of a process, unique for the lifetime of the process.

process A process is a basic construction in the UNIX operating system. In operating systems, a process is a task or thread of execution. Each process is identified independently and consists of program code, a data area and a stack area. These components may be shared with other processes.

ps A UNIX command to print process-status statistics.

RPC Remote Procedure Call. A specification by which a client can request that a service be performed by a server.

SCSI The Small Computer Systems Interconnect/Interface (SCSI) is an interface standard for connecting a computer and other devices via a fast, high-speed, parallel interface. This interface is commonly used for connecting CD-ROM players, disk drives and tape units to a computer system.

sector A sector is the smallest contiguous region on a disk that may be accessed with a single I/O operation.

server A machine that provides services to another machine (client) via a communications channel.

synchronous I/O The appearance of an operation being synchronized with the current process. In UNIX, `read` and `write` system calls are synchronous: The `read` and `write` system calls do not return until the operation has been completed. In the case of the `write` system call, the data may not actually be written to the final destination until some time later, for example, writing to a disk file.

system call A request by a user program to the operating system for some service to be carried out.

UID/User ID In the UNIX operating system, a numerical identification assigned to a user. Typically, such identifiers are assigned uniquely so that each user on a system is assigned one User ID (although this need not always be the case). User IDs are used in the control of resources such as file and directory access and process control.

UNIX An operating system developed originally at the AT&T research laboratories and finding popular usage in many research and educational establishments partly because of the availability of source-code.

XDR eXternal Data Representation. A specification for the exchange of data in a hardware-independent manner.

Bibliography

[1] Anderson, P. Effective Use of Local Workstation Disks in an NFS Network. In *USENIX LISA VI October 19-23, 1992* (October 1992), pp. 1–8.

[2] ANSI. *Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications.* IEEE, 1984. ANSI/IEEE Std 802.3 - 1985, ISO Draft International Standard 8802/3.

[3] ANSI. *Token-Passing Bus Access Method and Physical Layer Specifications.* IEEE, 1984. ANSI/IEEE Std 802.4 - 1985, ISO Draft International Standard 8802/4.

[4] ANSI. *Token Ring Access Method and Physical Layer Specifications.* IEEE, 1984. ANSI/IEEE Std 802.5 - 1985, ISO Draft International Standard 8802/5.

[5] Bach, M., and Gomes, R. Measuring File System Activity in the UNIX System. In *EUUG Spring '88* (London, UK, April 1988), pp. 43–52.

[6] Baker, M., Asami, S., Deprit, E., Ousterhout, J., and Seltzer, M. Non-Volatile Memory for Fast, Reliable File Systems. In *Proceedings 5th International Conference on Architectural Support for Programming Languages and Operating Systems, October '92* (October 1992).

[7] Baker, M., and Sullivan, M. Recovery box: Using fast recovery to provide high availability in the UNIX environment. In *USENIX Conference Proceedings, Summer 1992* (San Antonio, TX, June 1992), pp. 31–44.

[8] Baker, M. G., Hartman, J., Kupfer, M., Shirriff, K., and Ousterhout, J. Measurements of a Distributed File System. In *Proceedings of the 13th Symposium on Operating System Principles* (Pacific Grove, CA, October 1991), ACM, pp. 198–212.

[9] Baker, M. L. G. *Fast Crash Recovery in Distributed File Systems.* PhD thesis, University of California, Berkeley, 1994. Also available as UCB:CSD technical report UCB:CSD-94-787.

[10] Barnett, L., and Malloy, M. K. ILMON: A UNIX network monitoring facility. In *USENIX Conference Proceedings, Winter 1987* (Washington, D.C., 1987), pp. 133–144.

[11] Blaze, M. NFS Tracing by Passive Network Monitoring. In *USENIX Conference Proceedings, Winter 1992* (San Francisco, CA, January 1992), USENIX, pp. 333–344. Also available as a Technical Report with the Department of Computer Science, Princeton University.

[12] Blaze, M. `nfstrace` network monitoring tool, January 1992. Availablity anonymous ftp `ftp.uu.net:/networking/ip/nfs/nfstrace.shar.Z`.

[13] Blaze, M. *Caching in Large-Scale distributed file systems*. PhD thesis, Princeton University, January 1993.

[14] Blaze, M., and Alonso, R. Dynamic hierarchical caching in large-scale distributed file systems. In *USENIX Conference Proceedings, Summer 1991* (Nashville, TN, 1991), pp. 3–19. Also cs-tr-353-91, Computer Science Technical Report, Dept of Comp Sci, Princeton, NJ , Availability anonymous ftp `samadams.princeton.edu:~ftp/cstr/cs-tr-353-91.ps.Z`.

[15] Blaze, M., and Alonso, R. Long-Term Caching Strategies for Very Large Distributed File Systems. In *USENIX Conference Proceedings, Summer 1991* (Nashville, TN, 1991), pp. 3–16.

[16] Blaze, M., and Alonso, R. Issues in Massive-Scale Distributed File Systems. In *USENIX File System Workshop, May 21-22, 1992* (Ann Arbor, MI, 1992), pp. 135–136.

[17] Boggs, D. R., Mogul, J. C., and Kent, C. A. Measured Capacity of an Ethernet: Myths and Reality. Tech. Rep. 88/4, Digital Western Research Laboratory, April 1988.

[18] Bozman, G., Ghannad, H., and Weinberger, E. A trace-driven study of CMS file references. *IBM Journal of Research and Development 35*, 5/6 (September/November 1991), 815–828.

[19] Cao, P., Felten, E., and Li, K. Implementation and performance of application-controlled file caching. In *First USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Monterey, CA, November 1994), pp. 165–178.

[20] Carson, S., and Setia, S. Analysis of the periodic update write policy for disk cache. *IEEE Transactions on Software Engineering 18*, 1 (January 1992), 44–54.

[21] Carson, S., and Setia, S. Optimal Write Batch Size in Log-Structured File Systems. In *USENIX File System Workshop, May 21-22, 1992* (Ann Arbor, MI, 1992), pp. 79–92.

[22] Cheriton, D., and Mann, T. Decentralizing a Global Naming Service for Improved Performance and Fault Tolerance. *ACM Transactions on Computer Systems 7*, 2 (May 1989), 147–183.

[23] Clark, D. W., Bannon, P. J., and Keller, J. B. Measuring VAX8800 Performance with a Histogram Hardware Monitor. In *Proceedings of the 15th Annual International Symposium on Computer Architecture* (Honolulu, HI, May 1988).

[24] Dahlin, M. D., Mather, C. J., Wang, R. Y., Anderson, T. E., and Patterson, D. A. A quantitative analysis of cache policies for scalable network file systems. Tech. Rep. UCB:CSD-94-798, Department of Computer Science, University of California, Berkeley, February 1994. Also appeared in 1994 ACM SIGMETRICS Conference on Measurements and Modeling of Computer Systems, Nashville, TN, May, 1994, pp 150-160.

[25] Davies, N. A., and Nicol, J. R. Technological perspective on multimedia computing. *Computer Communications 14*, 5 (1991), 260–272.

[26] de Jonge, W., Kaashoek, M. F., and Hsieh, W. C. The Logical Disk: A New Approach to Improving File Systems. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles* (Asheville, NC, December 1993), pp. 15–28.

[27] Digital. *packetfilter - Ethernet packet filter*, Ultrix 4.3a User Manual ed., 1987.

[28] Digital. *rwhod(8c) - system status server*, Ultrix 4.3a User Manual ed., 1987.

[29] Digital. *inetd(8c) - internet service daemon*, Ultrix 4.3a User Manual ed., 1992.

[30] Digital. *vi(1) - screen editor*, Ultrix 4.3a User Manual ed., 1992.

[31] Douglis, F., Ousterhout, J. K., Kaashoek, M. F., and Tanenbaum, A. S. A Comparison of Two Distributed Systems: Amoeba and Sprite. *Computing Systems* (Autumn 1991), 353–384.

[32] Ebling, M. R., and Satyanarayanan, M. Synrgen: An extensible file reference generator. Tech. Rep. CMU-CS-94-119, School of Computer Science, Carnegie Mellon University, February 1994. Also appeared in 1994 ACM SIGMETRICS Conference on Measurements and Modeling of Computer Systems, Nashville, TN, May, 1994, pp 138-149.

[33] Emer, J. S., and Clark, D. W. A Characterization of Processor Performance in the VAX-11/780. In *Proceedings of the 11th Annual International Symposium on Computer Architecture* (Ann Arbor, MI, May 1984).

[34] Endo Y. et el. VINO: The 1994 Harvest. Tech. Rep. TR-34-94, Harvard University, Center for Research in Computing Technology, December 1994. Availability anonymous ftp `das-ftp.harvard.edu:/techreports/tr-34-94.ps.gz`.

[35] Feder, J. The Evolution of UNIX System Performance. *AT&T Bell Laboratories Technical Journal 63*, 8 (October 1984), 1791–1814.

[36] Floyd, R. Short-Term File Reference Patterns in a UNIX Environment. Tech. Rep. TR-177, Department of Computer Science, University of Rochester, March 1986.

[37] Floyd, R. A., and Ellis, C. S. Directory reference patterns in hierarchical file systems. *IEEE Transactions on Knowledge and Data Engineering 1*, 2 (June 1989), 238–247.

[38] Griffioen, J., and Appleton, R. Reducing file system latency using a predictive approach. In *USENIX Conference Proceedings, Summer 1994* (June 1994), pp. 197–207.

[39] Gusella, R. A Measurement Study of Diskless Workstation Traffic on Ethernet. *IEEE Transactions on Communications 38*, 9 (September 1990), 1557–1568.

[40] Hartman, J., and Ousterhout, J. Zebra: A Striped Network File System. In *USENIX Workshop on File Systems, May 1992* (May 1992), pp. 43–52.

[41] Hill M. D. et al. SPUR: A VLSI Multiprocessor. Tech. Rep. UCB-CSD-86-273, Department of Computer Science, University of California, Berkeley, April 1986.

[42] Howard, J., Kazar, M., Menees, S., Nichols, D., Satyanarayanan, M., Sidebotham, R., and West, M. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems 6*, 1 (February 1988), 51–81.

[43] Irlam, G. UNIX file size survey - 1993, April 18th, 1995. Available via the World Wide Web `http://www.base.com/gordoni/ufs93.html`.

[44] ITU. *ISO 9314 - Fibre Distributed Data Interface*. ISO, 1989.

[45] Jensen, D. W., and Reed, D. A. File archive activity in a supercomputer environment. Tech. Rep. UIUCDCS-R-91-1672, Department of Computer Science, University of Illinois at Urbana-Champaign, April 1991.

[46] Keith, B. Perspectives on NFS file server performance characterization. In *USENIX Conference Proceedings, Summer 1990* (Anaheim, CA, 1990), pp. 267–277.

[47] Kernighan, B., and Ritchie, D. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ, 1978.

[48] Kistler, J., and Satyanarayanan, M. Disconnected operation in the coda file system. *ACM Transactions on Computer Systems 10*, 1 (February 1992), 3–25.

[49] Kistler, J. J. *Disconnected Operation in a Distributed File System.* PhD thesis, School of Computer Science, Carnegie Mellon University, May 1993. Also available as CMU technical report CMU-CS-93-156.

[50] Kupfer, M. D. An Appraisal of the Instrumentation in Berkeley UNIX 4.2BSD. Tech. Rep. TR-CSD-85-246, Department of Computer Science, University of California, Berkeley, 1985.

[51] Kupfer, M. D. Performance of a remote instrumentation program. Tech. Rep. UCB-CSD-85-223, Department of Computer Science, University of California, Berkeley, 1985.

[52] Leffler, S. J., McKusick, M. K., Karels, M. J., and Quarterman, J. S. *The Design and Implementation of the 4.3BSD UNIX Operation System.* Addison Wesley, October 1990.

[53] Li, K. Towards A Low Power File System. Tech. Rep. UCB:CSD-94-814, Department of Computer Science, University of California, Berkeley, May 1994.

[54] Lyon, B. XDR : External Data Representation Standard, June 1987. Network Working Group Request For Comment (RFC) : 1014, Written in association with DARPA and Sun Microsystems Inc.

[55] Lyon, B. RPC : Remote Procedure Call Protocol Specification, April 1988. Network Working Group Request For Comment (RFC) : 1057, Written in association with DARPA and Sun Microsystems Inc.

[56] Macklem, R. Lessons Learned Tuning the 4.3BSD Reno Implementation of the NFS Protocol. In *USENIX Conference Proceedings, Winter 1991* (Dallas, TX, 1991), pp. 53–64.

[57] Macklem, R. *The 4.4BSD NFS Implmentation.* Computer Systems Research Group, University of California, Berkeley, 1993. from SMM:06-2.

[58] Makaroff, D., and Eager, D. Disk Cache Performance for Distributed Systems. In *IEEE 10th International Conference on Distributed Computing Systems* (1990), pp. 212–219.

[59] McGregor, A. J. PRIMON: The design and implmentation of a Primos Software Monitor. Master's thesis, Massey University, 1984.

[60] McKusick, M. K., Joy, W. N., Leffler, S. J., and Fabry, R. A Fast File System for UNIX. *ACM Transactions on Computer Systems 2*, 3 (August 1984), 181–197.

[61] Metcalfe, R., and Boggs, D. Ethernet: Distributed Packet Switching for Local Computer Networks. *Communications of the ACM 19*, 7 (July 1976), 395–404.

[62] Miller, E. L., and Katz, R. H. An Analysis of File Migration in a UNIX Supercomputing Environment. Tech. Rep. UCB-CSD-92-712, Department of Computer Science, University of California, Berkeley, November 1985.

[63] Mogul, J. C. Efficient Use of Workstations for Passive Monitoring of Local Area Networks. Tech. Rep. 90/5, Digital Western Research Laboratory, May 1990.

[64] Mogul, J. C. A better update policy. Tech. Rep. DEC-WRL-94/4, Digital Western Research Laboratory, April 1994. Also appeared in Summer USENIX Conference, Boston MA, June, 1994.

[65] Mogul, J. C., Rashid, R. F., and Accetta, M. J. The packet-filter: An efficient mechanism for user-level network code. In *Proceedings of the 11th Symposium on Operating Systems Principles* (Austin TX, November 1987), ACM SIGOPS.

[66] Morris, J., Satyanarayanan, M., Conner, M., Howard, J., Rosenthal, D., and Smith, F. D. ANDREW: A distributed personal computing environment. *Communications of the ACM 29*, 3 (March 1986), 184–201.

[67] Mummert, L., and Satyanarayanan, M. Long Term Distributed File Reference Tracing: Implementation and Experience. Tech. Rep. CMU-CS-94-213, School of Computer Science, Carnegie Mellon University, November 1994.

[68] Mummert, L., Wing, J., and Satyanarayanan, M. Using belief to reason about cache coherence. Tech. Rep. CMU-CS-94-151, School of Computer Science, Carnegie Mellon University, May 1994.

[69] Nelson, B., and Cheng, Y.-P. How and Why SCSI is Better than IPI for NFS. In *USENIX Conference Proceedings, Winter 1992* (San Francisco, CA, 1992), pp. 253–270.

[70] Nowicki, B. NFS : Network File System Protocol Specification, March 1985. Network Working Group Request For Comment (RFC) : 1094, Written in association with DARPA and Sun Microsystems Inc.

[71] Ousterhout, J. K. Why Aren't Operating Systems Getting Faster as Fast as Hardware. *USENIX Summer Conference June 11-15* (June 1990).

[72] Ousterhout, J. K., Cherenson, A. R., Douglis, F., Nelson, M., and Welch, B. The Sprite network operating system. *IEEE Computer 21*, 2 (February 1988), 23–36.

[73] Ousterhout, J. K., DaCosta, H., Harrison, D., Kunze, J., Kupfer, M., and Thompson, J. A trace-driven analysis of the UNIX 4.2 BSD file system. In *10th Symposium on Operating System Principles* (Orcas Island, WA, December 1985), ACM, pp. 15–24.

[74] Patterson, D. A Case for Networks of Workstations: NOW. Paper to appear in IEEE Micro. Presented at Hot Interconnects II and Principles of Distributed Computing, August 1994.

[75] Peterson, J. L., and Silberschatz, A. *Operating System Concepts*, world student series edition ed. Addison-Wesley, Reading, Massachusetts, 1985, ch. 9, pp. 332–333.

[76] Postel, J. Internet protocol, September 1981. Written in association with DARPA.

[77] Postel, J. Transmission Control Protocol, September 1981. Written in association with DARPA.

[78] Postel, J. Unreliable datagram protocol, September 1981. Written in association with DARPA.

[79] Ramakrishnan, K. K., and Emer, J. Performance analysis of mass storage service alternatives for distributed systems. *IEEE Transactions on Software Engineering 15*, 2 (February 1989), 120–133.

[80] Reddy, A. L. N., and Banerjee, P. An Evaluation of Multiple-Disk I/O Systems. *IEEE Transactions on Computers 38*, 12 (December 1989), 1680–1690.

[81] Ritchie, D. M., and Thompson, K. The UNIX time-sharing system. *Communications of the ACM 17*, 7 (July 1974), 365–375.

[82] Rosenblum, M. *The Design and Implementation of a Log Structured File System.* PhD thesis, University of California, Berkeley, 1992. Also available as UCB:CSD technical report UCB:CSD-92-696.

[83] Rosenblum, M., and Ousterhout, J. K. The LFS Storage Manager. In *USENIX Conference Proceedings, Summer 1990* (Anahien, CA, June 1990), pp. 31–44.

[84] Rosenblum, M., and Ousterhout, J. K. The Design and Implementation of a Log-Structured File System. In *Proceedings of the 13th Symposium on Operating System Principles* (July 1991).

[85] Ruemmler, C., and Wilkes, J. UNIX disk access patterns. Tech. Rep. HPL-92-152, Hewlett Packard Laboratories, December 1992. Also published in the USENIX Winter '93 Technical Conference Proceedings, San Diego, CA, Jan 25-29, 1993 pp 405-420.

[86] Ruemmler, C., and Wilkes, J. A trace-driven analysis of disk working set sizes. Tech. Rep. HPL-OSR-93-23, Operating Systems Research Department, Hewlett-Packard Laboratories, April 1993.

[87] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., and Lyon, B. Design and Implementation of the Sun Network Filesystem. In *USENIX Conference Proceedings, Summer 1985* (Portland, OR, June 1985), USENIX, pp. 119–130.

[88] Satyanarayanan, M. A study of file sizes and functional lifetimes. In *Proceedings of the 8th Symposium on Operating System Principles* (1981), pp. 96–108.

[89] Satyanarayanan, M. The Influence of Scale on Distributed File System Design. *IEEE Transactions on Software Engineering 18*, 1 (January 1992), 1–8.

[90] Scheifler, R., and Gettys, J. The X Window System. *ACM Transactions on Graphics 5*, 2 (April 1986), 79–109.

[91] Seltzer, M. *File System Performance and Transaction Support.* PhD thesis, University of California, Berkeley, 1992. Also available as UCB:CSD technical report UCB:CSD-93-741.

[92] Shand, M. Measuring system performance with reprogrammable hardware. Tech. Rep. PRL-RR-19, DEC Paris Research Laboratory, August 1992.

[93] Shein, B., Callahan, M., and Woodbury, P. NFSSTONE: A Network File Server Performance Benchmark. In *USENIX Conference Proceedings, Summer 1989* (Baltimore, MD, 1989), pp. 269–275.

[94] Shirriff, K., and Ousterhout, J. A Trace-Driven Analysis of Name and Attribute Caching in a Distributed System. In *USENIX Conference Proceedings, Winter 1992* (San Francisco, CA, 1992), USENIX, pp. 315–332. Availability anonymous ftp `cs.berkeley.edu:~ftp/papers/nameUsenix92.ps.Z`.

[95] Siebenmann, C., and Zhou, S. *Snooper Users Guide.* University of Toronto, August 1993.

[96] Smith, A. Analysis of Long Term File Reference Patterns for Application to File Migration Algorithms. *IEEE Transactions on Software Engineering SE-7*, 4 (July 1981), 403–417.

[97] Smith, A. J. Long term file migration: Development and evaluation of algorithms. *Communications of the ACM 24*, 8 (August 1981), 521–532.

[98] Smith, A. J. Disk cache - miss ratio analysis and design considerations. *ACM Transactions on Computer Systems 3*, 3 (August 1985), 161–203.

[99] Spasojevic, M., and Satyanarayanan, M. A usage profile and evaluation of a wide-area distributed file system. Tech. Rep. CMU-CS-93-207, School of Computer Science, Carnegie Mellon University, October 1993. Also appeared in Winter USENIX Conference, San Francisco, CA, January, 1994.

[100] Steiner, J. G., Neuman, C., and Schiller, J. I. Kerberos: An Authentication Service for Open Network Systems. In *USENIX Conference Proceedings* (Dallas, TX, 1988), pp. 191–202.

[101] Strange, S. Analysis of long-term UNIX file access patterns for application to automatic file migration strategies. Tech. Rep. UCB-CSD-92-700, Department of Computer Science, University of California, Berkeley, 1992.

[102] Sun Microsystems. *nfsd(8) - NFS daemon*, SunOS Reference Manual ed., 1988.

[103] Sun Microsystems. *NIT(4) Network Interface Tap*, SunOS Reference Manual ed., 1988.

[104] Sun Microsystems. *dbm(3x) - database subroutines*, SunOS Reference Manual ed., 1993.

[105] Sun Microsystems Computer Company, Mountain View, CA. *SMCC NFS Server Performance and Tuning Guide*, November 1994. Part No. 801-7289-10.

[106] Tanenbaum A., et al. Experiences with the Amoeba Distributed Operating System. *Communications of the ACM 33*, 12 (1990).

[107] Thekkath, C. A., Wilkes, J., and Lazowska, E. D. Techniques for file system simulation. Tech. Rep. HPL-92-131, Hewlett Packard Laboratories, October 1992. Also published as Technical Report 92-09-08, Department of Computer Science and Engineering, University of Washington, Seattle, WA.

[108] Thompson, J. File Deletion in the UNIX System: Its Impact of File System Design and Analysis, April 1985. Computer Science Division,EECS,University of California, Berkeley CS 266 term project.

[109] Thompson, J., and Smith, A. Efficient (stack) algorithms for analysis of write-back and sector memories. *ACM Transactions on Computer Systems 7*, 1 (February 1989), 78–117.

[110] Thompson, J. G. *Efficient Analysis of Caching Systems*. PhD thesis, EECS, University of California, Berkeley, September 1987. Also available as UCB/EECS technical report CSD-87-374.

[111] Wall, L., and Schwartz, R. L. *Programming perl*. O'Reilly and Associates, Inc., Sebastopol, CA, 1990.

[112] Walsh, D., Lyon, B., Sager, G., Chang, J., Goldberg, D., Kleiman, S., Lyon, T., Sandberg, R., and Weiss, P. Overview of the Sun Network File System. In *USENIX Conference Proceedings, Winter 1985* (Dallas, TX, 1985), pp. 117–124.

[113] Watson, A., and Nelson, B. LADDIS: A Multi-Vendor and Vendor-Neutral SPEC NFS Benchmark. In *USENIX LISA VI October 19-23, 1992* (Long Beach, CA, 1992), pp. 17–32.

[114] Welch, B. *Naming, State Management and User-Level Extensions in the Sprite*. PhD thesis, University of California, Berkeley, 1990. Also available as UCB:CSD technical report UCB:CSD-90-567.

[115] Welch, B. The File System Belongs in the Kernel. In *2nd USENIX Mach Symposium, Nov 20-22, 1991* (November 1991), pp. 233–250. Availability anonymous ftp `ftp://sprite.berkeley.edu/papers/fs-in-kernel.ps`.

[116] Welch, B. Measured performance of caching in the Sprite network file system. Tech. rep., Computer Science Department, University of California, Berkeley, July 1991. Availability anonymous ftp `ftp://sprite.berkeley.edu/papers/cache-performance.ps`.

[117] Zhou, S., DaCosta, H., and Smith, A. J. A File System Tracing Package for Berkeley UNIX. *Proceedings 1984 USENIX Summer Conference Portland Oregon June 12-14,* (June 1985), 407–419.

[118] The Berkeley NOW Project, April 18th, 1995. Available via the World Wide Web `http://now.cs.berkeley.edu/`.

[119] The World Wide Web, April 18th, 1995. Available via the World Wide Web `http://www.w3.org/`.

[120] xFS : Serverless Network File Service, July 18th, 1995. Available via the World Wide Web `http://now.cs.berkeley.edu/Xfs/xfs.html`.