Eliminating File System Scan for Backup Using a Pseudo VFS Driver

by

Varghese Devassy

A Thesis submitted to the Faculty of Graduate Studies of

The University of Manitoba

in partial fulfilment of the requirements of the degree of

Master of Science

Department of Computer Science

University of Manitoba

Winnipeg, Manitoba, Canada

Eliminating File System Scan for Backup Using a Pseudo VFS Driver

By

Varghese Devassy

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirement of the degree

Of

Master of Science

Thesis advisor

Dr. Rasit Eskicioglu

Author

Varghese Devassy

Eliminating File System Scan for Backup Using a Pseudo VFS Driver

Abstract

A typical file system backup application operates in two phases. In the first phase, called *scan*, a list of candidate files modified since the last backup is generated. These files are then copied to some form of a secondary storage device in the second phase called, *backup*. Although improvements in the speed of secondary storage devices have reduced the time required for the *backup* phase, this is not the case with the *scan* phase. The *scan* phase must compare the modification time of a file with the time of the previous backup before the file is selected as a backup candidate. On most systems, getting the modification time of a file requires at least one system call. As the number of files on a computer increases, the number of context switches resulting directly from the system calls executed during the *scan* phase increases proportionally, resulting in lower system throughput. In this thesis, I introduce a novel method to speed up the *scan* phase through a pseudo Virtual File System (`VFS`) driver (referred to as *scandd*). `VFS` is a framework available on modern UNIX operating systems that allows the coexistence of different file system types. Ioctls provided by *scandd* can be used to specify the list of file systems that *scandd* monitors, enabling it to generate the list of modified files segregated by their file system. Using this method, the *scan*

phase would use the list of modified files generated by *scandd*, thereby eliminating the need to compare the modification times of individual files. This method can be used for any file system that conforms to the `VFS` semantics. For reasons to be cited later, the Solaris 10 operating system (OS) will be used to demonstrate the performance improvements obtained by this method.

Being a device driver, *scandd* has negligible impact on unmonitored file systems. The tests conducted indicate that there is about 30% degradation in file creation operations on monitored file systems. On dormant file systems such as the standard Solaris file systems (`/`, `/usr`, `/var` and `/opt`) the gains observed are dramatic. For file systems with directories containing greater than 1 million files *scandd* provided 60% improvement even if all files in the file system were modified. For the deepest directory containing 6 million files at 492 levels deep, *scandd* provided 50% improvement even if all files were modified. Further, my tests indicate that the time taken for *scan* using the list generated by *scandd* depends only on the number of files modified and the pathname length of the modified files.

Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank my advisor, Dr. Rasit Eskicioglu, for his guidance and encouragement all these years. Being a part-time student and working as a full-time employee in a software development firm, it was hard to motivate myself to complete this thesis. Moreover, doing my thesis remotely, I missed some registration deadlines. Dr. Eskicioglu, thank you for motivating me and for being patient with me all these years.

I would like to acknowledge the following members of the Computer Science department for their help and advice. I take this opportunity to thank Dr. Peter Graham for spending time with me brainstorming various ideas and implementation hurdles. Dr. Neil Arnason for helping me elevate my writing style (although I am still not there yet). Ms. Lynne Hermiston for processing my late registrations and time extension forms without any further delay.

I am grateful to my employer, CommVault Systems Inc., for partly funding my Master's studies at the University of Manitoba and allowing me to take time off to attend lecture sessions.

Many people have spent time reading this thesis and provided valuable feedback. Especially, Mr. Thomas Barnwell, Mr. Ian Austen, Mr. Kiran Koala and last but not least my wife, Dr. Smita Pakhale. This thesis would not have been in this current form without your valuable comments.

This thesis is dedicated to the system administrators who spend hours waiting for completion status for their large file system backups.

Chapter 1

Introduction

The backup process is used to safeguard against unexpected data loss due to software or hardware failures. File system backup operates in one of three modes: *full*, *incremental* or *differential*. *Full* backup saves all files on a system to secondary storage; *incremental* backup saves files modified since the last *full*, *incremental* or *differential* backup, and *differential* backup saves all files modified since the last *full* backup. Apart from the above three types, another backup type, namely, *synthetic full*, is available in some recent backup applications [1, 2]. *Synthetic full* backup creates a full backup from a previous *full* backup and the subsequent incremental or differential backups combined with any modified or new files from the system. In this method, most of the files are available from secondary storage, so a *synthetic full* backup uses limited system resources. In addition, *synthetic full* backup dramatically reduces the backup time, because files that were not modified are read from backup storage.

Files to be backed up must be specified or selected from the list of files by a process

called *enumeration*. To enumerate files for an *incremental* or a *differential* backup, the *enumeration* process must obtain the modification times of all files and compare them with the time of the previous backup. To obtain the modification time of a file, the *enumeration* process must issue a system call to the file system. Execution of system calls by the *enumeration* process on each file affects the performance of other applications running on the system due to the context switches they introduce. In addition, the *enumeration* process can take a long time on extremely large file systems and on file systems with deep directories. Section 1.2 explains the reasons why *enumeration* can be slow on large file systems and on file systems with deep directories using a pseudo-code. In this thesis, I have developed a method whereby files for *incremental* backups can be enumerated without comparing their modification times thereby reducing the *enumeration* time.

1.1 The Need for Faster Enumeration

Backup of critical computer data is one of the most important elements of any organization's disaster recovery planning. Most organizations rely on third party applications to perform this vital task. These backup applications save modified files at a predetermined schedule to some form of secondary storage devices (most commonly tape drives) connected to one or more backup servers. In a typical organization, the number of systems to be backed up exceeds the number of backup servers and secondary storage devices. To accommodate all systems in an organization, most backup applications automate backups to secondary storage devices using medium-changer devices. The use of medium-changer devices accomplishes the movement of backup

media to and from drives without human intervention. Medium-changer devices typically consist of a set of storage slots where backup media are stored and a set of drives where the backup media can be mounted and then read from or written to as specified by the Small Computer System Interface (SCSI) standard [3].

The explosive growth of data has created numerous challenges for every organization: one of them being the time required to complete backup within a predetermined period. To reduce the amount of data to be backed up, vendors of backup applications have devised novel methods. Chervenak et al. [4] describe some of the methods used by these vendors. However, the growth of data has surpassed the advantages these methods could deliver. To make matters worse, the total time available for backup, called the backup window, remains constant.

A file system backup generally has two phases: *scan* and *backup*. The *scan* phase enumerates files and the backup phase saves these files onto a secondary storage device connected to a backup server. Over the years, the speed of secondary storage devices has improved dramatically, which has improved the performance of the *backup* phase. However, the speed of the *scan* phase largely depends on the performance of directory lookup in file systems. Moreover, directory lookup speed reduces drastically as the number of files in a directory increases. Phillips [5] cites some of the enhancements in the *ext2* file system resulting in faster directory lookup. Newer file systems use BTrees [6] and their variants to improve directory lookup. The method employed by Phillips [5] uses hash keys to improve the directory lookup performance for which he has coined the name "HTree" [5]. Although such newer techniques have improved the performance of directory lookups, large directories still present a performance

bottleneck.

Large directories may not be a common occurrence in regular file systems. However, there are instances such as the cache directories of large web servers and directories containing the files created by various applications, where directories with large numbers of files are common. It is also possible to come across large directories in systems used for archiving medical records and medieval records. In addition, researchers working in the area of file system design and/or file system performance usually create large directories intentionally to study and to improve file system performance.

Even in the absence of large directories in a file system, *scan* may take a long time (the exact amount of time depends on the number of files in a file system) on very large file systems (file systems that house the home directories of the employees in a large organization is a prime example of this). This is because, *scan* must use at least one system call per file for enumeration. As the number of files in a file system increases, the amount of time taken by *scan* increases proportionally. In addition, the number of context switches resulting from these system calls reduces the throughput of other applications running on the system.

1.2 Discussions of a Sample *Scan*

The method developed in this thesis focuses on improving the time taken for the *scan* phase of an *incremental* backup. Figure 1.1 shows how *scan* using the traditional method operates. In this algorithm, `scan()` is the main function called to scan all mounted file systems. For each mounted file system, *scan* invokes the `traverse()` function, which performs a depth-first search of the selected file system listing the

names of files modified since the previous backup time (referred to as the "reference time").

```
traverse(fs)
{
   for each child ch of fs {
     if ch is a directory {
       traverse(ch)
     }

     obtain modification time (mtime) of ch
     if mtime of ch is greater than the reference time{
       print ch
     }
   }
}

scan()
{
   for each mounted file system fs {
     traverse(fs)
   }
}
```

Figure 1.1: Pseudo-code for a traditional *scan* algorithm

For the `traverse()` function in Figure 1.1 to compare modification times, it must first obtain each file's modification time using a `stat()` system call. The `stat()` system call retrieves the important attributes of a specified file such as: size, modification time (`mtime`), status change time (`ctime`), access time (`atime`), owner and group, etc. On UNIX systems, a system call allows a user process to request OS services (Vahalia [7, p. 31] describes the system call interface on UNIX systems). Access to OS services however, comes with a price. To decide whether a file is selected for

backup in the algorithm above, the `traverse()` function must compare the results of `stat()` with the reference time. A file system with a small number of files (number of files ranging up to 100,000 files), can perform the above *scan* algorithm within a few minutes (upper bound of 15 minutes on a standard UNIX system). However, the time taken by the above algorithm increases proportionally as the number of files in a file system increases. Scalability tests performed by Sweeney et al. [8] on the *XFS* file system confirm this assumption. According to the test conducted by Sweeney et al. [8], 6716 lookups/second can be done on a directory with 10^4 entries where as only 66 lookups/second can be performed in a directory with 10^6 entries. This means that the lookup performance also decreases proportionally as the number of files in a directory increases.

Implementation of directories in file systems differ from those of files in that the contents of a directory are the names of its children [9]. Directory lookup on large directories are slow because, to lookup the name of a child in a directory, a file system must search for the file name in the directory contents. However, as the number of children in a directory increases, so does the time taken for a lookup because the search space is much larger.

Although newer file systems such as *ext3* on Red Hat Linux, *ReiserFS* on Suse Linux and *UFS* on Solaris perform better than *XFS* due to the various caching strategies implemented in them, *scan* still takes a long time on a system with very large file systems, directories containing a large number of files (greater than 1 million files), and/or file systems with deep directories. The exact amount of time it takes to complete a *scan*, of course, also depends on factors such as the memory size, CPU

speed, etc.

1.3 The Method

In modern operating systems, many file system implementations follow a standard portable architecture, namely, `vnodes` [10]. Figure 1.2 shows a block diagram of a sample `vnode` implementation. One of the design goals of the `vnode` architecture was to split file system implementation into file system dependent and file system independent modules. To effect this goal, the `vnode` was introduced to manage files independent of the file system they reside in; the `VFS` was introduced to manage file systems independent of their implementation. As a consequence, each file is represented in the kernel by a `vnode` and each file system by a `VFS`. All file systems adhering to the `VFS` architecture must define a set of predefined entry points, namely, `vnodeops` (acronym for 'vnode operations'). These entry points are C-function pointers that are initialized when a new file system is added to the system. The OS maintains an array of function pointers for all file systems configured. This array is indexed by the file system number, `vfs_fstype`, which is assigned by the OS when a file system is configured. To perform a file-related operation, the OS determines the file system number the file resides in, which is then used to access the function pointer specific to the file system.

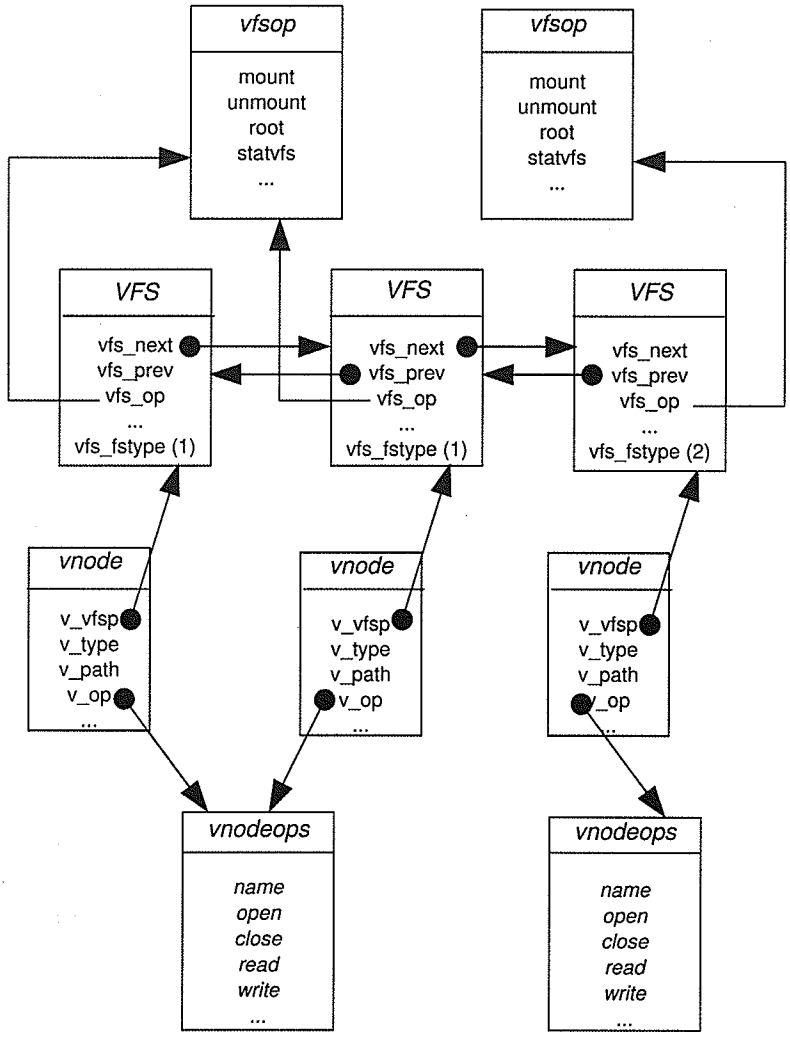

Figure 1.2: Block diagram showing the relationship between `vnode` and `VFS`

In Figure 1.2, a `vnode` refers to the file system that it is part of through the `v_vfsp` pointer. The pointer, `v_op`, points to the `vnodeops` for the file system type. `Vnodes` that belong to file systems of the same type share a common `vnodeops` structure. Similarly, a `VFS` points to its file system specific functions through the pointer `vfs_op`. File systems of the same type share a common *vfsop* structure. All file systems configured on the system are doubly-linked through the pointers: `vfs_next` and `vfs_prev`.

In this thesis, I have developed a new method to enumerate files for *incremental* backup without the normal expensive enumeration process. Using this method, a pseudo driver, *scandd*, is stacked between the `VFS` layers and the actual file systems configured on the system. At initialization, *scandd* replaces the `vnodeops` function pointers defined by the configured file systems with its own functions. A file system defines many `vnodeops` functions; *scandd* however, replaces only those functions that modify a file's contents or a file's metadata (time, permissions, etc.). Once the required functions are replaced, all file system access from the `VFS` layer is routed through *scandd*. This enables *scandd* to determine which files are being modified and to create a log of those files segregated based on the file system in which they reside. After logging the names of the modified files, *scandd* calls the original file system recipient function to complete the file system operation requested of the OS.

The Solaris 10 is selected as the platform for implementation because only Solaris 10 and later versions provide full pathname of a file modified as a member of the `vnode` structure. `VFS` implementations on other operating systems and on earlier Solaris versions either provide only the base-name of a file or do not even have the file name

as a member of the vnode structure. On some operating systems, such as Linux, the full pathname is available using certain internal kernel functions; on other operating systems, there are no well defined kernel functions to obtain the full pathname from the vnode of a file. As the full pathname of a file is provided by the Solaris 10 kernel without invoking extra kernel functions, the Solaris 10 implementation should provide the best performance improvement allowing me to demonstrate the full potential of my approach.

To measure the performance improvements obtained by this method, I have developed file system *scan* utilities using both the traditional and the new methods. I have used these utilities to compare the performance improvements obtained by the new method on various file system configurations, such as: file systems containing large numbers of files, file systems with directories containing large numbers of files, and file systems with deep directories. These results are discussed in Chapter 4.

1.4 Thesis Overview

The rest of this thesis is organized as follows. Chapter 2 details the related work in file systems, especially those which improve the performance of directory lookups; some of the techniques commonly used by backup applications to bypass the *scan* phase, and concludes with a section describing why *scan* is slow on directories containing large numbers of files. Chapter 3 explains the solution strategy and the implementation details with particular emphasis on the data structures and the algorithms used in the implementation. Chapter 4 details the methods that I have used to evaluate the correctness of my thesis implementation and in addition, describes

the measurements carried out to compare the performance of *scan* using the method that I have developed with the traditional *scan* method. Chapter 5 details the future work required and offer some concluding remarks on this thesis.

Chapter 2

Related Work

This chapter contains an overview of the various enhancements that have been made to improve file system performance. Section 2.1 discusses caching strategies. As memory gets cheaper, there is more emphasis on the use of larger caches to improve file system performance. As Solaris is a UNIX variant, references to UNIX file system concepts and caching techniques also apply to Solaris 10. In Section 2.2, I discuss some of the novel methods used in some backup applications to eliminate *scan* altogether. In the same section, I also introduce the NT file system (NTFS) change journal developed by Microsoft Corporation [11].

2.1 File System Caches

Implementers of early UNIX file systems noticed the impact of disk latency on file system performance. Since then, operating system designers have used several caching strategies to improve file system performance. Tannenbaum [12, pages. 270–

272] explains the use of a *buffer cache* in early UNIX file systems to improve read and write performance. A *buffer cache* is a cache layer common to all file systems. Functionally, it resides between a file system and the hardware devices on which the file system resides, usually disks. The *buffer cache* is designed to cache individual disk blocks [13, p. 474]. Cache items pertaining to individual disks are kept as separate lists [13, p. 482] in the *buffer cache* to facilitate searches for blocks of a given disk in the *buffer cache.* Thus, an individual block in the buffer cache can be accessed by the tuple *[disk-id, block-no].* Before a file system reads a block from the disk, it checks for the existence of the same block in the *buffer cache.* If the requested block is present in the buffer cache, the read request is satisfied from the cache, avoiding a disk access. However, if the requested block is unavailable in the *buffer cache*, it is read from the disk through the *buffer cache.* In a similar manner, a write request to the file system is written into the *buffer cache.*

In general, caches can be implemented as *write-through* or *write-behind* [7, p. 285]. In write-through caching, data is written to disk at the same time it is written to the cache. In write-behind caching, data is written to the cache first; the cached data is written to disk at a later time. Write-through caching is simpler to implement than write-behind caching. However, write-through caching results in an immediate write of the cached disk block eliminating the benefits of caching. As disk access takes much longer than memory access, delaying disk I/O improves the system throughput as the CPU can be utilized for other tasks. Therefore, the UNIX *buffer cache* is implemented as a write-behind cache [7, p. 285].

Buffer cache entries are sorted in least-recently used (LRU) order. As the size

of the *buffer cache* is limited, LRU cache entries are purged in favor of newer blocks accessed from the disk. The kernel process, *fsflush* [14, p. 563], monitors the use of the *buffer cache* to ensure that free blocks are available, and that the modified blocks (dirty blocks) are synchronized to the disk. The *fsflush* process, at regular intervals, cycles through the modified cache blocks and synchronizes them to the disk.

As the processing power of computers has increased, file systems have increased in size and complexity. Use of the *buffer cache* by itself no longer provides the required performance improvements on large file systems. In addition, as the *buffer cache* resides below the file system layer, file system functions must be invoked to convert a file offset to a disk block and then to determine whether that block resided in the *buffer cache* [14, pages. 589–590]. Invoking file system functions to determine the presence of a block in the *buffer cache* is expensive [14, pages. 589–590]. Therefore, modern operating systems use a different cache layer, namely, the *page cache* instead of the *buffer cache*. Unlike the *buffer cache*, the *page cache* is designed to cache a file's content and not disk blocks. Moreover, the page cache is designed to use all available memory that is not used by applications running on the system. Vahalia [7] states that with the introduction of the *page cache*, the *buffer cache* is only used to cache file system metadata [7, p. 284]. Modern OSes use some variant of the *page cache*. Bar [15, pages. 29–32] explains the relevance of the *buffer cache* and *page cache* [15, pages. 79–80] with regard to Linux file systems.

These caching strategies have substantially improved file system performance for reads and writes. However VFS refers to a file using a vnode structure rather than the file name. Prior to the introduction of the vnode architecture [10], a file in a file

system was uniquely represented by an `inode`. With the introduction of the `vnode` architecture, a file is now represented by a `vnode` regardless of the file system it is part of. However, the `inode` representation is still used in file system implementations. Thus, within the `VFS` layer, a file is represented by a `vnode` and within specific file system implementations, a file is represented by an `inode`. The `VFS` therefore must convert a file/directory name to its `vnode` before any file/directory operation. In many cases, files are accessed using the full pathname from the root directory. McKusick et al. [16, p. 222], explain the actions required to translate a pathname to its `vnode`. To convert a pathname to a `vnode`, the pathname is traversed component-by-component (e.g., `/a/b` is converted by traversing `/`, `/a` and then finally `/a/b`). To make pathname to `vnode` lookup efficient, a cache has been implemented in modern UNIX operating systems. This cache is called the *Directory Name Lookup Cache* (DNLC) [14, pages. 554–562] in Solaris and the *dentry cache* [15, pages. 79–80] in Linux.

2.2 Backup Methods using Newer File System Features

The *Buffer cache*, *page cache*, and *Directory Name Lookup Cache/dentry cache* have greatly improved file system lookup performance. However, performance improvements derived from these caching techniques do not guarantee completion of backups in the *backup* window. This is because, a traditional file system backup will backup the entire contents of a file even though, only a single byte in a file was modified. Thus, the amount of data backed up in a traditional file system backup could

be large even for the meager amount of file modifications in a system. Moreover, the backup data might have to be sent across a network to be written to the secondary storage media. Thus, the high volume of data in a traditional file system backup results in high network bandwidth requirements and higher consumption of backup media. Therefore, some backup vendors have introduced new backup methods such as: block-level backup [17], backup using file system snapshot, file-level replication techniques [18] and single instancing [19, 20].

Block-level backup applications eliminate the aforementioned shortcomings of the traditional file system backup by saving only the modified blocks in a file system. A separate device driver may be required to enable access to the modified blocks in a file system. Chervenak et al. [4] state that since block-level backup operates at the file system block level, it cannot correlate an individual block to a file unless file system specific information is included with the backup. Therefore, block-level backups are file system specific and require additional effort to be usable on different file system types. Moreover, depending on the block modification time, blocks pertaining to a file may be backed up on different days in a block-level backup. Therefore, restore of a single file from a block-level backup requires special processing and may take longer than restoring the same file from a traditional file system backup. This is because, restores from block-level backups are done as individual disk blocks rather than as individual files. In addition, block-level restores bypass file system code to restore the modified blocks. Therefore, administrators must unmount file systems before restoring from block-level backups.

Some newer file systems provide a snapshot feature [17]. IBM Corporation's

jfs2 file system and the *ext3* file system on Linux both support a snapshot feature. Garimella [21] gives an overview of the different snapshot methods used. This feature provides a read-only, point-in-time view of all files in a file system. To implement the snapshot feature, file systems employs *copy-on-write* scheme when modifying blocks. When a block in a "snapped" file system is written, the file system preserves data from the modified block by copying it to a new location. Once the copy is completed, the new data is written to the original location. To provide a consistent view of files in the "snapped" file system, references to the modified blocks are transparently redirected to the preserved blocks. Prior to the availability of the snapshot feature in file systems, critical applications had to be shut down before backing up their files. However, as the snapshot provides a consistent view of the "snapped" file system, a shut-down is no longer required.

File system snapshot can be combined with block-level changes to implement asynchronous file system level replication [22]. This method is also referred to as "continuous data protection" [21]. The replicated data may reside locally or remotely. This method uses the snapshot facility provided by file systems to create a reference point for replication, which is also used to synchronize the replication target. Methods such as remote copy can be used to synchronize the source snapshot with the target system. Once the target is in sync with the source snapshot, all changes to the file system since the snapshot was taken are applied asynchronously to the replication target. If the file system cannot provide block-level changes since the snapshot was taken, a device driver might be required to capture this information. Similar to replication at the file system level, file-level replication is also available. The only

difference between the above two methods is in the way the changes are applied. While the former applies block-level changes to the target, the latter applies file-level changes to the target. To provide fault tolerance, more than one destination site may be specified, increasing data availability in case of a disaster. In addition to the above replication techniques, replication may be built into the file system. Liskov et al. [18] describe a file system in which, replication of file system data is achieved using NFS. Replication built into the file system achieves the replication atomically as the writes are synchronous. A replicated file system may be used as the secondary storage media for backup thereby protecting the backed-up data itself.

In all of the above methods, compression may be employed to reduce the amount of data sent across the network or to reduce the amount of data stored on the backup media. However, the degree of compression achieved will depend on the contents of a file and in some cases can expand the data. With the wide spread use of email in the enterprise for communication, the same email or the same email attachments may be sent to multiple recipients. In a similar manner, many user-level directories may contain copies of the same file resulting in multiple copies of the same file being backed up. Backup vendors have developed a single instance feature whereby only one copy of a file is saved during the *backup* phase. Hong et al. [19] describe a file system in which blocks that have the identical data are logically associated to a single block. Bolosky et al. [20] explain the implementation of single instancing in Windows 2000. The implementation of single instancing on Windows provides support for backup and restore applications to detect files that have been "single instanced". Using this feature, backup applications can ensure that only a single copy of files with the same

content is saved during the *backup* phase.

Microsoft Corporation introduced a change journal in its NT file system (NTFS) for their Windows 2000 operating system [11]. Each NTFS has its own change journal, which is described as a database of changes made to the file system. This journal contains entries for every file and directory modified, added, or deleted. Microsoft envisions that the change journal will be used by services, virus scanners, and backup applications. The change journal is implemented as a hidden file. Because the change journal creates a hidden file on the file system being tracked, it takes away some space from the file system.

2.3 Summary of Related Work

The size of file systems will keep increasing with the reducing price of hardware and the increasing processing power of computers. Therefore, the traditional methods of file system backup will not be able to complete backups in the prescribed time frame. As we have seen in the previous sections, novel methods for backing up files are being developed and file systems are absorbing these features continuously. It is only a matter of time before the method developed in this thesis will be made available as a standard feature in the coming years.

Chapter 3

Solution Strategy and *Scandd's* Implementation Details

In this chapter, I explain the method that I have devised to speed up file system *scan*. As explained in Section 1.3, this method requires the introduction of a pseudo driver, *scandd*. The various sections in this chapter are arranged as follows: Section 3.1 explains Solaris device driver fundamentals and Section 3.2 details the installation procedures required for installing *scandd*. Section 3.3 details *scandd's* configuration values and Section 3.4 explains the initialization steps of *scandd*. The mechanism by which *scandd* is stacked between the `VFS` layer and the file system implementation layer is explained in Section 3.5 and Section 3.6 explains the *ioctls* provided by *scandd* for communication with user-level applications. A *scan* using the method developed in this thesis uses these *ioctls* to monitor file systems and enumerate the list of files for backup without issuing `stat()` system calls.

Section 3.7 details the data structures used to store the list of monitored file sys-

tems and the *mutexes* used to serialize access to this list (a *mutex* is a data structure provided by the OS to implement mutual exclusion of critical sections in a kernel module). While monitoring a file system, it is possible that some files/directories may be modified more than once resulting in multiple entries for these files/directories in the log created by *scandd*. Section 3.8 details the data structures and method by which *scandd* eliminates some of these duplicate entries in its log. Section 3.9 mentions the two ioctls that applications may use to check for errors in the driver and Section 3.10 show how a sample application (referred to as *iscan*) can be developed using the method developed in this thesis. Section 3.12 highlights a special case in *scandd's* implementation.

3.1 Solaris Device Driver Fundamentals

Every device driver in the Solaris OS must conform to a specific standard mandated by the Solaris Driver Development Guide [23]. This standard specifies a set of mandatory routines and a set of mandatory data structures that a device driver must define and export before it can be activated on the Solaris OS. The mandatory routines are: `_init`, `_fini` and `_info`. The `_init` and `_fini` routines provide the initialization and termination functionalities for a device driver; the `_info` routine provides information on one of the mandatory data structures exported by a driver, `modlinkage`. The `modlinkage` data structure, in turn, exports two other important data structures, namely, `dev_ops` and `cb_ops`. The structure, `dev_ops`, specifies the driver revision and the names of routines for some of the driver-related tasks such as: attach, detach, reset, etc. On the other hand, the structure, `cb_ops`, provides infor-

mation on routines related to the I/O functionality of a driver including: *open, close, read, write, ioctl,* etc. A driver may choose the routines it exports depending on the functionalities that it provides. All undefined routines in `dev_ops` and `cb_ops` must be either set to `NULL` or to a predefined values as mandated by the Driver Development Guide. For example, if a driver does not allow open on the device it controls, the open routine in the `cb_ops` structure must be set to `nodev`, a predefined value indicating that the device cannot be opened. To prevent name clashes with other driver's routines, driver developers are encouraged to use a prefix (usually the driver name itself) in the routines or data structures a driver exports. Since *scandd* is a pseudo driver, the only routines exported by `dev_ops` are `scandd_attach` and `scandd_detach`. I will be focusing on the functionalities implemented by `scandd_attach` in Section 3.4 and by the `scandd_detach` routine in Section 3.11. *Scandd* exports only three routines from the `dev_ops` structure, namely, `scandd_open`, `scandd_close` and `scandd_ioctl`. The above routines are the entry points for *open, close* and *ioctl* system calls respectively.

3.2 *Scandd* Installation

In this section, I describe the process of installing *scandd.* On Solaris, all device drivers that are not required during system startup and the files containing their configuration values must be placed in the directory `/usr/kernel/drv/sparcv9` and `/usr/kernel/drv` respectively. In the case of *scandd,* the installation script places the driver binary, `scandd`, and the configuration file `scandd.conf` in the above mentioned directories. In addition, the installation script places the entry '`type=ddi_pseudo; name=scandd scandd`' into the file `/etc/devlink.tab`. This line instructs the Solaris

OS to create the file, `/dev/scandd`, when *scandd* is loaded. The file, `/dev/scandd`, is a device file used to communicate with *scandd* through the *ioctl* interface. Once *scandd* and its configuration files have been copied to the above-mentioned directories, the installation script invokes the command '`add_drv scandd`', which initiates the process of loading *scandd* dynamically. The program, `add_drv`, is a standard Solaris program used to load device drivers dynamically. Once the driver is loaded into memory, the Solaris kernel invokes the routines `_init`, `_info` and `scandd_attach`, in that order.

As mentioned in Section 3.1, the `_init` routine provides the kernel with the address of the `dev_ops` and the `cb_ops` structures. It is through the `dev_ops` structure that the kernel determines the address of the attach routine to be called, namely, `scandd_attach`. All the necessary initializations of *scandd* are performed by the `scandd_attach` routine. The next section details the configuration values used by *scandd* followed by the initialization actions performed by the `scandd_attach` routine.

3.3 Configuration Values

At present, *scandd* uses only one configuration value, `SCANDD_TARGET_DIRECTORY`. The above configuration value specifies the directory into which the files containing the names of modified files are written. As mentioned in Section 1.3, file names from different file systems are segregated into different files. *Scandd* expects the above configuration value to be set to a directory that exists. *Scandd* will fail to load if this condition is not met.

It is mandated that the directory specified by the configuration value, `SCANDD_-`

TARGET_DIRECTORY, be on a separate file system from the ones being monitored and *scandd* will not monitor this file system due to the following reasons:

- If the modified file names from file systems, fs1, fs2, etc., are allowed to be written to files (mod-files1, mod-file2, ...) in any file system monitored by *scandd*, changes in these files will in turn result in newer change records, which will cause an endless recursion in the kernel.
- I do not want other applications to take away the space reserved for updates from the driver.

Free space requirement for the directory specified by the SCANDD_TARGET_DIRECTORY configuration value is not directly dependent on the size of the file systems monitored by *scandd*. Rather, it is dependent on the number of files in a file system, the average pathname length of files in the file system and the frequency with which these files change. In my test setup, the size of this file system was 10 GB. From my tests I observed that a file system containing 5 million files with pathname length of 1020 bytes (the maximum pathname length allowed being 1024) occupied $\approx$ 5 GB of disk space on this file system when all files in the file system were modified exactly once. Since it is unlikely that all files in a file system would change within a backup period, the space requirement per file system can be reduced. A more logical approach would be to reserve space for 10% of the files changing with an average pathname length of 128 bytes per *incremental* cycle per file system. To accommodate any unexpected space needs, the administrator should reserve some extra space in the file system specified by the configuration value, SCANDD_TARGET_DIRECTORY.

3.4 *Scandd* Initialization

This section details the actions performed by the `scandd_attach` routine during *scandd's* initialization. At the onset, `scandd_attach` calls the routine, `ddi_create_minor_node`. This device driver interface (DDI) function creates the device file, `/dev/scandd`, based on the entry inserted into the file, `/etc/devlink.tab`, during the installation. The remaining initialization actions of *scandd* are performed by the routine, `scandd_init`, which is called from `scandd_attach`.

The routine, `scandd_init` reads the configuration entry mentioned in the file, `/usr/kernel/drv/scandd.conf`. Reading the configuration is achieved by the standard DDI function, `ddi_prop_lookup_string`. Once read, *scandd* verifies whether the above configuration string identifies a directory that exists. *Scandd* will fail to load if this is not so. *Scandd* preserves the name of this directory in one of its internal data structures and in addition, obtains its `vnode` entry. When a `vnode` for a file/directory is obtained, the kernel increments the reference count of the `vnode` obtained, which is decremented when the `vnode` is released; a file system with active `vnodes` (`vnodes` with reference count greater than zero) cannot be unmounted. Thus, to prevent unmounting of the file system in which the above directory resides, the `vnode` for the above directory is not released until *scandd* is unloaded.

Further, *scandd* initializes the *mutexes* that are used to control access to the list of monitored file systems. Section 3.7 details the data structures used and the method by which the monitored file systems are stored and accessed in *scandd.* At this phase, the driver is ready to perform its last couple of initialization steps before it can start intercepting file system calls, which are detailed in the next section.

3.5 Intercepting File System Changes

As shown in Figure 1.2 and as described in Section 1.3, every file system configured under the VFS interface exports a list of function pointers that provides a list of file related operations, namely vnodeops. Similarly, every file system configured in the system is identified by a unique VFS C-language structure which, in turn, exports a list of file system operations known as vfsops. Every active file on the system is represented by a vnode; a vnode points to its vnodeops and to its VFS through the pointers, v_op and v_vfsp, respectively.

```
typedef struct scandd_savedvop
{
   /* is vnodeops already saved?  */
   char slot_taken;

   /* location for the original vnodeops */
   vnodeops_t vop;
} scandd_savedvop_t;
```

Figure 3.1: Data structure to store replaced vnodeops

During the initialization phase, *scandd* allocates an array of structures mentioned in Figure 3.1. The number of such structures allocated is computed by traversing the file systems configured on the system. As mentioned earlier, every configured file system has a VFS structure; all VFS structures configured on the system are linked together as a doubly-linked circular list. Solaris provides access to the above linked list through a pointer to the VFS of the root file system, namely, rootvfs. By traversing the list starting at the rootvfs pointer, *scandd* counts the number

of file systems configured on the system and in turn allocates the same number of `scandd_savedvop` structures. Once the array of structures has been allocated, *scandd* saves the `vnodeops` for all file system types encountered in the `VFS` linked list. This is achieved by obtaining the root `vnode` for a file system through the `vfs_root` function in `vfsops`. In addition, every `VFS` structure has a member, `vfs_fstype`, which specifies the type identifier for the file system type it implements; `VFS` structures for the same file system type have the same value for the field, `vfs_fstype`. *Scandd* uses the `vfs_fstype` member in the `VFS` structure to index into the array of `scandd_savedvop` structures. Once *scandd* has saved the `vnodeops` for a file system, it sets the variable, `slot_taken`, to 1 indicating that the `vnodeops` for that file system type have already been saved. It then replaces only the functions that modify the content or the metadata of a file or directory in the `vnodeops` with its own functions. Since these operations modify critical kernel data structures, *scandd* acquires a write lock on the `VFS` list by calling the function, `vfs_list_lock`, until `vnodeops` for all file systems have been inspected.

To intercept file modification calls for a file system, *scandd* must replace the `vnodeops` function pointers with the pointers to its own functions in the `vnodeops` for that file system. This operation need to be done only once per file system type. In addition, as *scandd's* functionality is limited to intercepting only the files or directories that have been changed in a monitored file system, only pointers to the functions that modify the contents of a file, directory or its metadata need to be replaced (e.g., write, chmod, remove, mkdir, rename, link, symlink, etc.). Before replacing the function pointers in the `vnodeops` for a file system type, *scandd* must save the original function

pointers. This is because, *scandd* must restore the replaced function pointers with their original values when it is unloaded from memory. *Scandd* uses the data structure in Figure 3.1 to save the `vnodeops` for a file system type.

Once the `vnodeops` functions for all file systems have been replaced, file modification calls are routed through the *scandd* driver. Every `vnodeops` function called receives a pointer to the `vnode` on which the file operation is done. It is through this `vnode` pointer that *scandd* is able to access the remaining data structures to perform its functions. In the replaced functions, *scandd* verifies whether the call is for a file/directory in a monitored file system. If the file system is being monitored, *scandd* must record the name of the modified file. Section 3.7 specifies the method by which the modified files in a file system is recorded and the related data structures used. Regardless of whether a file system is monitored or not, the original `vnodeops` function must be called to complete the file operation. *Scandd* achieves this through the original `vnodeops` function pointers saved in `scandd_savedvop`. This is achieved as follows:

1. for every function *Fn* intercepted by *scandd*
2. obtain the `VFS` pointer for the file system through the `vnode` member `v_vfsp`
3. obtain `vfs_fstype` value from the `VFS` pointer
4. access the original file system call as `scandd_savedvop[vfs_fstype].function(args)`

Since *scandd* intercepts some of the file system calls, it is extremely important that failures from *scandd's* internal operations should not be translated as a failure in

the file system call. Otherwise, file system operations may appear to report random failures even though they were successful.

3.6 Ioctl Interface

This section explains the *ioctls* supported by *scandd* and the data structures an application must use when invoking these *ioctls*. Section 3.10 explains how these *ioctls* can be used to build an efficient *scan* method. As is customary on UNIX, these *ioctls* return 0 for success and -1 for failure. In case of a failure, `errno` (a UNIX standard variable used to return error codes from system calls and library functions) specifies the error code.

In all the structures mentioned in this section, `MAXPATHLEN` is a constant defined in the system header files specifying the maximum pathname length in Solaris OS, set to 1024.

3.6.1 SCANDD_ADD_FS

An application using this *ioctl* must use the structure shown in Figure 3.2. *Ioctl* `SCANDD_ADD_FS` instructs *scandd* that the file system specified by the field, `mntpt`, be added to the list of monitored file systems. *Scandd* returns the error, `EEXISTS`, if the file system is already being monitored.

Upon successful completion, *scandd* creates a file to record the names of modified files for the specified file system in the directory specified by the configuration value, `SCANDD_TARGET_DIRECTORY`. The file created is named by replacing all '`/`' characters in the file system name with the '`#`' character. E.g., if the file system, `/a/b/c`, is

```
typedef struct scandd_add_fs
{
   /* IN: name of the mount point */
   char mntpt[MAXPATHLEN];
} scandd_add_fs_t;
```

Figure 3.2: Data structure for SCANDD_ADD_FS and SCANDD_DEL_FS ioctls

being added, *scandd* creates the file, `#a#b#c`. This method uniquely names the files created for file systems, `/usr` and `/a/usr`.

3.6.2 SCANDD_DEL_FS

This *ioctl* uses the same structure used by the *ioctl*, SCANDD_ADD_FS, described in Figure 3.2. *Ioctl* SCANDD_DEL_FS instructs *scandd* that the file system specified by the field `mntpt`, be deleted from the list of monitored file systems. Upon successful completion of this *ioctl*, all files created by the driver to store the names of modified files for the file system are removed.

3.6.3 SCANDD_IS_FS_TRACED

Ioctl SCANDD_IS_FS_TRACED is used by an application to query whether a file system is being monitored by *scandd* or not. Figure 3.3 shows the data structure used as an argument to this *ioctl*. On successful completion, the field, `is_traced`, is set to 1 if the file system is monitored and 0 otherwise.

```
typedef struct scandd_is_fs_traced
{
   /* IN: name of the mount point */
   char mntpt[MAXPATHLEN];

   /* OUT: is traced or not */
   int is_traced;
} scandd_is_fs_traced;
```

Figure 3.3: Data structure for SCANDD_IS_FS_TRACED *ioctl*

3.6.4 SCANDD_SYNC_CHANGED_FILES

Invoking this *ioctl* is an indication to *scandd* that the file used to record the names of modified files be closed and a new file be opened to record the modified file names. As will be described in Section 3.8, *scandd* caches the modified file names in its memory and this *ioctl* instructs *scandd* to write the modified file names from its cache to the change file. Since more than one *incremental* backup may be run between two *full* backups, *scandd* must segregate the files changed in each incremental into separate files. To achieve this, *scandd* renames the file containing the modified file names to a file appended with a version number. The version number is a running sequence starting at 0 and incremented after each successful SCANDD_SYNC_CHANGED_FILES *ioctl*.

Figure 3.4 shows the data structure used by this *ioctl*. The field, `chfile_no_rev`, returns the full pathname of the change file without the version number; the field, `revision`, specifies the version of the change file that was written and closed by *scandd*; the field, `error`, returns any error that the driver may have encountered while executing this *ioctl*.

```
typedef struct scandd_sync_changed_files
{
   /* IN: name of the mount point */
   char mntpt[MAXPATHLEN];

   /* OUT: change file without revision */
   char chfile_no_rev[MAXPATHLEN];

   /* OUT: latest revision saved */
   int revision;

   /* OUT: any error in the driver */
   int error;
} scandd_sync_changed_files_t;
```

Figure 3.4: Data structure for SCANDD_SYNC_CHANGED_FILES *ioctl*

3.6.5 SCANDD_GET_LAST_ERROR

Figure 3.5 shows the data structure used for the *ioctl*, SCANDD_GET_LAST_ERROR *ioctl*. This ioctl returns any error condition from the driver.

```
typedef struct scandd_get_last_error
{
   /* OUT: any error in the driver */
   int error;
} scandd_get_last_error_t;
```

Figure 3.5: Data structure for SCANDD_GET_LAST_ERROR *ioctl*

All of the described *ioctls* except SCANDD_GET_LAST_ERROR return the error ENOENT if the specified file system does not exist on the system or if it is not monitored

by *scandd*. In addition, if the file system specified in calling any of the *ioctls* is the same as the file system on which the directory specified by the configuration, `SCANDD_TARGET_DIRECTORY`, resides, *scandd* returns the error, `EINVAL`.

```
/* structure that represent a single traced file system */
typedef struct scandd_fs
{
  /* mount point of the file system */
  char mntpt[MAXPATHLEN];

  /* vnode of the file system being traced */
  vnode_t *fs_vp;

  /* count of threads using this structure */
  volatile int ref_count;

  /* conditional variable to wait for ref_count */
  kcondvar_t cond_var;

  /* variable indicating a thread is waiting to delete the FS */
  int waiting_to_delete;

  /* If this field is non-zero the change list is unreliable */
  uint_t error;

  /* device of the traced file system */
  dev_t fsid;

  /* list of files changed */
  scandd_change_list_t change_list;

  /* pathname of the change file */
  char change_file[MAXPATHLEN];

  /* vnode of the file to write the change list to */
  vnode_t *change_vp;

  /* version of the change file */
  int version;

  /* pointer to the next traced file system */
  struct scandd_fs *next_fs;
} scandd_fs_t;
```

Figure 3.6: Data structure to save a monitored file system

3.7 How Monitored File Systems are Stored

In this section, I explain the data structures used to store the monitored file systems information and the methods *scandd* employs to efficiently verify whether a file/directory modified is part of a monitored file system or not. *Scandd* uses the structure in Figure 3.6 to save the name of a file system monitored upon a successful `SCANDD_ADD_FS` *ioctl*.

The field, `fs_vp`, refers to the `vnode` of the root directory of the monitored file system. Reference to the root `vnode` is added when a file system is monitored by *scandd* and deleted when the file system is removed from *scandd's* monitored list. The field, `ref_count`, is for reference counting the number of kernel threads accessing a file system structure. The reason for implementing a reference count is that, many kernel threads may perform file modifications on the same file system at the same instant. Therefore, using a lock to access an individual file system structure for the entire duration of a file modification will degrade file system performance drastically. Instead, the lock controlling access to the file structure is taken only for the duration of time required to increment the `ref_count` field. The member, `cond_var`, is a variable of type `kcondvar_t`, a conditional variable [23, Ch. 3] used to wake up kernel threads waiting for the reference count, `ref_count`, to drop to zero. This may occur before a file system is removed from the list of monitored file systems and the thread trying to do so detects that the reference count for the file system is not zero. In this case, the thread trying to remove a file system data structure must sleep until all threads referring to the same file system structure decrements the reference count, `ref_count`. Before sleeping, the thread deleting the file system sets the field, `waiting_to_delete`,

to 1 and sleeps on the conditional variable, `cond_var`. If `waiting_to_delete` is set, *scandd* wakes up the sleeping thread whenever `ref_count` drops to zero. The field, `error`, is set if there was an error for this file system, which is returned as part of the processing of the *ioctl*, `SCANDD_SYNC_CHANGED_FILES`. The field, `fsid`, stores the file system identifier (FSID) for a file system. FSID is a combination of the `major` and `minor` numbers of the device from which a file system is mounted. Since every file system is mounted from different devices, FSID is unique for each file system. The field, `change_list`, stores the list of modified files until they are written to the `change_file` (the file that stores the names of modified files) whose `vnode` is stored in `change_vp`. Section 3.8 explains how *scandd* stores the list of modified files in its memory. The field, `mntpt` stores the name of the file system and the field, `version` stores the version of the file containing the modified file names last written for the file system. *Scandd* stores the file systems monitored as a linked list; the field, `next_fs`, points to the next file system in the list.

In response to the `SCANDD_ADD_FS` *ioctl*, *scandd* allocate the structure, `scandd_fs_t`, and copies the mount point information after initializing all the members to a known state. Next, *scandd* obtains the file system identifier (FSID) for the file system added. As described in Section 3.3, *scandd* verifies that the file system added is not the same as the file system containing the directory specified by the configuration value, `SCANDD_TARGET_DIRECTORY`. *Scandd* verifies this by comparing the FSID obtained with the FSID of the directory mentioned in the configuration field. The FSID for the directory in `SCANDD_TARGET_DIRECTORY` is obtained by traversing the `VFS` from the `vnode` for the directory saved during the initialization phases of *scandd* (as mentioned

in Section 3.4).

Since *scandd* has replaced the `vnodeops` for file systems as part of its initialization, every file modification call intercepted by *scandd* needs to verify if the file for which the call is made is part of any file system in the monitored list. Therefore, *scandd* must be able to lookup the list of monitored file systems efficiently. This is achieved by storing the collection of monitored file systems as a hash table. The hash table employed uses a number of buckets that is a power of two and the modulo operator is used to efficiently locate the appropriate bucket for a file system. The current implementation uses 16 buckets for the hash table and the hash value for a file system is computed as 'FSID modulo 16'. Since the number of buckets is a power of two, the modulo operation can be efficiently calculated using bitwise-AND operation (bitwise-AND with 15 in this case). *Scandd* uses an array of pointers to `scandd_fs_t` as the hash table. If more than one file systems hashes to the same bucket, they are linked together into a list using the field, `next_fs` in the `scandd_fs_t` structure. To serialize the kernel threads adding or deleting file systems to or from a hash bucket, *scandd* uses the same number of mutexes as the number of hash buckets. *Scandd* follows the following steps for adding or deleting a file system:

1. Obtain the FSID of the file system by traversing the `VFS` from the `vnode` of the file system's mount point.

2. Find the hash value for the FSID.

3. Acquire mutex for the hash bucket that the file system belongs to.

4. Verify whether the file system already exists in the hash bucket.

5. Perform addition/deletion of the file system to/from the hash bucket. If adding, always add the new file system at the beginning of the list

Number '16' is selected for the number of buckets due to the following reasons:

- Most systems tend to have a small number of file systems with a large number of files in them and hence a small number of hash buckets should suffice to hash them efficiently.
- The array of pointers to `scandd_fs_t` is declared as a global variable in the driver and I have selected a smaller value for the modulo operator there by balancing the space requirement for the array and the performance of the driver.
- There are no studies available that portray the number of mounted file system to make an informed selection on the number of buckets required. From my experience with Solaris systems, I have noticed that the number of mounted file systems is often in the range 5-10.

3.8 Identical File Name Elimination in the Modified File List

As mentioned in Section 1.3, *scandd* segregates the names of the modified files by their file system. It may be that a list of files in an file system might be modified many times resulting in identical entries for them in the list of modified file names. *Scandd* tries to eliminate these identical entries by using a cache of modified file names; the amount of identical file name reduction achieved depends on the file access pattern

of applications on a file system and is not generally predictable. Therefore, *scandd* makes no effort to predict any file access patterns. In this section, I discuss the method *scandd* employs for identical file name elimination. In addition, this section details the scenarios in which duplicate elimination is not possible.

In Figure 3.6, I had explained the data structure used to store a monitored file system. The member, `change_list`, in the above structure is the primary data structure that implements the cache of modified files for a file system. Figure 3.7 show this data structure in detail.

```
/* hash table structure of all changed entries */
typedef struct scandd_change_list
{
   /* first item in change list ordered by access */
   scandd_change_file_t *lru_first;

   /* last item in change list ordered by access */
   scandd_change_file_t *lru_last;

   /* number of changed entries */
   int num_entries;

   /* mutex lock for entries in the change list */
   kmutex_t change_list_lock;

   /* next offset to write */
   long long file_offset;

   /* hash table of changed files */
   scandd_change_file_t *change_files[SCANDD_INODE_HASH_NUMBER];
} scandd_change_list_t;
```

Figure 3.7: Data structure for caching a list of modified files

```
/* structure to save a single file name that has been changed */
typedef struct scandd_change_file
{
  /* pointer to the previous entry in the hash list */
  struct scandd_change_file *prev;

  /* pointer to the next entry in the hash list */
  struct scandd_change_file *next;

  /* pointer to the previous entry in the LRU list */
  struct scandd_change_file *lru_prev;

  /* pointer to the next entry in the LRU list */
  struct scandd_change_file *lru_next;

  /* inode number -- va_nodeid in vattr */
  u_longlong_t inode;

  /* size of memory allocated including space for file name */
  int size;

  /* length of file name including the newline character */
  int name_len;

  /* name of the file.  will be allocating the correct size for
  ** the file name when allocating memory.
  */
  char file_name[1];
} scandd_change_file_t;
```

Figure 3.8: Data structure for a modified file entry

The list of modified files is stored as a hash table, `change_files`, which is an array of pointers to `scandd_change_file_t` shown in Figure 3.8. This hash table is similar to the hash table used to store the list of monitored file systems with the exception

of the number of hash buckets used, 1024 (defined as `SCANDD_INODE_HASH_NUMBER` in the source code). The hash bucket is located using the `inode` number of the modified file (`inode` modulo 1024). Similar to the hash table for file systems monitored, the number of buckets must be a power of two for efficient hash computation. As `scandd_change_list_t` is designed to store the list of modified file names, it requires more buckets for faster lookup and hence consumes more memory than `scandd_fs_t`. To make lookups efficient and to limit the memory usage to a deterministic value, *scandd* stores only a maximum of 1024 entries at any given time in the above cache. If the addition of a new entry into the cache would increase the number of entries beyond the predefined value, 1024, *scandd* removes one entry from the cache and writes the file name corresponding to the removed entry into the file where modified file names are written for the file system. The entries in this cache are maintained in least-recently-used (LRU) order where the members `lru_first` and `lru_last` point to the 'most recent' and the 'least recent' entries in the cache. Before adding a file name into the cache, *scandd* checks if it is already present in the cache. If present, *scandd* moves the cache entry corresponding to the file name to the beginning of the LRU list. This is achieved by removing the entry from its current LRU location and adding it to the location pointed to by `lru_first`. The field, `lru_last`, points to the location from which entries are removed to make space when a new entry is added and the number of entries is already 1024. The members `num_entries` and `file_offset` specify the number of entries in the cache and the offset in the file where the next modified file name is written. Every time an entry is written to the change-list, *scandd* increments the member variable, `file_offset`, by the file name

length. For any modification in `scandd_change_list_t`, *scandd* acquires the mutex, `change_list_lock`, to achieve mutual exclusion between various kernel threads.

The structure, `scandd_change_file_t`, contains the pointers `lru_prev` and `lru_next`, which point to the previous item and the next item in the LRU cache, respectively. Similarly, the pointers `prev` and `next` point to the previous item and the next in a hash bucket if more than one modified file hashes to the same bucket. Thus, the modified files are arranged as a doubly-linked list, which allows *scandd* to do quick addition or deletion of any entry from the cache in a constant time once an entry's location in the cache has been determined. In addition, `scandd_change_file_t` contains the members: `inode` and `file_name`, which store the `inode` number of the modified file and the name of the modified file.

When a file is removed from a file system monitored by *scandd*, it is removed from the above cache entirely. The flowchart shown in Figure 3.9 shows the steps *scandd* performs during a file addition or deletion to keep the cache in sync with file system. *Scandd* uses the `inode` number of a file to verify if it is present in the cache.

As the cache size is limited, a file might have been removed from the cache to make space for another file. If a file removed from the cache is modified again, *scandd* will reinsert a new entry for it. It is in such cases that *scandd* is not able to eliminate identical file names even though the same file was modified more than once and hence, the file containing the list of modified files may have identical entries. *Scandd* makes no effort to remove these identical entries. Programs using the list of modified files generated by *scandd* must remove these identical file names before backup. Removal of identical file names can be accomplished using standard UNIX utilities such as

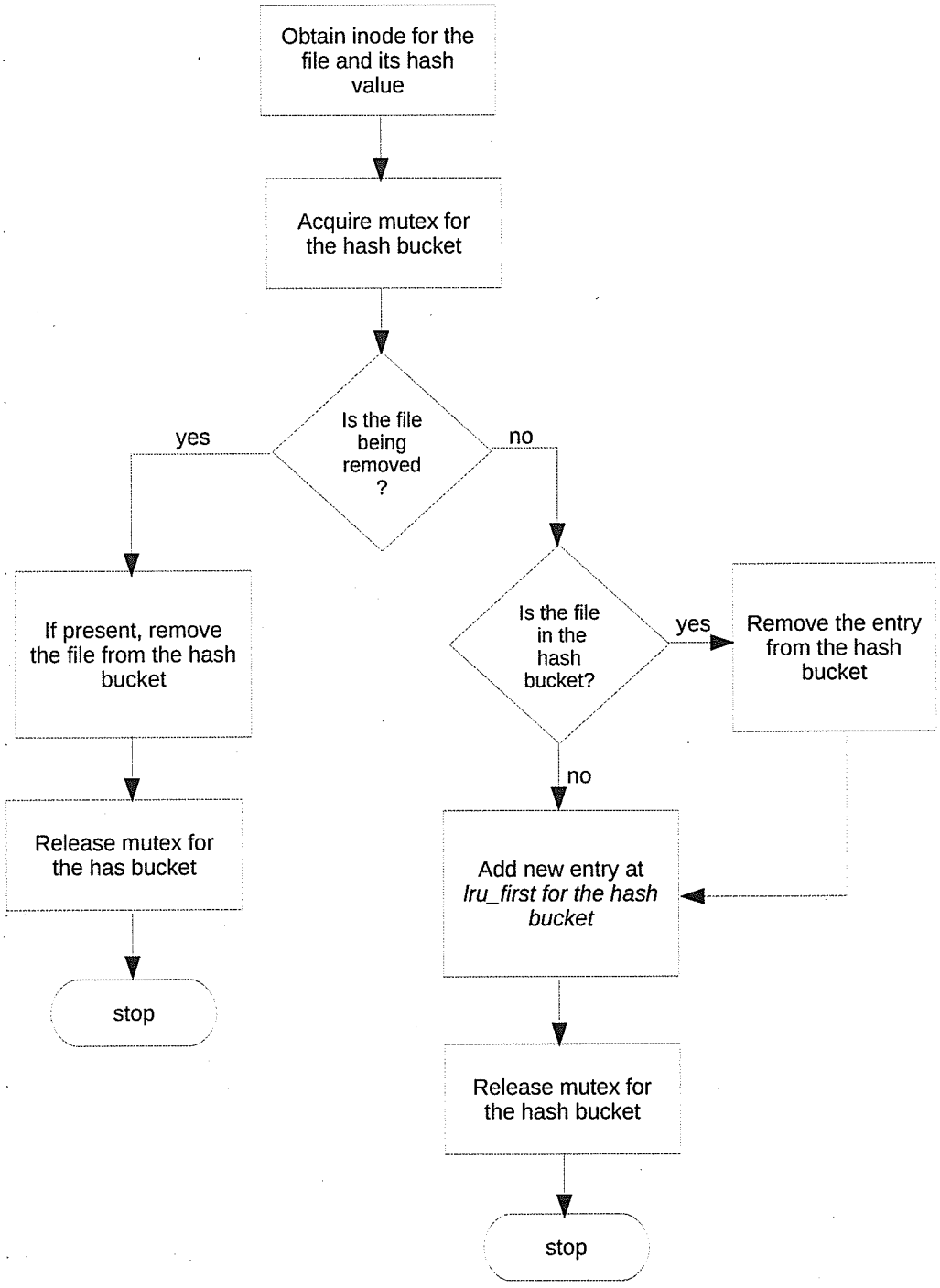

Figure 3.9: Flowchart showing addition/deletion of a file to/from the cache

`sort` and `uniq`. The utility `sort` sorts the list of changed files so that all identical file names occur together. The output of `sort` can be piped to the utility `uniq`, which in turn removes identical entries. E.g. a backup application can use '`sort -r | uniq`' to remove the identical file names. The '`-r`' option instructs the `sort` program to perform a reverse sort to generate the list of modified files in a depth-first manner as performed by the traditional *scan* code shown in Figure 1.1.

3.9 Error Conditions

During normal operations, if *scandd* encounters any error conditions, it will set an error flag; a *scan* program using *scandd* must first verify that there were no errors flagged by the driver using the *ioctl*, `SCANDD_GET_LAST_ERROR`. If there was an error, the list of modified files generated by *scandd* must be considered unreliable. In addition, the *ioctl*, `SCANDD_SYNC_CHANGE_FILES`, also returns an indication of the driver error enabling backup applications to check whether writing the modified file names itself resulted in any error conditions in the driver. If *scandd* has flagged an error in any of the above cases, the list of files generated by *scandd* cannot be used. Backup applications must therefore switch to one of the traditional methods described in Figure 1.1.

3.10 Methodology

The flowchart shown in Figure 3.10 shows the pseudo-code of an example *iscan* application. In this code, *iscan* first checks to see if *scandd* encountered any error

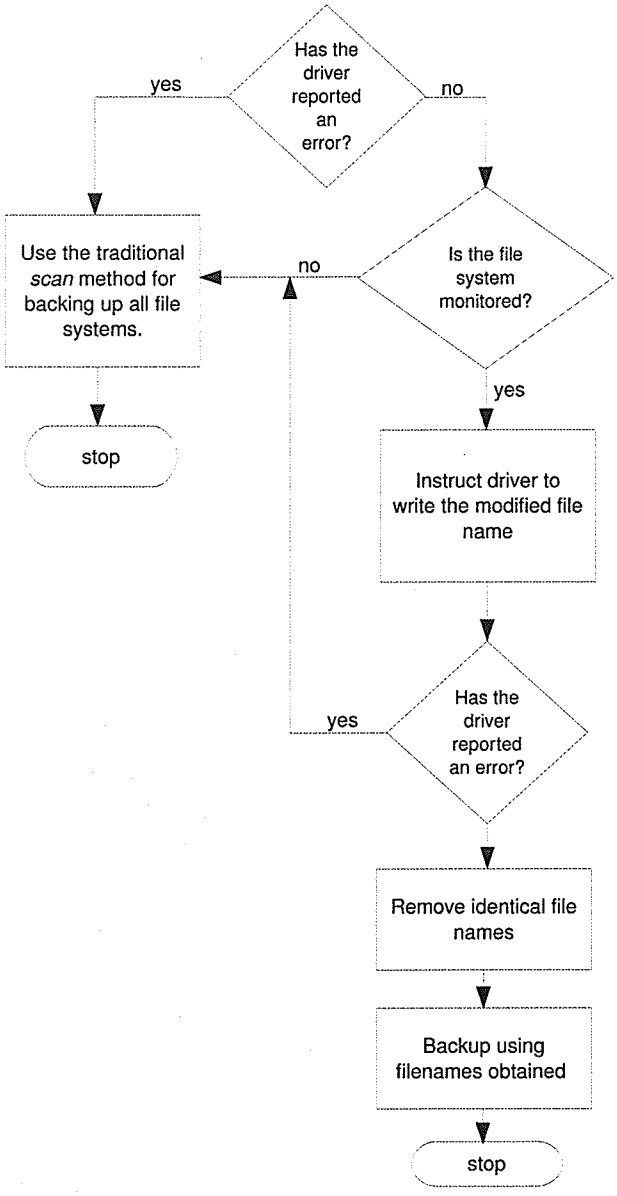

Figure 3.10: Flowchart depicting a sample *iscan*

conditions by invoking the *ioctl*, SCANDD_GET_LAST_ERROR. If *scandd* reported an error, the traditional *scan* method must be used for all file systems. If *scandd* did not report any errors, *iscan* performs the following operations for each file system. Regardless of whether the backup is an *incremental* or a *differential*, *scandd* issues the *ioctl*, SCANDD_SYNC_CHANGED_FILES that instructs the driver to write all modified file names from the cache and return the version number for the last file written. If the backup is an incremental backup, *iscan* only needs to use the latest version of the modified file. On the other hand, if the backup is a *differential* backup, *iscan* must combine all versions of the modified file list to a single file, the first version being 0 and last version as returned by the *ioctl*, SCANDD_SYNC_CHANGED_FILES. In both cases, elimination of identical file names, as described in Section 3.8, must be performed before the list of modified files can be handed off to the backup process.

3.11 *Scandd* Termination

The termination actions performed by *scandd* are somewhat the reverse of those performed during the initialization phase. With Solaris OS, a driver can be unloaded using the command '`rem_drv`' and the argument to this command specifies the driver to unload. As part of the unload procedure, the Solaris OS calls the routine, `scandd_detach`, to terminate and detach the driver from the devices its may be controlling. In the case of *scandd*, `scandd_detach` invokes `scandd_fini`, which performs the following steps:

1. Checks to see if any file systems are being monitored. If so, it returns the error, EBUSY, which indicates that the driver cannot be unloaded.

2. Restore the `vnodeops` pointers that were replaced during the initialization phase of *scandd.*

3. Free any kernel memory that was allocated for *scandd's* internal use including the memory allocated for storing the saved `vnodeops` pointers.

4. Free all mutexes that were allocated during the initialization phase.

5. Release the `vnode` for the directory specified by the configuration field, `SCANDD_TARGET_DIRECTORY` that was obtained during the initialization phase of *scandd.*

The function `scandd_detach` will succeed if and only if, all of the above steps complete successfully. Once `scandd_detach` returns success, the kernel invokes the routine `_fini` to complete the unload of the driver from memory.

3.12 Special Case

When a file is created in a directory, *scandd* must log the name of the newly created file as well as the directory in which it was created. Because, the creation of a file adds an entry to the directory, this changes the `ctime` on the directory itself. My efforts to intercept the `vnodeops` function responsible for writing the directory contents resulted in several crashes and hence I opted to log both the names of the file and the directory as part of a create. I have explained the reason for the crash in Section 5.1.1.

3.13 Driver Logs

Any errors from *scandd* are logged using the system logger. By default, all log messages from kernel modules are written to the file, `/var/adm/messages` and so are the logs from *scandd.*

3.14 Concluding Remarks

In this chapter, I have discussed the implementation details of *scandd* including its initialization and termination steps. In addition, I have detailed the *ioctls* defined by *scandd* and discussed a sample *iscan* application using these *ioctls.* The performance comparison of the traditional method and *iscan* is the topic of the next chapter.

Chapter 4

Correctness, Performance and Evaluation

For *scandd* to replace the *scan* phase in a backup, the list of modified files generated by *scandd* and that created by the *scan* program in a traditional backup must be the same. Otherwise, there is potential for files to not be backup using the method developed in this thesis. Therefore, verifying the correctness of *scandd's* implementation is an important evaluation step.

4.1 Testing Tools

To test *scandd's* functionality, I have developed several different programs. They include a program to exercise file system operation as well as a program to test *scandd's ioctls*. Some of the important test programs I used for the verification of *scandd* are:

4.1.1 `testioctls`

This program was developed to test the *ioctls* supported by *scandd*. Upon invocation, this program enumerates all ioctls supported by *scandd*. The *ioctls* supported by *scandd* were described in Section 3.6. Any individual *ioctl* from the above list may be selected for testing. This program simply reads any arguments required by the *ioctl* from `stdin` and prints the results on `stdout`.

4.1.2 `test_find`

I developed this tool to verify the correctness of *scandd* in generating the list of modified files. Although the UNIX utility, `find`, has an option `-newer` that could list files newer than a specified file, I was not able to use it for testing *scandd*. Because, the option `-newer` lists modified files based on the `mtime` only. Whereas, the *scan* phase in a backup must list files modified based on both `mtime` and `ctime` of a file. The fields, `mtime` and `ctime` refer to the modification time and status change time of a file, respectively. When a file's contents have been modified, the system updates the `mtime`. On the other hand, when a file's permissions have changed, the system updates the `ctime`.

Program, `test_find`, takes two arguments; the first argument can be one of `'-ctime'`, `'-mtime'` or `'-both'` which specifies what time on a file should be used to verify if it newer and the second argument specifies the file whose `mtime` is to be used as the reference time for comparing. The file system to list for modified files is read from the standard input. This program employs a depth-first search of the file system specified and uses the `stat()` system call to obtain the `mtime` and `ctime` of

the files or directories it encounters. Depending on the first option to this program, one or both of the times associated with each file are compared with the `mtime` on the reference file (second argument). This program lists all files and directories whose `mtime/ctime` is more recent compared to the reference file.

4.2 Verification of Correctness

To verify the correctness of *scandd's* implementation, I used the tools, `test_find` and `testioctls`, just described. Steps in the verification process were as follows: at the start of the verification process, use `testioctls` to monitor one or more file system(s). Next, instruct *scandd* to synchronize its cache of modified files for the file systems of interest using the *ioctl*, `SCANDD_SYNC_CHANGED_FILES`. The above *ioctl* creates a new version of the file containing the modified files and outputs its version number (let's assume this version to be 1). We use the time on this file as the reference time for listing the modified files by providing it as the second argument to the `test_find` program. After some number of file modifications and namespace changes on the file system, the *ioctl*, `SCANDD_SYNC_CHANGED_FILES`, is issued again to create a newer version of the list of modified files (version 2). If *scandd* generates the list of modified files correctly, the list of modified files generated by `test_find` and the latest version (version 2) of the modified files generated by *scandd* must be the same.

In my testing with `testioctls` and `test_find`, there are some cases where I found differences between the list generated by *scandd* and `test_find`. Section 4.3 explains the reasons for the differences and its potential effect on backups and restores.

4.3 Can *Scandd* Miss Files?

Hard links and renaming of directories poses a significant technical challenge for *scandd* resulting in some scenarios where the list generated by *scandd* will differ from the list of files generated by a traditional *scan* program. In addition, *scandd* omits special files such as block and character device files, and named pipes from its list of modified files. This is because, the Solaris kernel does not provide the pathname for the above file types when they are modified.

4.3.1 Hard Links

In comparison with symbolic links, hard links are implemented differently in UNIX. The main noticeable difference to a user is that hard links cannot cross file system boundaries where as symbolic links can. Apart from the above, there is a major implementation difference between hard links and symbolic links that poses an implementation difficulty for *scandd* in some specific cases. Within a file system, all hard-linked files have the same `inode` number (which may be verified by the '`-i`' option to the UNIX '`ls`' command) whereas symbolic links to a file or directory have different `inode` numbers.

If a file has many hard links and if one of the hard-linked files is removed, the link count (representation of the number of hard links to a file) on the file changes, which results in a change in the `ctime` of the file. As the `ctime` change is effected through some internal file system call resulting from the invocation of the remove operation, *scandd* cannot intercept the `ctime` change. As a result, there will be a difference between the list of modified files generated by the traditional *scan* program

and *scandd* if some hard-linked files are removed. Since no file data was modified, however, backup using the list generated by *scandd* will not cause any data loss. Moreover, the `ctime` change is irrelevant as the restore program can never set the `ctime` to the original value. This is because, a restored file will always have the last write time as its `ctime`.

If a file has many hard links and if only one of the files is modified, a traditional *scan* program will list all of the hard-linked files as modified whereas *scandd* will list only the actual file that was modified. In essence, a traditional backup program will backup multiple copies of a hard-linked file whereas backup using the list of files generated by *scandd* will backup only the files that were actually modified regardless of whether the files were hard-linked or not. Since traditional backup saves multiple copies of the same file, restore from a traditional backup will restore multiple copies which are no longer hard-linked. This is because, in comparison to symbolic links, backup programs cannot determine the link information of a hard-linked file. *Scandd* too is not immune to the side effects of hard-linked files. Restore from a backup with the list generated by *scandd* may also restore multiple regular files instead of hard-linked files.

In addition, there is one case where *scandd* can miss the modification of a hard-linked file thereby causing possible data loss. This case occurs when a hard-linked file is modified and deleted immediately after its modification. As the file was removed, *scandd* deletes any reference to the modified hard-linked file from its cache. As all hard-linked files share the same `inode`, the modification will be visible through the other hard links, which enables a traditional *scan* program to list them. The

current implementation of *scandd* has no mechanism for avoiding this situation. To prevent any data loss, *scandd* will flag an error on the specific file system when any file with link count greater than one is removed; this error will be visible to the application invoking the *ioctl* `SCANDD_SYNC_CHANGED_FILES`. As shown in the flowchart in Figure 3.10, a *scan* using *scandd* must check for any errors on the file system after an application issued the above *ioctl*. If *scandd* has flagged an error on the file system, *scan* must be performed using the traditional *scan* method. Switching to the traditional method prevents any data loss in this scenario.

4.3.2 Rename

The issue with rename is that the `vnode` for the renamed file/directory contains the pathname before the rename was performed. This is a Solaris bug and there are many references to this bug on the Internet and also at the official site where Solaris updates are available, `www.sunsolve.sun.com`. Sun microSystems [24] has issued an update to resolve this issue. However, the update addresses only rename of files, not directories. Therefore, *scandd* is still vulnerable to this issue for renamed directories and its children. Since *scandd* cannot identify whether a file/directory being modified is the child of a renamed directory or not, it performs the following tests to verify if a file/directory was either renamed or if it belonged to a renamed directory:

- *Scandd* performs a lookup on the pathname obtained from the `vnode` of the modified file/directory.
- If the lookup is successful, the pathname corresponds to an existing file. However, it may be possible that a file or a directory was renamed and another

file/directory took its previous name and that the modification request is for the new file/directory. To verify this condition, *scandd* compares the original `vnodes` and the `vnode` obtained from lookup. If the `vnodes` are different, then the modification request is for a file/directory that took the name of a renamed file/directory. In this case, *scandd* sets an error for the file system specifying that the list of modified files is unreliable, which can be detected by the *scan* program. On the other hand, if the `vnodes` are the same, the pathname is simply added to the list of modified files.

- If the lookup fails, the file/directory is the child of a renamed directory. In this case too, *scandd* sets an error to flag to indicate that the list of modified files is unreliable.

It is also possible that a user only renamed a directory but did not modify any files under the renamed directory. If a traditional backup was performed in such a situation, it will save all files under the renamed directory. This is because, a system restored from the recent backup must bring back the file systems in the exact same state as at the time of the backup. Although the current implementation of *scandd* generates the name of the renamed directory into the list of modified files, the backup phase cannot determine whether a directory was renamed or not. Therefore, the current implementation of *scandd* and its associated test programs do not handle renamed directories as traditional backup programs do. Section 5.1 discusses some possible enhancements to *scandd* to handle renamed directories so that the backup represents a true image of the system when restored.

4.3.3 Special Files

When special files such as character or block devices, and named pipes are modified, the pathname component in the `vnode` is `NULL`. Therefore, *scandd* cannot generate a pathname for such modified files. Instead, *scandd* flags an error on the file system to indicate the unreliability of the list of modified files.

4.4 Performance Evaluation

This section lists the various performance evaluation steps that I performed to evaluate *scandd's* performance. Some of the initial tests mentioned in this section do not evaluate the performance of *scandd* when used as a replacement for the traditional *scan* method, *scandd's* primary purpose. However, such tests are of primary importance as these tests measure the impact of *scandd* on the performance of the system. This is because, *scandd* is a device driver and hence can adversely affect the system performance if badly designed and/or implemented. Tests that measure the performance of file system *scan* are listed later in this section.

4.4.1 System Information

The characteristics of the system that I used to develop *scandd* and measure its performance are as follows:

Model: SunBlade 150.

CPU: UltraSPARC-IIe 550 MHz.

Number of CPUs: 1 CPU.

Memory: 512 MB.

Swap: 1 GB.

Disks: Two IDE drives each of 74.5 GB capacity of which, the first one was solely used for all the system partitions and home directories. Whereas, the second drive was reserved solely for testing *scandd*.

Update Level: System was up-to-date with all the latest updates as of May, 2009.

4.4.2 *Scandd's* Impact on Regular File Operations

I designed the first test in the performance evaluation series to measure the impact of *scandd* on regular file/directory operations: create and mkdir. Table 4.1 shows the time taken to create many files under a directory for three different scenarios: *scandd* not loaded, *scandd* loaded but no file system monitored and *scandd* loaded and monitoring the file system of interest. Figure 4.1 shows a plot of these three cases. In each of the three cases, I ran the tests in 10 steps, starting with the creation of 100,000 files, then incrementing the number of file created by 100,000 to create 200,000 files in the second step, and so on.

From Table 4.1, it can be seen that the time taken for the test is almost the same between Cases I and II, which can be verified from the overlapping plots in Figure 4.1. The purpose of considering Case II is to measure the impact of *scandd* when loaded with no file systems monitored. As mentioned in Section 3.5, when loaded, *scandd* intercepts file system calls and hence this case measures any performance degradation

No. of Files	*Scandd* not loaded Case I	*Scandd* loaded not monitoring Case II	*Scandd* loaded and monitoring Case III
(× 100,000)	Time in Seconds		
1	25.4	29.2	33.4
2	52.0	51.8	65.8
3	77.7	77.9	101.2
4	104.6	106.0	135.7
5	131.1	133.3	167.7
6	159.3	161.1	202.5
7	185.6	185.8	235.8
8	212.5	216.0	271.0
9	238.4	240.6	305.5
10	269.3	267.8	341.4

Table 4.1: Effect of *scandd* on file creation

on file systems that are not monitored by *scandd* but as a result of *scandd* intercepting the file system calls. If a file system is not monitored, the only operation the intercepted call performs is to verify whether the `vnode` is part of a monitored file system. As can be seen from Table 4.1 and from Figure 4.1, this operation causes negligible effect on the performance on an unmonitored file system. Some anomalies can be noticed in case of rows 2 and 10, which I attribute to the caching of the test program by the OS. This may have happened when I ran the same test program back to back.

Case III shows the time taken for creating files in a monitored file system. In this case, once *scandd* determines that the `vnode` is part of a monitored file system, the file name is added to the list of modified files as described in the flowchart shown in Figure 3.9. This involves operations such as allocating memory to save the file name, checking whether the file name is already part of the 'Identical File Name Elimina-

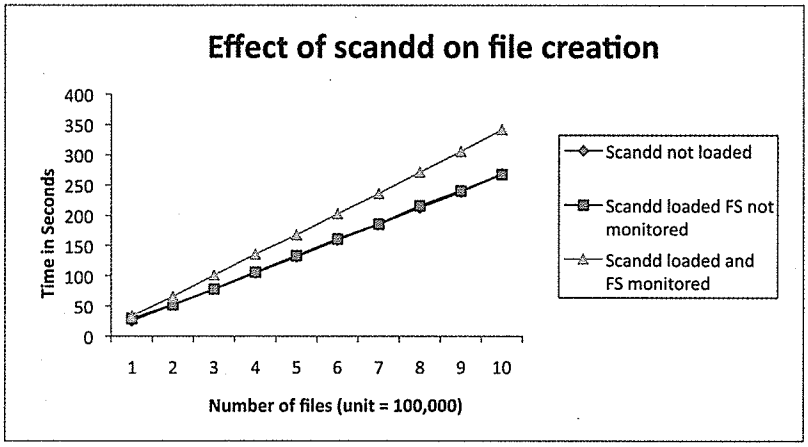

Figure 4.1: Plot of file creation times

tion Cache' and if not, adding the file name to the above cache. The performance degradation on a monitored file system varies between 14% for 100,000 files and 27% for 1 million files. One can easily visualize the increasing performance degradation as the number of files created increases from Figure 4.1.

The variation in the degradation between the smallest and the largest times could be due to the following factors. As mentioned in Section 3.8, memory is allocated from the kernel to cache the file names and is later freed when the file names are written to the file containing the modified file names. This may cause the kernel memory to be fragmented and hence it may be that a defragmentation operation may be scheduled by the kernel which may steal some CPU cycles. Writing modified file names will also result in the consumption of memory pages and later when they are flushed to the disk, will result in disk I/O. As the size of the file containing the modified files increases, the kernel daemon *fsflush* will become more active and steal CPU cycles to synchronize cached pages to the disk. To give an example, the size of the file containing the modified filenames when 100,000 files were created was 4 MB,

which results in about 14% degradation in the file system performance and in the case of 1 million files created, the size of the file containing the modified file names was 40 MB, resulting in 27% degradation of the file system performance.

I used the results from this test primarily to fine tune *scandd* so that its impact on the system is minimal. Using the results from this test, I was able to use the best compiler optimization flags and inline certain code fragments to get the best results. Comparing the results from various runs, I decided to use optimization level 3 (`-O3`) for compiling the driver sources and inline the function that returns the file system structure (described in Figure 3.6) given a `vnode`. This is because, the function to return the file system structure is invoked from the intercepted file system calls and must be very efficient to reduce its overhead since it is called frequently.

4.4.3 Improvement on Standard File Systems

In this section, I focus on the improvements *scandd* provides over traditional *scan* on standard file systems. Solaris requires five important directory trees for its operation. They are: `/`, `/usr`, `/var`, `/opt` and `/tmp`. Traditionally, they are configured as separate file systems although, the newer Solaris installations tend to put all the above directories under one file system. Of the above five directories, the standard Solaris 10 installation constructs the `/tmp` directory tree off the swap partition as it provides a file system whose contents are not retained across reboots and hence is not important to be backed up to secondary storage. Although, files in the above mentioned directories change very little (with the exception of `/tmp`), backups include the above directories as they are vital for the correct operation of

the system.

To demonstrate the performance improvements using the methods developed in this thesis on the above file systems, I have tabulated the number of files and directories in them and the performance metrics I have used to compare the two methods. Table 4.2 details the performance figures using traditional *scan* and *scan* using *scandd*.

File System Name	Traditional *Scan*					Using *Scandd*
	No. of Dirs (× 1,000)	No. of Files (× 1,000)	*Scan* Time (Sec)	System Calls (× 1,000/Sec)	Context Switches (Per Sec)	*Scan* Time (Sec)
/	1.5	3.7	3	5.1	331	≈0
/usr	8.6	126.7	33	5.8	737	≈1
/var	11.9	16.8	38	2.8	737	≈0
/opt	7.1	120.7	33	5.3	763	≈0

Table 4.2: Comparison of time between traditional *scan* and *scandd*

I measured the number of system calls and the number of context switches using the Solaris-provided commands: `trapstat` [25] and `vmstat` [26]. The command, `trapstat`, reports the traps occurring on the system. A trap is a mechanism provided by modern CPUs to switch from one operating mode to another. Operating systems use the trap instruction to switch from user mode to kernel mode while executing system calls. Traps are used for the implementation of many other operating system features, system calls being only one of them. The command, `trapstat`, provides a convenient way to run a given command and report all traps that occurred on the system resulting from the command. Similarly, the command, `vmstat`, reports statistics on virtual memory such as page faults, context switches, etc.

To analyze the impact of traditional *scan* on the system, I measured the system

calls per second and the number of context switches per second on the same system when *scan* was not running. I have provided the results in Table 4.3. Comparing the values from Tables 4.2 and 4.3, it can seen that the number of system calls per second when traditional *scan* is run is about 30 times than that of a standalone system (when *scan* was not run). Similarly, the number of context switches is about 3.5 times that of a standalone system without *scandd.* The performance figures for the / file system represent an outlier as the number of files and directories is too small to make a good average reading of the above parameters.

System Calls (Per Sec)	Context Switches (Per Sec)
190	206

Table 4.3: System calls and the context switches measured on a standalone system

It is interesting to note that the time taken by traditional *scan* on the `/var` file system is comparable to that of `/usr` and `/opt` file systems even though the number of files and directories combined under `/var` is only roughly 22% of the number of files in `/usr` or `/opt` directories. The interesting fact about the `/var` file system is, it has many more directories than the other file systems. In addition, the directory structure under `/var` is deeper, the maximum depth being 18. As mentioned in Section 2.1, Solaris maintains a DNLC cache for optimizing lookup speed. In the case of the `/var` file system, I presume that the increased time for performing *scan* is due to the fact that the DNLC cache hit ratio drops as the number of directories scanned increases. In the case of `/usr` and `/opt`, lookup is performed in the same

directory for a longer duration as the ratio of files to directories is higher in these file systems. Whereas, lookup on `/var` moves from one directory to another directory at a much higher pace putting higher demand on the operating system services to perform more page-ins and populate more DNLC entries. As lookup moves from one directory to another, DNLC entries need to be populated for the new directory encountered and in addition, the system needs to read the disk blocks containing the information from the new directory. An earlier study by Tamches and Miller [27] has confirmed the same results on DNLC misses and lookup performance when the number of files/directories are large.

In the case of *scan* using *scandd*, all file systems except `/usr` take less than 1 second to process. This is because these file systems have very minimal changes over a long period. Since, the granularity of time measurement was in seconds, I was not able capture any small amount of time *scan* using *scandd* might have taken on these file systems. Therefore, the time taken on these file systems have been approximated to the nearest second in Table 4.2. Since *scan* using *scandd* does not invoke `stat()` system calls, the parameters I used for gauging performance of traditional *scan*: context switches and system calls are not valid performance metrics in this case.

As can be seen from the preceding discussion, standard file systems such as `/`, `/usr`, `/var` and `/opt` will benefit significantly from using the method developed in this thesis. Similarly, there may be other file systems where the number of files modified is very few compared to the total number of files in the file system. The next section illustrates the improvement achieved on file systems with many files in

a directory.

4.4.4 Improvement on File Systems with Many Files in one Directory

To validate the performance improvements on file systems with many files, I created a file system of size 63 GB and created a folder with files under it. The test was carried out starting with 1 million files until 5 million files in increments of 1 million. At each iteration, the previous files were removed and new ones were created in their place. Since the removal of files using `rm` took an extremely long time, I resorted to recreating the file system between each iteration (removing 5 million files took in excess of 4 hours). At each iteration, I measured the time for traditional *scan* and *scan* using *scandd*. In addition, I evaluated the number of context switches and the number of system calls per second to validate the system overhead when both *scan* methods were running. Table 4.4 shows the results from the test. The baseline context switches and system calls per second are already shown in Table 4.3. Figure 4.2 shows a graph of the time taken for traditional *scan* and *scan* using *scandd*, Figure 4.3 shows a graph of the number of context switches incurred when traditional *scan* and *scan* using *scandd* were run, and Figure 4.4 shows a graph of the the number of system calls executed in each of these cases.

In the case of *scandd*, it does not matter whether the files are part of a single directory or whether they are scattered among different directories in a file system. Therefore, I would argue that, with respect to *scandd*, this test simulates a large file system (with many millions of files) in which 1 million to 5 million files are modified.

No. of Files (× 1,000,000)	Traditional *Scan*			Using *Scandd*		
	Scan Time (Sec)	Context Switches (Per Sec)	System Calls (× 1,000/Sec)	*Scan* Time (Sec)	Context Switches (Per Sec)	System Calls (× 1,000/Sec)
1	80	813	12.9	30	320	0.7
2	173	767	11.9	62	299	0.7
3	241	675	12.8	98	253	0.6
4	377	764	11.0	137	277	0.7
5	1910	500	2.8	178	302	0.6

Table 4.4: Comparison of *scan* times when many files in one directory were modified

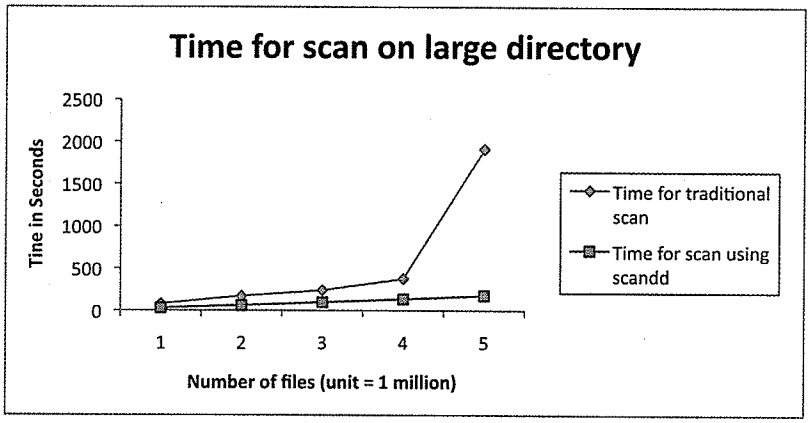

Figure 4.2: Plot of *scan* times for many files in one directory

As can be seen from Table 4.4 and Figure 4.2, the time for traditional *scan* exhibits a non-linear increase in time beyond 4 million files. On the other hand, the time for *scan* using *scandd* remains linear as do the number of context switches and the number of system calls. In the case of traditional *scan*, it can be seen that the number of context switches and the number of system calls per second dropped progressively as the number of files increased. The drop in the number of system calls when the number of files increased is because, `stat()` on files now takes longer, an argument I used for a different enumeration method in Section 1.1. As the time for an individual

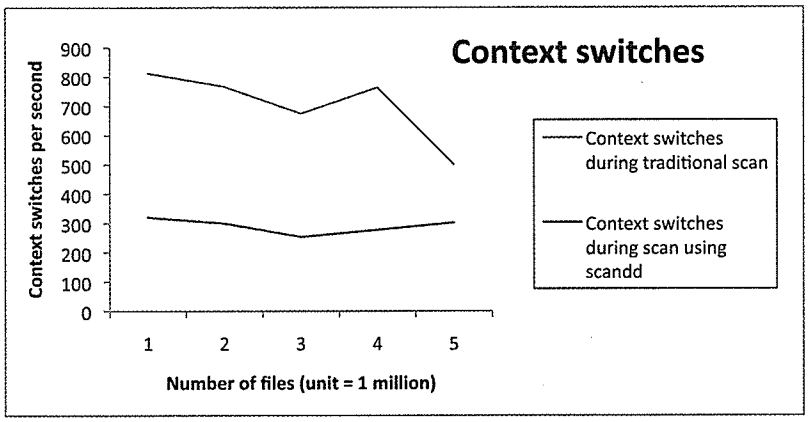

Figure 4.3: Plot of the number of context switches during *scan* for many files in one directory

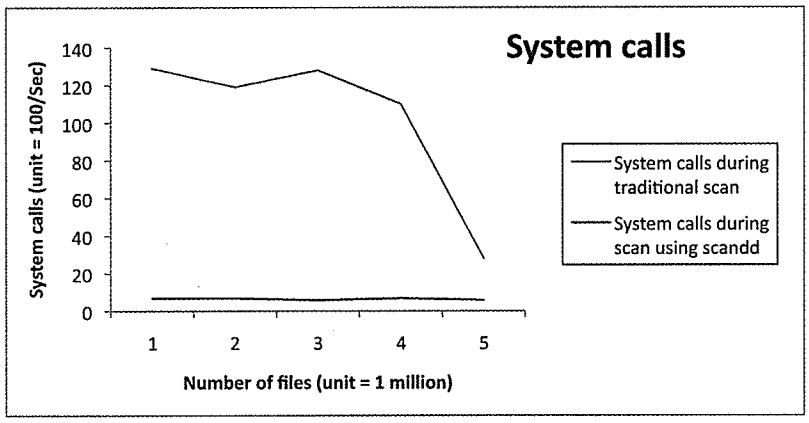

Figure 4.4: Plot of the number of system calls during *scan* for many files in one directory

`stat()` increased, the number of context switches have reduced because the file system has to spend more time processing an individual `stat()` system call. Moreover, as the directory search space is larger, more directory pages will be required in memory introducing more page faults and page-ins to be processed by the kernel. As the number of page faults and page-ins increases, the performance of other applications running on the system may suffer (depending, of course, on the memory size of the system).

Using the `-p` option of `vmstat`, I was able check the paging activity on the system while *scan* was running. The `-p` option provides detailed information on the paging activity of the system. Using this option, I was able to note that the parameter, `fpi`, denoting the number of 'file system page-ins' was consistently larger – hovering in the range of 5,000 to 8,000 page-ins (an idle system has occasional page-ins less than 100). If there are more applications that require file system pages, traditional *scan* will compete for file system pages and will result in lower throughput for those applications. The large number of page-ins during *scan* is because the directory size, as indicated by the `ls` command, is large (about 150 MB for a directory with 5 million files each files having names of 21 bytes in length) and the file system tries to keep these pages in memory once they are read. However, some of the pages may need to be evicted to make room for other pages from different applications or the *scan* itself. Therefore, the performance of *scan* will depend greatly on the amount of memory on the system and the number of applications competing for memory on the system. The extra page-ins introduced by traditional *scan* will thus drastically affect the throughput of other application running on the system.

Apart from the performance figures, the user experience on the system also confirms that the system throughput has dropped drastically. While traditional *scan* was running, simple commands such as `ls` and `vi` took much longer than usual. In some cases, I had to wait as much as 10 seconds for my interactive commands to be echoed on the screen.

4.4.5 Improvement on File Systems with Deep Directories

Till now, most of the results presented are on a shallow file systems with many files in them. Table 4.5 illustrates the time taken by traditional *scan* on a file system that contains a directory with 5 million files at various depths. In all of the test cases, the number of files is 5 million.

Directory Depth	Pathname Length	*Scan* Time (Minutes)
2	18	10.5
42	120	13.3
112	260	22.6
230	500	43.8
490	1020	132

Table 4.5: Comparison of traditional *scan* time for a directory at various depths

As can be seen from Table 4.5, time for traditional *scan* on the same directory containing the same number of files increases as the depth of the directory increases. In the most extreme case, *scan* of 5 million files with a directory depth of 490 took 2 hours and 12 minutes. It can be inferred that a system containing some file systems with deep directories with many files may take an extremely long time for traditional *scan*.

In the case of *scan* using *scandd*, the time taken depends only on the number of files modified. Even if all of the 5 million files were modified, the time for the deepest case (490 directories deep) is half of the time taken by traditional *scan*, 1 hour and 8 minutes. Virtually all of this time was consumed by the `sort` program. The user experience on the system was pleasant as compared to the user experience

when traditional *scan* was running where long delays were noticed for interactive commands run from a terminal. The low impact of *scandd* on the system can be further verified from the output of the UNIX `time` command, which outputs the real, user and sys times for a command. In the case of *scan* using *scandd*, the real time was 68.5 minutes, the user time was 52 minutes and the sys time was 6.5 minutes. From the above figure, the time spent is system call was only 10% of the total time.

An interesting observation I made when performing this test is that *scan* took a very long time because the `stat()-s` were performed with the full pathname. In this case, the kernel has to traverse many levels in the pathname. However, if the traditional *scan* program were to be rewritten such that it changed its current directory to the directory being scanned and used relative pathnames for `stat()`, the time taken for *scan* could be drastically reduced! For the deepest level, 490, the modified *scan* program took only 43 minutes instead of 132 minutes!

4.4.6 Improvement on File Systems with Many Directories and Files

The idea behind the test in this section is to create a file system similar to those that exist on large servers. In each run of the test, the number of files created is fixed, 6 million, where as the number of directories varies from 10,000 to 50,000 in increments of 10,000. The maximum directory depth is set at 20 in all runs. A depth of 20 was selected because, most file systems average to a depth of 8 as per a study of file system contents performed by Bolosky et al. [20]. That study was for file systems on Windows systems and, in the absence of any other study detailing file system

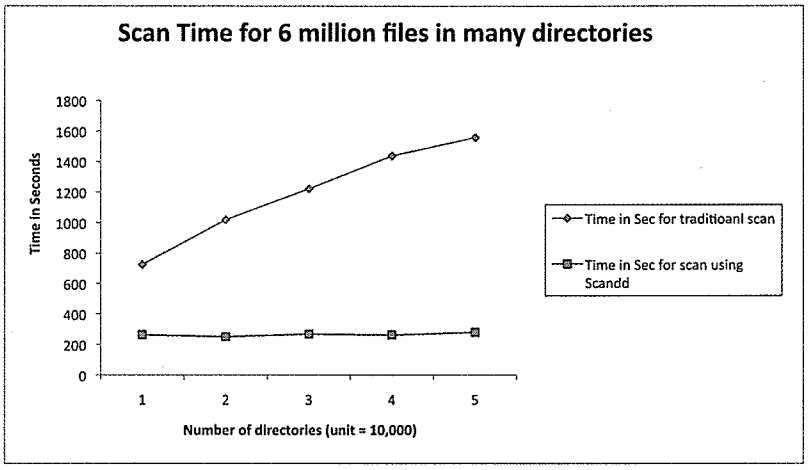

Figure 4.5: Plot of *scan* times for many files in many directories

contents and from what I have observed on Solaris system, I conservatively chose the value of 20 for the maximum depth in the test (the maximum depth of `/var` file system was 18 on my test system.

No. of Directories (× 10,000)	Traditional *Scan*			Using *Scandd*		
	Scan Time (Min)	Context Switches (Per Sec)	System Calls (× 1,000/Sec)	*Scan* Time (Min)	Context Switches (Per Sec)	System Calls (× 1,000/Sec)
1	12.1	493	8.6	4.4	316	0.6
2	17	427	6.2	4.2	338	0.6
3	20.4	417	5.1	4.5	360	0.5
4	24	404	4.4	4.4	313	0.6
5	26	410	4.1	4.7	357	0.5

Table 4.6: Comparison of *scan* times on file system with many directories

Table 4.6 shows the time taken for traditional *scan* and *scan* using *scandd* and Figure 4.5 shows a graph of the time taken in each of these cases. In addition, Figure 4.6 shows a graph of the number of context switches and Figure 4.7 shows a

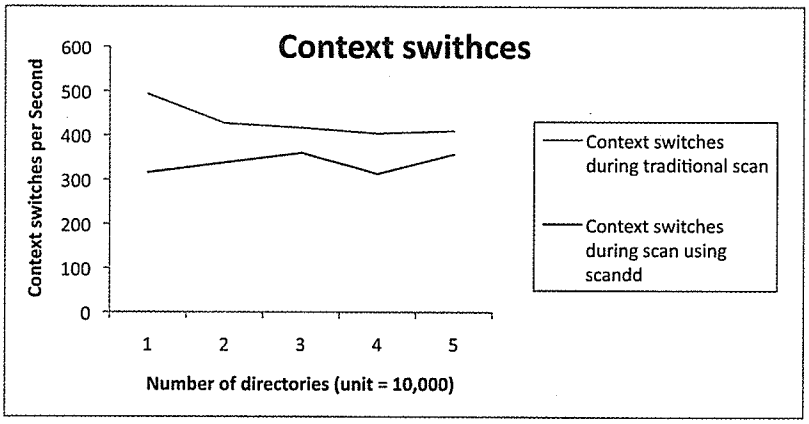

Figure 4.6: Plot of the number of context switches during *scan* for many files in many directories

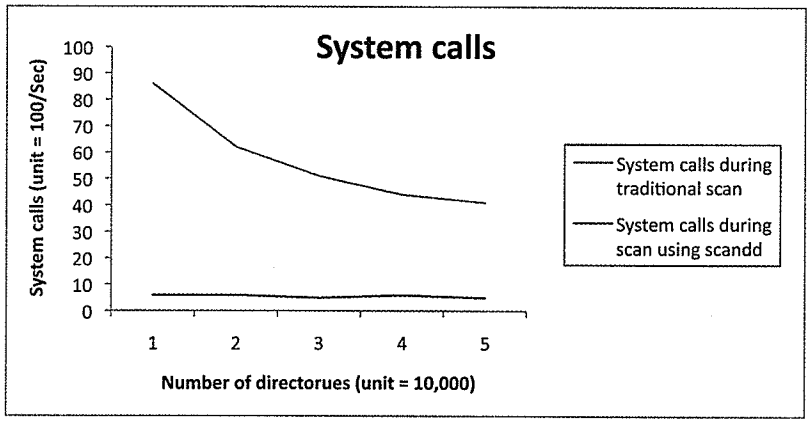

Figure 4.7: Plot of the number of system calls during *scan* for many files in many directories

graph of the number of system calls executed during the execution of traditional *scan* and *scan* using *scandd*, respectively. As can be seen from the table, the time taken for traditional *scan* increases progressively as the number of directories increases. Moreover, the number of system calls is reduced as the number of directories increases. As discussed in Section 4.4.5, this is because, execution of a system calls now puts higher resource requirements on the kernel. Contrasting this with *scan* using *scandd*, it can be seen that the number of context switches and the number of system calls

are relatively low. In addition, the time taken for *scan* in this case is almost constant.

Comparing the time for traditional *scan* in this case with the tests performed in Section 4.4.4, it can be seen that, even with 1 million fewer files traditional *scan* took longer if all files were in one directory. Extrapolating from the results on the `/var` directory in Section 4.4.3, it can be surmised that the current test should have taken about 50 times longer than the time taken for *scan* on `/var`, 33 seconds, which was the case.

4.5 When does *Scandd* Perform Worse than Traditional *Scan*?

As mentioned in Section 4.4.5, slight modification of the traditional *scan* program made it faster than *scan* using the list from *scandd* for the test case with the deepest level. Although, this is for a case where all files in a file system were modified, it reveals the conditions under which, traditional *scan* may be faster. Similarly, we can visualize many other situations in which, traditional *scan* may be faster. These include:

- The list of modified files generated by *scandd* being very large so that removing duplicates from this list takes longer than traditional *scan*.
- The number of files modified in a file system are many and hence the cache for identical file name elimination cannot eliminate many of the duplicates. When this happens, the number of entries in the list of modified files will be much greater than the number of files in a file system.

4.6 Summary of Results

From the tests conducted on traditional *scan* and *scan* using *scandd*, I conclude the following. As the number of files in a file system increases, the time taken for traditional backup will increase as the time for individual `stat()` system calls will add up. Moreover, parameters such as the number of files per directory will also affect the performance of the *scan* phase in a traditional backup in a number of ways. For example, crowding more than 5 million files in a single directory may not be the best thing to do if the same file system needs to be backed up regularly. In such cases, better results can be obtained if applications were to spread the files into multiple directories. (E.g., considering Table 4.4, a *scan* of 5 million files in one directory took 31 minutes but, 6 million files scattered among 10,000 directories took only 12 minutes to *scan*.) Similarly, deep directories can cause traditional backups to take longer than the time taken if the directories were shallow as seen in Table 4.5. Another observation is that as the directories scanned get larger, so do the system resources consumed by *scan*. If *scan* has to compete for system resources with other applications running on the system, the throughput of these applications will be reduced depending on the size of the directories scanned.

Chapter 5

Future Work and Conclusions

5.1 Future Work

In this chapter, I discuss the future of *scandd* as an efficient file system scanner for backup applications. By now, *scandd* has evolved from an idea to a Master's Thesis. However, there are some limitations in *scandd* that must be surpassed to make it more usable and efficient. Moreover, I do see a possibility for *scandd* to be part of future commercial file system backup applications. Since the emphasis of a Master's Thesis and a commercial product are very different, I present them as two separate subsections.

5.1.1 *Scandd* as a Research Thesis

The current implementation of *scandd* is by no means complete. Some of the shortcomings of *scandd* are due to some limitations in the Solaris OS while some others are due to items that came to light after the implementation phase of *scandd*

(rename of directory). In addition, there are some items that I chose not to tackle in this initial implementation of *scandd*. I have already discussed the limitations of the Solaris OS in Sections 4.3.2 and 4.3.3. Since *scandd* cannot work around these OS issues, I will not discuss them again.

Although there are many different file system types available on Solaris, the current implementation of *scandd* intercepts file operations only from the *UFS* file system. As mentioned in Section 3.8, *scandd* requires the `inode` number of a file for identical file name elimination and the function call needed to obtain the `inode` of a file on Solaris, `fop_getattr`, crashes the system due to recursive call on a read-write lock in the `inode` of the file. Initially, I was perplexed by this unexpected crash. However, the availability of Solaris source files at the OpenSolaris site [28] helped me solve the mystery. After browsing the kernel sources, I was able to deduce that every `inode` has a read-write lock to serialize write and setattr operations on the file. The crash was the result of `fop_getattr`, introduced by *scandd*, trying to acquire a read lock on the file before a write lock on the same file was released. The above crash happened in the initial implementation of *scandd* as mentioned in Section 3.12. Since then, I have modified *scandd* to not intercept the `vop_putpage` calls and to use the macros defined for the *UFS* file system to obtain the `inode` number of a file without invoking the `fop_getattr` call.

The *ioctl* structure specified in Section 3.6.3 lacks a field to indicate to the *scan* program the time when monitoring for a file system was initiated. The *scan* program could use this new field to compare the time of the last incremental/differential/full backup and make additional determination as to whether *scandd* can be used for

listing files for backup. If monitoring was started after the previous backup, there is a possibility *scandd* may have missed some of the modified files. The flowchart shown in Figure 3.10 needs modification for checking this new field.

The hash table used for maintaining the list of file systems described in Section 3.7 is limited to sixteen buckets. Although this is suitable for most systems, it would be ideal if the number of hash buckets were configurable through the driver configuration file, `/etc/scandd.conf`, for systems with many file systems that need to be monitored. The driver would, of course, have to be reloaded for any configuration files changes to take effect.

Similarly, the number of buckets in the hash table for identical file name elimination mentioned in Section 3.8 should also be configurable. The current value of 1024 is adequate for most purposes and increasing this value will result in *scandd* consuming more kernel memory as more instances of the structure described in Figure 3.8 will be allocated. Although the number of such structures allocated depends on the number of file modifications happening on a monitored file system, the above number mandates the upper limit on the number of structures allocated. Making the number of buckets configurable provides a way to reduce the memory footprint of *scandd*, if required.

There is no mechanism to change the logging level of *scandd* in the current implementation. To change the logging level, *scandd* has to be recompiled and reloaded. It would be much easier if the logging level were also configurable. However, there may be situations where reloading the driver may not be possible and in that case, an *ioctl* to modify the logging level is highly desirable.

5.1.2 *Scandd* as a Backup Product

As a component in a backup product, *scandd* must provide the following functionalities in addition to the items mentions in Section 5.1.1:

- Provide the exact image of the system for restoring only the files that existed on the system.
- Handling of renamed directories and its descendants.

To provide the above functionality, an implementation using *scandd* must move away from writing into files directly. Rather, the implementation that would be ideal is to have a producer-consumer setup between *scandd* and another user process. Instead of writing modified file names into files, *scandd* would then send records that consist of various information about modified files to the user process. The function of the user process would be to wait for the availability of records from the driver and populate a database with the received information. *Scandd* would have to provide a new *ioctl* for the user process to obtain an individual record. Having a database will enhance the functionality by supporting different queries based on the modification time of a file. In such a scenario, for every modified file, *scandd* would have to send the following items:

- File/Directory Name: Pathname of the modified/deleted/created file
- Inode: Inode of the entity modified
- File System Identifier: this can be the combination of `major` and `minor` number of the device the file system is mounted from.

- File Type: whether file, directory, character device, block device, pipe etc.
- Modification Times: `mtime` and `ctime` of the file/directory
- Operation on the File: creation/deletion/rename
- Prior Name: applicable only in case of a rename

The cache mentioned in Section 3.8 can still be retained in *scandd* as it will reduce the amount of updates coming into the user-level process if some files in the monitored file system are being continuously updated.

For creation, the user-level process adds an entry for the file/directory into the database and similarly, remove the entry for the file/directory for a removal. For all other cases: writes, change of permission, etc., an entry must be inserted into the database if not already present. However, if a modified file name is already present in the database, the modification times must be updated to reflect the recent file modification time. In the case of a renamed directory, the user-level process can correct the pathname of a directory and its descendants using database update statements.

When a file system is to be monitored using *scandd*, a full *scan* of the file system is required for the first time. This is to populate all files in the file system into the database. Once all files in the file system have been populated into the database, the user-level process can continually update modification on the file system into database.

Using this method and assuming that the driver has not reported any errors, all files for backups including *full* can be obtained from the database using simple queries.

Special processing is required only for renamed directories. A possible implementation may be as follows:

1. Save the current time (t) and instruct *scandd* to flush its internal cache of modified file names using the *ioctl* `SCANDD_SYNC_CHANGED_FILES`.

2. Make sure all change records for the file system whose modification time is less recent than t has been populated into the database.

3. Generate a list of renamed directories from the database whose rename time (`ctime`) is greater than or equal to the reference time specified for the backup. Let us assume that is list be *l1*.

4. Generate a list of all files and directories in the file system that have been modified since the reference time minus the list of directories and its descendants in *l1*. Let us call this list *l2*.

5. Concatenate lists *l1* and *l2* to form the list of files for backup.

6. Finally, sort the file name list and this is the final list for backup.

Apart from the items in the record generated by *scandd*, the database would require some additional fields. One of the fields required would indicate whether an entry was renamed or not, valid only for directories. In addition, carefully designed indices would be important for the database table for efficient queries.

Similar to the method of generating modifies filenames for backup, an image of a file systems monitored by *scandd* (list of all files and directories existing on the file system at the time of backup) can also be generated from the database. In this case,

it is enough to list all files in the file system whose modification time is greater than or equal to zero using a database query.

5.2 Thesis Conclusion

In this thesis, I have developed and assessed a new method for enumerating files for traditional backup. This method uses a pseudo device driver that intercepts file operations in the kernel; the driver logs the full pathname of files modified, segregated by their file systems. Currently, only Solaris 10 provides the full pathname of a modified file in the kernel which is a requisite for the method developed in this thesis. The improvements seen using the method developed in this thesis can reduce the time for backup by hours on large servers. It is my hope that other operating systems will provide the Solaris-like functionalities used in this thesis in the near future.

Appendix A

Miscellaneous Information

Here I provide some information not directly related to the thesis such as the resources and tools that helped me prepare this thesis and the applications used for performance measurements in the thesis.

A.1 Compiler Information

When I started developing *scandd*, I was using the GNU C compiler (GCC) that is distributed freely with Solaris 10. However, after applying some OS updates, the GNU compiler started giving compilation errors. This resulted from the OS updates modifying some of the system header files, which rendered GCC useless for compiling the driver. Since then, I used the Sun Studio 12 compiler that is also freely available for download.

A.2 Driver Development Tools Used

The tools that I used for developing the driver were:

Compiler: Sun Studio 12.

Compiler Options Used:

- `-xO3` – for level 3 optimization
- `-D_KERNEL` – define required for kernel module compilation
- `-m64` – for 64 bit compilation as Solaris 10 is a 64 bit OS
- `-v` – requests the compiler to perform stricter semantic checking.

Other Tools:

- `add_drv` – for loading of the driver
- `rem_drv` – for unloading the driver.

Debugging Tools:

- `dbx` – for debugging user level programs
- `mdb` – Solaris modular debugger for debugging kernel crashes
- `adb` – for debugging kernel crashes.

A.3 An Important Note for Driver Development on Solaris

Many times, a driver developer may want to test a driver he/she is developing. To test the driver, it needs to be copied to the `/usr/kernel/drv/sparcv9` directory before it can be loaded. However, if there are bugs in the driver, it may render the system unstable and may cause it to crash. Moreover, when the system reboots after a crash, the same driver will be loaded again causing successive crashes. To avoid such situations, it is highly recommended that the driver binary be maintained under the `/tmp` directory and a symbolic link be created to the actual location. If the system crashes and reboots, the driver will then not be loaded again as Solaris cleans all files under the `/tmp` directory.

A.4 Documentation Tools Used

All documentation was prepared on a Linux system using the following tools:

Kile: This is one of the best tools I have come across for preparing LaTeXdocuments on Linux. It is a graphical integrated tool that understands LaTeXformatting, with one click compilation options and many more features. Kile can be downloaded from `http://kile.sourceforge.net/`.

OpenOffice: I created all flowcharts and figures in this thesis using OpenOffice 2.4 on a system running Fedora 9 Linux.

Bibliography

[1] Streamlining backup and recovery operations using disk-based protection. WWW page. http://www.dell.com/downloads/global/power/1q04-ved.pdf.

[2] Veritas netbackup 5.0 server. http://eval.veritas.com/mktginfo/products/Datasheets/Data_Protection/nbu_50_server_ds.pdf.

[3] G. Bartlett. Scsi-2 specification. WWW page, Jul 2004. http://scsi2.garybartlett.com/.

[4] A. L. Chervenak, V. Vellanki, and Z. Kurmas. Protecting File Systems: A Survey of Backup Techniques. In *Joint NASA and IEEE Mass Storage Conference*, 1998.

[5] D. Phillips. A directory index for ext2. In *Proceedings of the* 5^{th} *Annual Linux Showcase & Conference*, pages 11–20, Oakland, CA, USA, Nov 2001.

[6] Mikulas Patocka. An Architecture for High Performance File System I/O. In *INTERNATIONAL JOURNAL OF COMPUTER AND INFORMATION SCIENCE AND ENGINEERING*, pages 182–187, 2007.

[7] U. Vahalia. *UNIX Internals—The New Frontiers.* Prentice Hall, Upper Saddle River, NJ, first edition, 1996.

[8] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS file system. In *Proceedings of the USENIX 1996 Technical Conference*, pages 1–14, San Diego, CA, USA, Jan 1996.

[9] Ext2 directories. `http://www.science.unitn.it/~fiorella/guidelinux/tlk/node99.html`.

[10] S. R. Kleiman. Vnodes: An architecture for multiple file system types in sun unix. In *USENIX Summer*, pages 238–247, 1986.

[11] J. Cooperstein and J. Richter. Keeping an eye on your ntfs drives: The windows 2000 change journal explained. WWW page, Sep 1999. `http://www.microsoft.com/technet/prodtechnol/windows2000serv/maintain/featusability/msjntfs5.mspx`.

[12] A. S. Tannenbaum. *Operating Systems: Design and Implementation.* Prentice-Hall Inc., Englewood Cliffs, NJ, 1987.

[13] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel.* O'Reilly, Sebastopol, CA, second edition, 2003.

[14] J. Mauro and R. McDougall. *Solaris Internals—Core Kernel Architecture.* Sun Microsystems, Inc., Palo Alto, CA, first edition, 2001.

[15] M. Bar. *Linux File Systems.* Osborne/McGraw-Hill, Berkeley, CA, first edition, 2001.

[16] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System.* Addison-Wesley, Reading, MA, first edition, 1996.

[17] S. Quinlan and S. Dorward. Awarded best paper! – Venti: A New Approach to Archival Data Storage. In *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*, Berkeley, CA, USA, 2002. USENIX Association.

[18] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, and L. Shrira. Replication in the harp file system. In *ACM Symposium on Operating Systtem Principles*, pages 226–238, Pacific Grove, California, United States, 1991.

[19] B. Hong, D. Plantenberg, D. D. E. Long, and M. Sivan-Zimet. Duplicate Data Elimination in a SAN File System. In *Proceeding of the 21st IEEE/12th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST)*, pages 301–314, 2004.

[20] W. J. Bolosky, S. Corbin, D. Goebel, and J. R. Douceur. Single Instance Storage in Windows 2000. In *WSS'00: Proceedings of the 4th Conference on USENIX Windows Systems Symposium*, 2000.

[21] N. Garimella. Snapshot technology overview. WWW page, Apr 2006. `http://www-128.ibm.com/developerworks/tivoli/library/t-snaptsm1/`.

[22] R. H. Patterson, S. Manley, M. Federwisch, D. Hitz, S. Kleiman, and S. Owara. Snapmirror: File-System-Based Asynchronous Mirroring for Disaster Recovery.

In *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*, Berkeley, CA, USA, 2002. USENIX Association.

[23] Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, CA 95054, U.S.A. *Writing Device Drivers*, Nov 2007. `http://docs.sun.com/app/docs/doc/816-4854?l=en`.

[24] Sun Microsystems. Sunos 5.10: kernel patch. WWW page, Aug 2007. `http://sunsolve.sun.com/search/advsearch.do?collection=PATCH&type=collections&max=50&language=en&queryKey5=118833&toDocument=yes`.

[25] trapstat(1m). WWW page, May 2004. `http://docs.sun.com/app/docs/doc/816-5166/trapstat-1m?a=view`.

[26] vmstat(1m). WWW page. `http://docs.sun.com/app/docs/doc/816-5166/vmstat-1m?a=view`.

[27] Ariel Tamches and Barton P. Miller. Using dynamic kernel instrumentation for kernel and application tuning. *International Journal of High Performance Computing Applications*, 13(3):263–276, Nov 1999.

[28] Open solaris. WWW page, Jun 2009. `http://opensolaris.org/os/`.