# RadFS - Virtualizing Filesystems

by

Anoop Karollil

B.Tech Computer Science and Engineering, Cochin University of Science and Technology, 2004

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

**Master of Science**

in

THE FACULTY OF GRADUATE STUDIES
(Computer Science)

The University Of British Columbia
(Vancouver)

October 2008

# Abstract

Efficient disk space usage and quick deployment are important storage mechanism requirements in a virtualized environment. In a virtual machine farm, there is good potential for saving disk space as virtual machines are based off file system images that usually have a lot in common. Deploying virtual machines on demand should also be as quick as possible - close to or faster than the time taken in powering on and booting up a real machine.

We introduce RadFS, a shared root filesystem for virtual machines based on copy-on-write semantics. Creating a new root file system for deploying a virtual machine is instantaneous, with the ability to base the new file system on the clone of a previous file system or a snapshot of an existing file system. Disk space usage efficiency is guaranteed initially by the COW semantics and later by transparent file-system 'merges' based on content hashing. Thus any file that is identical among the file systems of the various virtual machines hosted off our file system will be shared which leads to large savings in disk space with increased provisioning of virtual machines running similar operating systems or applications that use similar sets of files. Live instantaneous snapshots of existing file systems also ensure easy backups or new bases for other virtual machines.

The filesystem or name space virtualization is implemented in user-space using FUSE and backed by a content hashing module to take care of merges.

# Table of Contents

# List of Tables

# List of Figures

# Acknowledgments

I am very thankful to my supervisors Mike Feeley and Andrew Warfield for their patient guidance and solid support. I would also like to thank Norm Hutchinson for graciously accepting to be my second reader. Thanks as well to the people I worked with at the DSG lab for their humor and genial camaraderie.

# Chapter 1

# Introduction

OS virtualization has moved from something to be tried out when there is spare time to fiddle with one's cluster/server farm to something essential that needs to be done because the benefits are too great to ignore. Virtualization adoption had been predicted to grow rapidly and with the advent of hardware virtualization support, this growth is predicted to be even quicker in the next 4 years. The growth in virtualization also sees a corresponding growth in disk storage requirements for the growing number of virtual machines per farm/cluster. These requirements should be provided not just by adding more storage, but also by exploiting the nuances of virtualization and the opportunities that it presents in ensuring efficient disk space usage and ease in provision of virtual machines.

Virtual machines (VMs) boot off filesystem images. In the Xen Virtual Machine Monitor, a privileged virtual machine (Dom0) hosts the filesystems for possibly several non-privileged virtual machines (DomU) as flat file images, physical disk partitions, LVM volumes or NFS export points and provisions virtual machines based off them. In a given server farm or cluster, it is highly likely that the images used for deploying virtual machines by a VMM have the same base files; they usually store the same operating system distribution with their associated applications to ease administration. Hence a copy-on-write approach, where there is a single read-only copy shared by all the virtual machines initially, with files modified getting copied out for each VM, is intuitively a method for efficient disk space usage. Another requirement for filesystems in general is data backup through

snapshots. A copy-on-write approach is very conducive to creating snapshots as a filesystem can be marked copy-on-write at a particular point which effectively creates a snapshot of the filesystem at that point. Added to this is also the possibility of using any of those snapshots as a base for a new filesystem deployment.

Copy-on-write filesystems are useful when there is common data (OS/application executables, packages, etc.) to be shared as it gives 'free' read sharing. Also the existence of COW in a filesystem leads to easy creation of snapshots which is a necessary requirement for most filesystems. The approaches though different at the level of implementation, basically involve copying a file (or a file block) to a different location and then linking the copied file into the filesystem transparently so that any future accesses to the file are routed to the copy. The level of implementation for this copying on modification or write can be at the file block level or at the whole file level. The former provides finer per-block granularity and hence more control over the copy-on-write semantics but with the overhead of more complex meta data handling. The latter, at the higher level of whole file copy-on-write, provides lesser control but with lower meta data handling complexity.

Virtualizing filesystems using RadFS caters to the requirements for disk space usage efficiency and quick and easy deployment of filesystems for provisioning virtual machines. By using copy-on-write semantics at the whole file level, common files in a root (bootable) filesystem are shared among all virtual machines. This provides obvious savings in the initial deployment phase of virtual machines where they start off with virtual filesystems based on the same disk image. But given a deployment environment, like a cluster or an office, there is also a high probability that a sizeable number of newly created files, for example software packages, updated system files or even PDF documents, in each virtual machine are identical to each other even after individual copy-on-write or 'create' operations. To provide continuing disk space usage efficiency, a content hashing module keeps track of identical files among the different filesystems and maintains a single copy that is shared even after a copy-on-write or create operation. By virtualizing the filesystem name space and with the COW mechanism, very fast snapshots and new filesystem deployments are also made possible. New filesystems can be based on the initial base filesystem, on one of the branches spawned off the base filesystem or even on a snapshot.

The goals of disk space saving and instantaneous snapshots/virtual filesystem deployment is achieved by virtualizing the namespace using FUSE (Filesystem in User Space [1]) and then exporting the virtualized filesystems for Xen DomUs using NFS. FUSE filesystems, though implemented in userspace, perform as good as 'in-kernel' filesystems like ext3, with proper optimizations [14]. FUSE provides the means to multiplex requests from different DomUs to a single shared root filesystem with the guarantee that the base root filesystem remains unchanged and updates/modifications are redirected to each virtual filesystem's writeable 'branch' on disk. This copy-on-write operation is the basic underlying mechanism to provide namespace virtualization. The namespace is virtual and comes into true existence, backed by writable files, only when written to. Virtualizing a filesystem namespace involves not only the virtualization of disk locations and filesystem attributes for files and directories but also the mechanisms used to access these attributes. FUSE provides the interface for facilitating this virtualization.

## 1.1 A quick introduction to FUSE

RadFS virtual filesystem is based on FUSE. FUSE consists of a kernel module and library which is used by a userspace application handling a virtual filesystem, to call and get called by the FUSE kernel module. FUSE lets a user mount the virtual filesystem after which any access to the mount point is routed by the FUSE kernel module to the userspace application through the FUSE library. The userspace application can then implement its own file operation methods or reroute the requests to an existing filesystem. The latter approach avoids implementing all the 'heavy lifting' mechanisms needed in a traditional filesystem and is what is used in building RadFS. RadFS is the userspace application that provides the COW, snapshot and content hashing mechanisms for virtual filesystems which are backed by an EXT3 filesystem or any other filesystem supported by the Linux kernel Virtual Filesystem (VFS).

Figure 1.1 shows the flow of control for a typical file operation. In the case of RadFS, an *open* filesystem call to a file on a virtual filesystem gets routed to the RadFS application, which might perform a copy-on-write operation or initiate content hashing, and then re-issues the open to the backing (EXT3) filesystem.

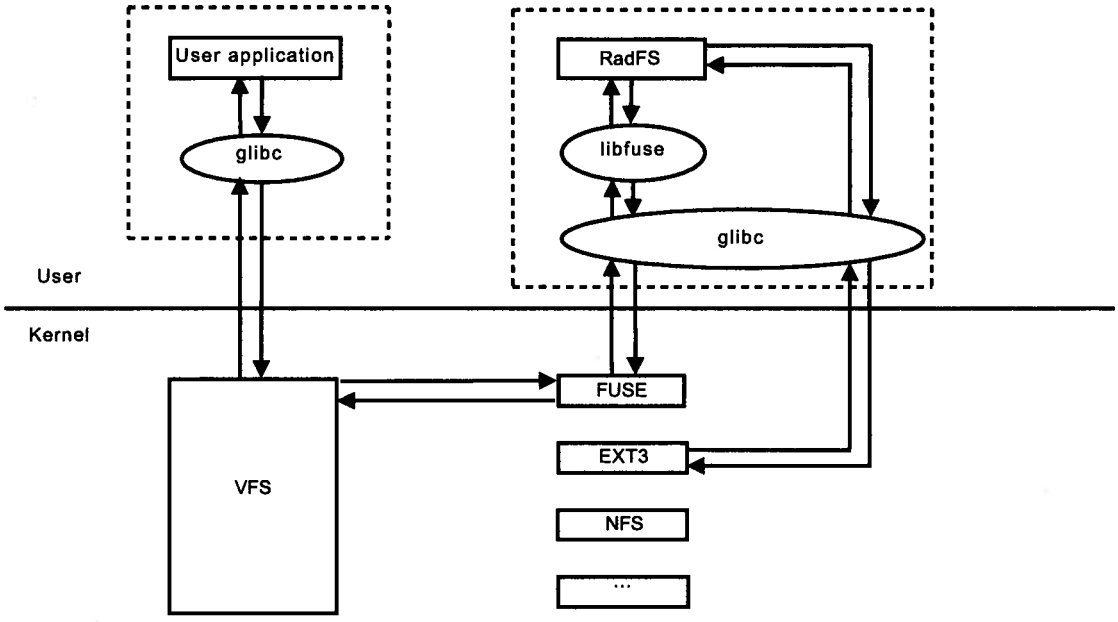


**Figure 1.1:** FUSE working

The file descriptor thus obtained is returned back to the user application that did the initial open request.

# Chapter 2

# Related Work

RadFS provides quickly deployable virtualized filesystems that guarantee savings in disk space and instantaneous snapshots, using a COW mechanism and content hashing. It has been built specifically with virtual machines in mind. A number of COW filesystems are prevalent in the field, having snapshot and disk space saving features, and a few of them specifically for virtual machines. This chapter tries to cover related virtual machine filesystem work, having COW or content hashing mechanisms. The main difference between RadFS and most other related filesystems is the level in the filesystem at which the COW/content hashing takes place. RadFS works at the whole file level while most other filesystems work at the file block level. Another common difference is the backing store used by the virtual machine filesystems which vary from real filesystems to filesystem image files.

## 2.1 qcow

qcow is a filesystem image format used by by the QEMU [5] processor emulator as well as the Xen VMM as a virtual machine filesystem. Root filesystem images are whole root filesystems stored in a single file that can be used by virtual machines. The filesystem image is a representation of a fixed size block device which is exported to the virtual machine by the emulator or VMM using a loop device (blktap [16] in Xen). Files modified by the virtual machine are modified in place in the file image. The qcow filesystem image format [4] has the advantage that it

supports sparseness even when the underlying filesystems doesn't support sparseness and it also has COW support. These coupled together can be used to quickly deploy filesystems based on other qcow images. The new image will use very little disk space as it has 'read-only' pointers to the original image with writes to files leading to modifications in the new image file. A problem with this approach is that the backing store is an image file which isn't as efficient as a regular filesystem backed store, especially in disk space reclamation on file deletes which needs special handling.

## 2.2 Unionfs

Unionfs [7] had a lot of influence on the design of the COW mechanism in RadFS and so there are several similarities among the two in the way they create virtual filesystems. Unionfs merges different filesystems called branches and creates a virtual filesystem formed of the union of the two or more filesystems. Directories with same paths in the individual filesystems are represented as a single directory in the virtual filesystem with the contents from the individual filesystem directories merged. Each base filesystem is identified by a branch id that is used to specify priorities in choosing files from a particular branch when there are files with identical paths from more than one branch. COW is achieved in Unionfs by using two branches and marking one of the branches, possibly the one with data as read-only and the other possibly empty branch as read-write. Initially all read requests are satisfied by the read-only branch. Writes/modifications to files lead to a copying over of the modified file to the read-write branch and further requests for the file are serviced from the read-write branch, by giving the read-write branch id a higher priority.

Since the virtual filesystem is a union of two or more real filesystems, to maintain consistent filesystem semantics of file operations in the virtual filesystem, Unionfs needs to deal with the problem of handling the various branches of the union as a single entity. Errors in unlinking files in a particular branch are propagated to the user only if the unlinking fails in the highest priority branch. That is, if there is an unlink error in one of the lower priority read-write branches, then the operation still succeeds if the unlink is successful in the highest priority branch.

Relating this to COW, deleting a file in the union should lead to the file in a read-write branch being deleted if present while it shouldn't be unlinked from a lower priority read-only branch. So UnionFS needs to mask a lower priority read-only file that has been deleted from a higher priority branch and it does so by creating high priority files called white-outs, each of which signifies the absence of a particular file from a particular high priority read-write branch. So in the case where a read-only file is deleted from a union of read-only and read-write branches, a white-out is created for the particular file in the read-write branch (after deletion of the file from the read-write branch, if present). Further file operations like stat, readdir, open on the union check for the presence of a white-out for the file being accessed and return a no-existence error if a white-out is found. So even if the file is present in lower priority branches, the white-out masks its presence from the union.

One problem with this approach is that white-outs pollute the filesystem namespace. Each deleted file that has a read-only counterpart in another branch needs to have a white-out in the read-write branch. Another problem is that of creation of white-outs recursively. To handle the case of a read-only directory being deleted and then recreated, white-outs have to be created for all the files and subdirectories in the directory when its deleted to prevent them from showing up in the newly created empty directory. The basic problem is the absence of a dedicated metadata structure. This has been remedied in a version of Unionfs that has a metadata structure called On Disk Format (ODF) [6]. The ODF is a regular filesystem on its own within the kernel that keeps track of white-outs and also does caching.

## 2.3 Parallax

Parallax [11] is a distributed storage system specifically built for Xen virtual machines. Parallax works at the block level on a shared global block store and provides a block device interface called a Virtual Disk Image (VDI) to virtual machines. The VDI has certain similarities to the qcow image; they both split a linear block address into levels that are used to then look up the block, they both support sparseness wherein a block is allocated only when written to, and they both use their support for sparseness to provide fast snapshots and COW features. Parallax

has a distributed architecture. Multiple physical hosts (physical machines with a Virtual Machine Monitor and multiple virtual machines), each running their own Parallax server in a storage virtual machine, access the common block store. The block store is divided into extents that each Parallax server locks to provide storage for its virtual machines. Each virtual machine also locks a VDI and associated data blocks for exclusive in-place writes. RadFS isn't distributed as there is only one instance of the RadFS metadata server that runs on a physical host. Though this might decrease the availability of virtual filesystems served by RadFS, it increases opportunities for easy sharing and more savings in disk space.

Parallax uses a radix tree look-up mechanism in addressing blocks, which RadFS borrows and which is the crux of the COW mechanism. While Parallax splits the bits in a block's linear address to do its look-up of a block, RadFS uses path segments within a disk path to look-up a particular file or directory. Both have read-only pointers; they are to blocks in the case of Parallax and to whole files in the case of RadFS. Comparing Parallax with RadFS should be enlightening on the benefits and drawbacks of implementation at the block and whole-file levels respectively, once sufficient optimizations to RadFS have been done.

## 2.4 Ventana

Ventana [12] is an object store based distributed virtualization-aware filesystem which aims to combine file-based storage and the sharing benefits of a distributed filesystem with the versioning, mobility and access control features of virtual machine disk images. The motivation behind Ventana is to provide read/write sharing between multiple users, to track files in various virtual disks that are related, in a cohesive manner, and to provide finer than whole disk rollbacks in virtual disks. Like RadFS, Ventana permits spawning virtual filesystems based on existing virtual filesystems recursively, in a hierarchical manner. To meet these requirements, Ventana provides abstractions called branches and views. A branch is a particular version of a file tree, and can be private or shared. The concept of a branch in Ventana and RadFS are more or less the same except for the scope. For example, a branch in Ventana could be a version of the /usr subdirectory or of a whole root filesystem whereas a branch in RadFS is always a version of a root filesystem.

Ventana splits up a root filesystem into file trees to aid in sharing and to deal with security/maintenance problems when using virtual disks [9]. In Ventana, a particular directory can be shared read-write while the rest of the filesystem is read-only by having a filesystem 'view', with the directory tree as a 'shared' branch while the root filesystem is a private branch. This is akin to having a network share mount point that is read-write within the root filesystem.

The problem of applying patches or security updates to all virtual machine disks, even those not currently being used is solved by having access to the Ventana common store which has the normal abstractions of files and directories, or branches used by each virtual machine, that can be read by normal malware scanning and backup tools. This is the case with RadFS too; the backing store is made of ext3 filesystem directories that can be read and written independent of RadFS while keeping the virtual filesystems consistent.

Ventana also has access control at various levels of abstraction:- there are normal file, file version and branch ACLs. These are mainly to prevent security leaks arising from the forking of virtual filesystems from existing virtual filesystems. Security updates and file access controls should transcend file versions in the various virtual filesystems. Security updates can be easily applied to a file-directory oriented common store but file access controls transcending file versions require ACLs that apply to the current as well as prior versions of a file. Version access control is future work in RadFS but given the design of the RadFS metadata tree, it should be trivial. Metadata nodes for versions of a particular file are linked together in the RadFS metadata tree.

Ventana also provides the user with the capability of viewing all older versions of a particular file. In the case of virtual disks, this is tedious as each version of the virtual disk is a separate unit that needs to be mounted. Given the file-directory oriented back store and using the linked list of metadata nodes for multiple versions of a file or directory, it would be again quite trivial for RadFS to have a utility that scours the store branches to find all versions of a particular file.

## 2.5 Content hash related work

RadFS uses a content hashing mechanism to keep disk space savings consistent even if individual virtual machine filesystems diverge from a common base. This section compares RadFS with other content hashing systems.

### 2.5.1 Microsoft Single Instance Store (SIS)

Microsoft's Single Instance Store [3] is a content hashing system for Windows Storage Servers that provides savings in disk space by consolidating duplicate files into a common store. It consists of a user level service that generates content hashes of files and compares them with a database of content hashes it maintains. The hashing service then reports duplicate files with identical content hashes it finds to a kernel level filesystem driver that copies the file to the common store if it not already present or replaces it with a link to an identical file in the common store. The kernel driver redirects reads to the common store and writes are handled by a copy-on-close mechanism as opposed to COW. The copy-on-close (COC) approach is more efficient that COW as only those portions of the file that haven't been overwritten are copied over from the common store. So all writes to a file happen without the initial copying over as in COW and then on close, only the portions of the file that haven't been written to are copied over. RadFS currently doesn't have a COC mechanism but its one of the future optimizations planned. SIS uses a 128 bit signature to check for duplicate files. The first 64 bits denote the size of the file while the remaining 64 bits contain the hash of 2 4KB blocks from the middle of the file. If these match, then SIS does a full binary comparison. RadFS checks the identity of files using SHA1 hashes and assumes that the hashing is collision free. Quinlan & Dorward [13] also uses SHA-1 to generate hashes for 8KB blocks and their analysis show that the probability of hash collision when there are approximately $10^{14}$ blocks is $10^{-20}$. Since RadFS uses SHA-1 hash digests at the file level, it can be assumed that the SHA-1 hashes are collision free.

### 2.5.2 Venti

Venti [13] is a content addressable storage system at the block level. Venti provides a interface to store and retrieve blocks from a common store based on a SHA-1 hash of block content, called a fingerprint. It doesn't provide the services of a filesystem but provides the infrastructure for an archival filesystem that can be built around it by applications. Rather than blocks being addressed by LBA, fingerprints of blocks, including recursive 'fingerprinting' of blocks that have fingerprints of other blocks themselves, help to build an addressing system that lets an application address blocks as on a regular disk. This provides the benefits of duplicate block coalescing, built in block integrity checks, and immutability of blocks and their addresses which are quite useful in archival systems where data needs to be retained for a long time without major changes. Coalescing of duplicate blocks is dependent on block sizes as well as alignments of the blocks in a file and makes disk space savings complicated if not unpredictable. Here too, a comparison of content hashing at the block level and file level involves a trade off between complexity and better control over the management of copy-on-write or coalescing semantics. A byte modified in a large file leads to only a few new blocks getting allocated in the case of content hashes at the block level. The whole file is copied over (this overhead can be limited using copy-on-close) in the case of content hashing at the file level, but with the advantage of having an existing high performance block addressable filesystem take care of the complexity in addressing, caching, scheduling, etc.

# Chapter 3

# Design

The goals of this thesis are to provide a filesystem for virtual machines that

- saves disk space,
- is quickly deployable,
- allows snapshots and recursive deployments.

In a virtualized environment with a server hosting multiple virtual machines, it is common practice to spawn multiple virtual machines using copies of the same virtual disk. Thus disk space savings can be achieved by exploiting the fact that virtual machines on a server/cluster have a high probability of sharing the same operating system distribution with common system and application packages. This commonality of files used continues even as updated packages or new applications are installed and the opportunities for exploiting this commonality grows as the number of virtual machines deployed increases. Thus sharing is the key to saving disk space and a filesystem should provide the ability to share files without major side effects on performance or resource usage.

Virtual machine deployment time should mirror real hardware; it should be possible to start a virtual machine within the same time frame that it takes to boot up a real machine, maybe faster. Ideally this should be the case even for dynamic virtual machine deployments where a sudden requirement necessitates starting up multiple virtual machines in as short a time as possible. And ideally, this should be

possible without pre-allocated virtual machine disks. The main deterrent to quick deployment of virtual machines is the time taken to provision a filesystem or virtual disk that the virtual machine can boot from. Thus a filesystem for virtual machines should also be very quickly provisioned.

It is usual practice when using virtualization to build a virtual disk or virtual filesystem based on a particular OS and distribution with a certain set of applications pre-installed for use by various users. These are then copied whenever needed to create new virtual disks with maybe other modifications to them. Hence the ability to base certain virtual disks/filesystems based on previous disks or filesystems is quite useful in virtualization and this ability goes hand in hand with snapshots as a snapshot makes a virtual disk immutable. Thus a snapshot sets the point in a virtual filesystem/disk which is interesting (for example a new kernel or a service pack install) for use as a base for other virtual disks/filesystems. Thus snapshots are the means to recursive deployment of virtual disks and also provide a much needed backup mechanism for filesystems in general.

To meet these needs, RadFS is designed to use a shared root filesystem with virtual machines booting off virtual filesystems, based on a single root filesystem and using copy-on-write techniques for write sharing. copy-on-write makes it easy to provide the ability to create snapshots. By marking a particular virtual filesystem read-only, any further modification requests are redirected to a new ondisk location, leaving the original virtual filesystem untouched. This also satisfies the requirement of quick deployment, either from the base filesystem image, which is already read-only, or from existing active virtual filesystems, by creating snapshots of them. And since all virtual machines start with virtual filesystems based on a single base filesystem, there are obvious space savings. Thus a COW mechanism is definitely the crux of the design that lets RadFS achieve the goals mentioned above.

## 3.1 Architecture

RadFS is based on FUSE. FUSE provides the interface to build a virtual filesystem based on an existing filesystem, which is ext3 in the case of RadFS. RadFS consists of a FUSE application, a radix tree oriented metadata server and a content hashing

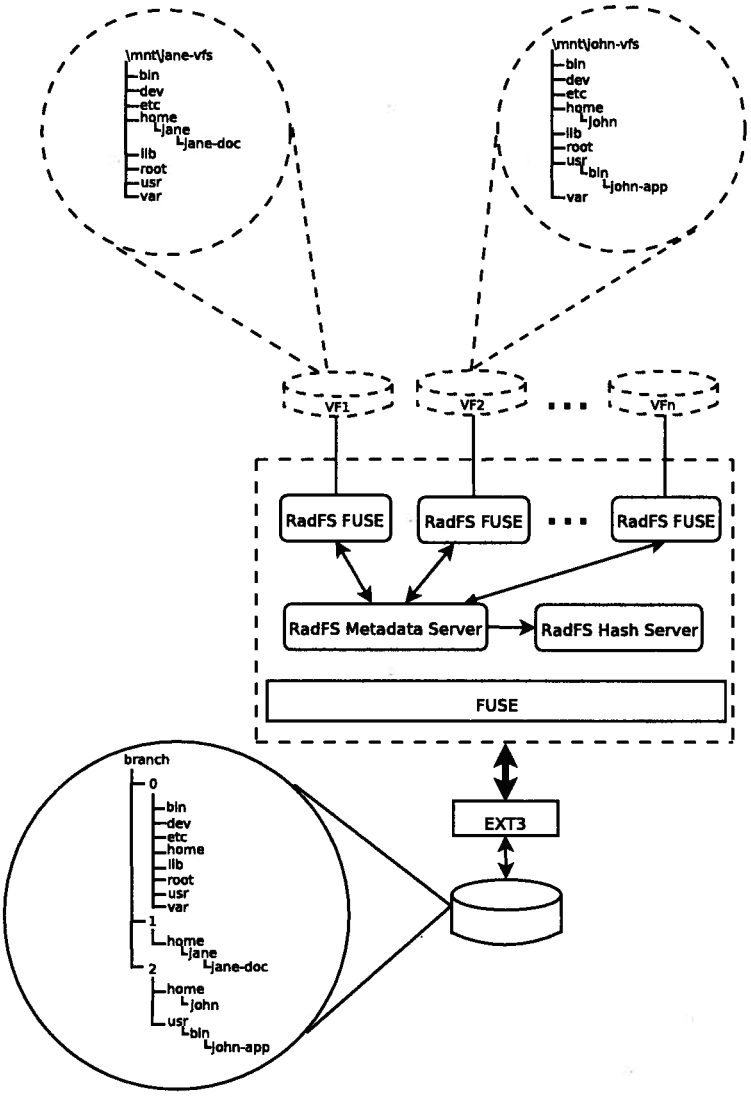


**Figure 3.1:** RadFS architecture

server. Figure 3.1 describes the high level architecture of RadFS.

The RadFS FUSE application uses the FUSE library to intercept filesystem calls to the virtual filesystem it is associated with (see Section 1.1). The virtual filesystem, which is based on a read-only branch (a directory populated with a root

filesystem which is tagged read-only, /branch/0 in Figure 3.1), represents a virtual disk which a Xen virtual machine (VM) can boot off.

Each virtual disk is associated with a RadFS FUSE application that handles requests to it and relays the request either to the read-only branch or to a read-write branch (a directory populated with files copied over or newly created, /branch/1 and /branch/2 in Figure 3.1), with a copy-on-write operation if necessary. RadFS-FUSE processes filesystem requests after getting information about the current file/directory from the RadFS metadata server. The information includes the read-only/read-write status, the disk path of the file or directory, the file attributes for regular files and other information.

The metadata server responds to requests for look-ups, insertions, deletions and other operations that are necessary to keep track of changes to the different virtual filesystems. Each virtual filesystem that is initialized using the RadFS FUSE application gets a branch id that uniquely identifies it to the metadata server. This branch id changes only when a virtual filesystem is snapshotted when a new branch id is assigned to it, with the old branch id denoting the now read-only snapshot.

The metadata server also forwards requests for content hashing from the RadFS FUSE application to the RadFS Hash Server. The hash server's only purpose is to generate content hashes of files and update the read-write branch and the common store. Thus it is more or less independent of the RadFS FUSE application and the metadata server, and acts as a sink for asynchronous content hashing requests.

The RadFS FUSE application maintains the actual files in the backing EXT3 filesystem. Each virtual filesystem is made up of a base read-only root filesystem EXT3 directory and a read-write EXT3 directory that starts off empty but stores copies of files on modification or creation. The RadFS metadata server stores the metadata necessary to merge the backing EXT3 filesystem directories to create the virtual filesystem. This includes a representation for virtual filesystem directories that combines the contents of the directories from the read-only and read-write EXT3 directories, keeping track of files that are deleted by using metadata 'white-out' nodes, and other information needed for sharing data among the various virtual filesystems using content hashing and copy-on-write.

The design for the RadFS FUSE application is more or less laid out by the FUSE library call-back interface. FUSE gives the base for virtualization and virtu-

alizing a filesystem mostly involves handling filesystem metadata and methods to manipulate the metadata to provide features like copy-on-write and content hashing. Thus the design of the metadata server revolves around the core metadata structure that needs to handle multiple filesystems with COW semantics. Persistence of this metadata is also a crucial part of the metadata server as in the case of virtual filesystems, its the metadata that makes the filesystem. The RadFS hash server is more or less a separate entity whose content hashing mechanism design isn't influenced by either the RadFS FUSE or RadFS metadata server applications as the interface to it can be narrow and one way (see Figure 3.1) without affecting its functionality. The content hashing provides another level of sharing but it is again built on the copy-on-write mechanism and so is an easy extension to the design. The remainder of this chapter expands on the design of the metadata structure used to support COW for multiple virtual filesystems, persistence, the content hash mechanism and a CLI user interface to RadFS.

## 3.2 Radix tree metadata structure

A radix tree metadata structure is the core of RadFS and it is maintained by the RadFS metadata server. The metadata structure maintains information about the various virtual filesystem branches that are being served and the server handles requests for path look-ups, insertions, deletions and other operations needed to maintain the state of each virtual filesystem namespace. For example, the most basic operation performed by the RadFS-FUSE application is routing file requests to the read-only or read-write ondisk branches. To do this, it queries the radix tree metadata server, giving it a virtual filesystem path and a branch id to get the ondisk path of the file/directory needed. Similarly, most queries to the metadata server are made up of a virtual path and a branch id.

Each node in the tree represents a file or directory with the leaf nodes in the tree representing files and the other nodes representing directories. When the server is started, it builds the base read-only branch (the 'gold master' branch or branch 0) from a directory containing a root filesystem. This 'gold' branch forms the initial template for other branches. Any existing branch can be used as a template for new branches if it is read-only, which it can be made to be by creating a snapshot of it.

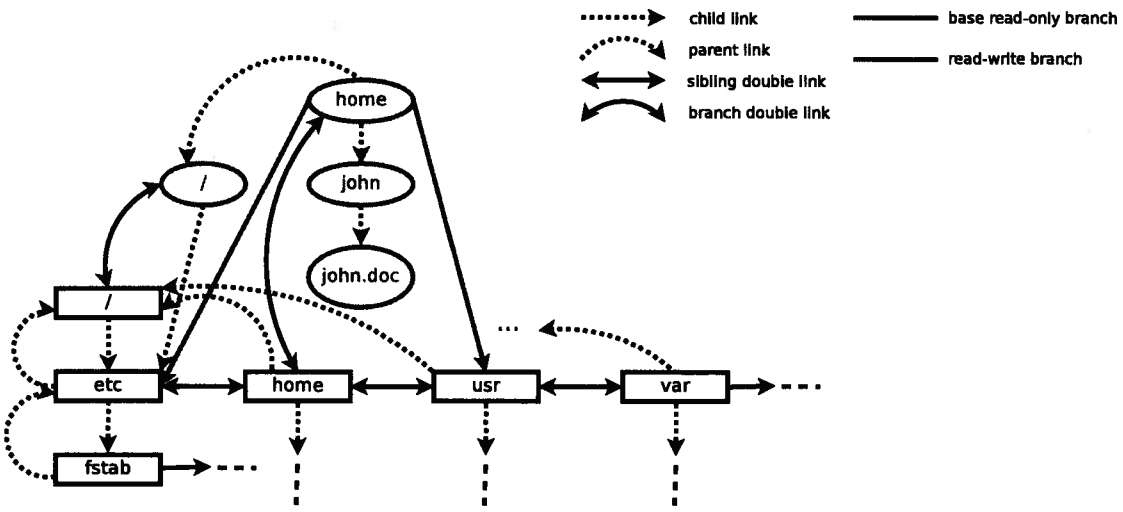


**Figure 3.2:** Radix tree metadata structure

This provides for recursive deployment of virtual filesystems based on previous virtual filesystems.

Each node in the tree has parent, child, sibling and branch links to other nodes. The parent link of a node links it to the node representing the directory that it is contained in. The only node that doesn't have a parent link is the root node. The child link links a directory node to one sub-directory or file contained in the directory. The other sub-directories and files are accessed through the sibling links. The sibling links are used to produce a doubly linked list of files and sub-directories of a directory, the head of the list being pointed to by the child link of the parent directory node. The branch links link nodes in the different branches that have the same virtual path. In Figure 3.1, the nodes representing /home in branches 0, 1 and 2 would be linked together by branch links to form a doubly linked list. Figure 3.2 shows a part of a metadata radix tree with a base read-only branch, a single branch based on the read-only branch and shows parent, child, sibling and branch linkages.

Each node in the tree has an associated branch ID which identifies it to be part of the read-write branch of a virtual filesystem with that branch ID. To all other virtual filesystems, a branch ID different from their own branch ID signifies that the node representing the file or directory is read-only. Thus the branch ID provides

the basis for the copy-on-write mechanism involving multiple virtual filesystems.

## 3.3 Persistence

The virtual filesystems spawned should remain consistent in the event of hardware failure or a reboot/crash of the host machine (Dom0) running the metadata server. Since RadFS virtual filesystems are implemented at the file level and backed by ext3, consistency at the block level is guaranteed by the backing filesystem but the virtual namespaces of each virtual filesystem maintained by the RadFS metadata server needs to be maintained consistent.

Each virtual filesystem is based on a common ondisk read-only branch and its own ondisk read-write branch which stores modified copied on write and newly created files and the associated directory structure. Hence it is possible to build the virtual filesystem back up from a union of the branches on disk with a special handling of delete and rename operations as they don't reflect on the read-only branch. But to keep the metadata recovery independent of ondisk data, and to also meet the requirement of keeping track of additional metadata like file attributes, hard link back pointers etc., RadFS logs all metadata operations that modify the metadata radix tree. It also writes out the whole metadata tree to disk when the operations log reaches a specific size after which it is reset. The ondisk metadata tree along with the log of metadata operations performed after the tree was written to disk is enough to rebuild the metadata tree structure to a consistent state.

## 3.4 Content hashing

In RadFS, disk space usage efficiency is achieved in part by the shared root filesystem approach wherein virtual machines share a common base root filesystem using a copy-on-write mechanism. But with time and use, the virtual filesystems associated with each virtual machine will show deviations from the base image, when they download new packages and install new applications or modify existing files. To counter this, and to keep disk space savings persistent, RadFS implements a content hashing module that ensures continued sharing of data.

The design involves a content hash dump which is a data store at the file level with each file in the dump being named with the 20 byte SHA-1 hash of its content.

Each regular file in RadFS has a content hash generated and is linked to the content hash dump. Thus if any two files in any of the branches have the same content, be that a Linux kernel image package, a common PDF document received through email, an MP3 file or a popular video clip, they are all linked to a single file in the content hash dump. Each new file created in a virtual filesystem, or modification of an existing file in a virtual filesystem leads to a content hash generation for the file and a linking to the content hash dump. The content hash generation is handled lazily by a the RadFS hash server whose only purpose is to process requests for generating content hashes for files created or modified by the various virtual filesystems and manipulating the links between the file in read-write branch and the content hash dump. The RadFS FUSE application issues a content hashing request to the RadFS metadata server which processes it and re-issues the request to the RadFS content hash server. The request for a content hash is one way; the RadFS FUSE application pushes the request out and continues processing other filesystem calls. Thus the RadFS hash server acts as a sink for all content hashing requests. See Section 4.3 for more details.

## 3.5 Command Line Interface

RadFS has a command line interface (CLI) that lets a user interact with the RadFS metadata server and perform the following:

- **Start the server** - starts the server based on a base root filesystem directory and a directory to use for read-write branches. The latter is also the location where the content hash dump would be built, if required.
- **Initialise content hash dump** - Creates a content hash dump which initially contains the content hashes of the files in the read-only base root filesystem
- **Create a new virtual filesystem** - Creates a new branch given a base root filesystem (which can be the initial read-only branch or any other subsequent snapshotted read-only branch)
- **Mount a created virtual filesystem** - Mounts a virtual filesystem given a mount point and the filesystem branch id. This starts the RadFS FUSE application that handles filesystem requests to the specified mount point.

- **Unmount a virtual filesystem** - Unmounts a virtual filesystem given the filesystem branch id
- **List virtual filesystems** - List filesystems being managed by the metadata server. The following is an example listing displayed by the RadFS CLI:

```
Branches served
(branch-id : tag : mountpoint)
0 : base-read-only : read-only-never-mounted
        1 : ubuntu-desktop : read-only-never-mounted
                3 : john-ubuntu 17Jul08-10:45:05 : read-only-never-mounted
                        5 : kernel-upgrade-2.6.25 02Aug08-15:57:44 : /mnt/john
                4 : jane-ubuntu : /mnt/jane
        2 : ubuntu-server : /mnt/lamp
```

- **Snapshot a virtual filesystem** - Saves the state of a virtual filesystem at a particular point. This snapshot is a virtual filesystem that is read-only and it serves as a backup as well as a base for other virtual filesystems.
- **Shutdown the server** - Shuts down the RadFS metadata server and content hash server.

### 3.5.1 Use Case

The branch listing above could be the result of the following operations performed using the CLI:

1. Start the server using a root filesystem directory (populated using, for example, an Ubuntu 7.04 image). This creates the base-read-only branch with branch id 0.
2. Create a virtual filesystem ubuntu-desktop based on branch 0, mount it, boot off it and install applications like KDE/GNOME, Firefox, Thunderbird, etc.
3. Create a virtual filesystem ubuntu-server based on branch 0, mount it, boot off it and install Apache and MySQL.

4. Snapshot ubuntu-desktop to create a writeable virtual filesystem for user John based on ubuntu-desktop and making the ubuntu-desktop branch with its installed applications read-only.

5. Create a virtual filesystem jane-ubuntu for user Jane based on the now read-only ubuntu-desktop branch.

6. Mount both john-ubuntu and jane-ubuntu.

7. Snapshot john-ubuntu and then John upgrades his kernel to 2.6.25. After the snapshot operation, branch 3 would be a virtual filesystem which has the changes made by John to the point where he upgrades his kernel. Branch 5 would be the branch John uses to download the kernel and which reflects other changes made after the snapshot.

# Chapter 4

# Implementation

RadFS is implemented in C and uses the Filesystem in Userspace (FUSE) library to create its virtual filesystems. It was developed under Ubuntu 8.04 GNU/Linux, with the Xen 3.2 VMM and a Linux 2.6.24 kernel. The privileged virtual machine, Domain0 run by Xen hosts RadFS and creates the virtual filesystems that are exported via NFS. Virtual machines spawned by Dom0 use NFS boot to boot off the virtual filesystems exported as NFS mount points, with RadFS taking care of data sharing using a copy-on-write mechanism. The RadFS FUSE application communicates with the RadFS metadata server using a custom protocol using Unix sockets. The metadata server talks to the RadFS content hash server also using Unix sockets. RadFS uses the GNU gcrypt library [2] to generate SHA-1 content hashes. The backend for the virtual filesystems, storing their copies of modified files is an ext3 filesystem. But since FUSE deals with the Linux VFS layer, the backend can be any filesystem that is supported by VFS.

## 4.1 RadFS FUSE application

The RadFS FUSE application is the virtual filesystem builder that is invoked each time a new virtual filesystem is created. It uses the FUSE library to mount a virtual filesystem and then handles any filesystem call directed to the associated mount point. The FUSE library lets the application create a list of call-backs for the standard set of filesystem calls. Any filesystem call directed to a virtual filesystem

mount point leads to the corresponding call-back being executed in the RadFS FUSE application. The RadFS FUSE application then uses the RadFS metadata server to ensure copy-on-write semantics for filesystem operations. It also issues content hash requests and maintains the links from the virtual filesystem to the content hash dump.

### 4.1.1 Virtual filesystem branches

Each virtual filesystem is based off a read-only base virtual filesystem which is represented in the metadata radix tree by a branch with a specific branch ID. Initially the only read-only filesystem is the branch with branch ID 0, but later on can be other read-only branches created through snapshots of existing virtual filesystems. Each virtual filesystem also has a read-write branch that stores any files that it creates or modifies from the read-only branch. This read-write branch ID is a handle that is passed to the RadFS metadata server along with every metadata request issued by a particular virtual filesystem. When a virtual filesystem is created/mounted, the RadFS FUSE application in charge sends a request to the metadata server to initialize the read-write branch. This request returns a branch ID that is then used by the RadFS FUSE application as a handle for all further requests to the metadata server. The branch ID given to the FUSE application at the time of virtual filesystem initialization when associated with a node in the virtual filesystem indicates that the node is writeable. When a virtual filesystem is initially mounted, only the node representing the root directory '/' for that branch would have the read-write branch ID. This root directory node would be linked to the nodes that correspond to the subdirectories of the root directory but these initially would be on the read-only virtual filesystem branch that the new filesystem was based on.

### 4.1.2 Virtual paths

A virtual filesystem mounted by RadFS will be a root filesystem rooted at '/' and populated with the standard system directories etc, home, usr, var etc. The virtual filesystem mount point thus could be thought of as representing a disk with a bootable filesystem on it. A particular path in the virtual filesystem is translated to an actual ondisk path by RadFS using the metadata tree. A virtual filesystem con-

sists of virtual paths that are identified as read-only or read-write by the branch ID of the node representing the path's target file or directory. At the time of initialization, each virtual filesystem will have just the root directory marked as writeable and backed by a directory ondisk. This directory is then used to store the files and directories that are created or copied over in a COW operation. Initially the root directory listing of a virtual filesystem would show sub-directories of '/' but they would be backed on disk by a directory associated with the read-only virtual filesystem which was used to build the virtual filesystem. Whenever a file is modified, the file gets copied over from the disk path associated with the read-only branch to the disk path associated with the read-write branch. This copying over also includes creating the directories leading to the file being copied over. File creations are also routed to the read-write disk path. Thus each virtual path in a virtual filesystem is either read-only, with read requests being satisfied from a disk path associated with the read-only branch, or read-write with the initial write and subsequent reads being satisfied from the disk path associated with the virtual filesystem's read-write branch. There can be more than one read-only ondisk branch associated with a particular virtual filesystem. This is because virtual filesystems can be based on other virtual filesystems that are read-only and this can happen recursively.

Figure 4.1 shows two virtual filesystems jane-vfs and john-vfs with their associated read-write and read-only ondisk backing. Both the virtual filesystems are based on the read-only branch with branch id 0. The virtual filesystems start with the ondisk directories branch/1 and branch/2 which act as the store for COW operations. The user of virtual filesystem *jane-vfs*, creates a home directory *jane* and a file *jane-doc*. This leads to the directory and file being created in the ondisk directory branch/1. Similarly, in the virtual filesystem *john-vfs*, the directories *john, bin*, and the file *john-app* are created in the ondisk directory branch/2. The creation of file *john-app* leads to the creation of the directories *usr* and *usr/bin* in branch/2. A request for an unmodified file in branch 0 like */usr/bin/bash* gets routed to branch/0 (*branch/0/usr/bin/bash*) while a request for a modified or newly created file goes to branch/1 (e.g., *branch/1/home/jane/jane-doc*) in the case of *jane-vfs* virtual filesystem and branch/2 (e.g., *branch/2/usr/bin/john-app*) in the case of *john-vfs*. This routing is done by RadFS FUSE by using the branch ID associated with each path

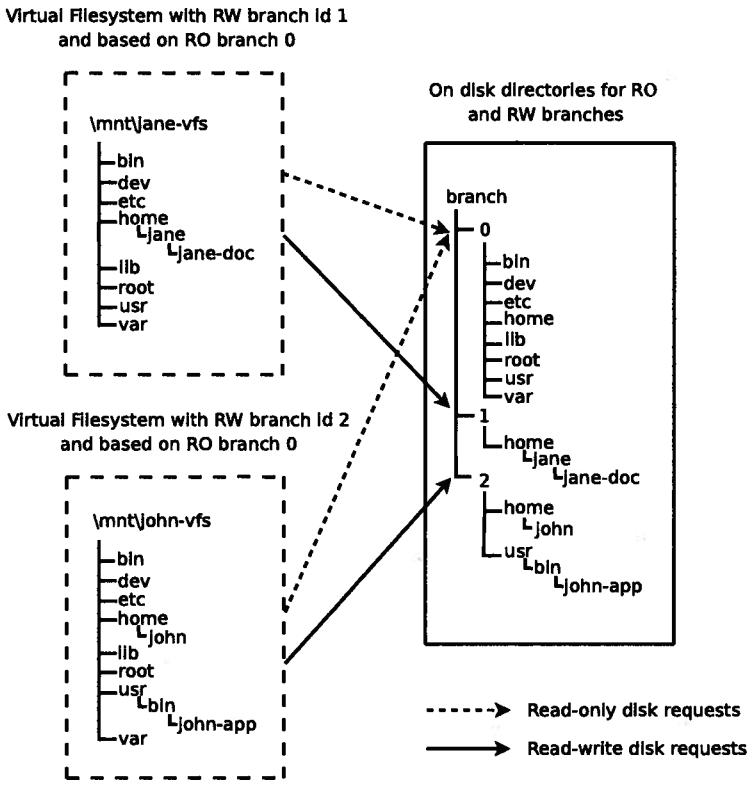


**Figure 4.1:** Virtual paths mapping to ondisk paths

in a filesystem call-back, which is obtained from the metadata server using RadFS metadata operations.

### 4.1.3 RadFS FUSE call-backs and metadata operations

Table 4.1 lists the FUSE callbacks used by RadFS and the corresponding RadFS metadata operations performed that update the metadata radix tree. Each RadFS operation results in exchange of data between the RadFS FUSE application and metadata server using UNIX sockets. This section gives an overview of how the RadFS FUSE application uses the metadata operations to create virtual filesystems with COW and content hashing. The metadata operations are described at the level of abstraction required at the RadFS FUSE application side. More details about

each RadFS metadata operation, as implemented in the RadFS metadata server are described in Section 4.2.2. The operations in *italics* are those that are conditionally executed in a call-back based on the nature of the file/directory.

**Table 4.1:** FUSE callbacks and associated RadFS operations

| **FUSE call backs** | **RadFS operations** |
|---|---|
| getattr (stat) | GET-STAT |
| fgetattr (fstat) | fstat |
| access | GET-BRANCH-ID, GET-DISK-PATH, access |
| readlink | GET-BRANCH-ID, GET-DISK-PATH, readlink |
| readdir | GET-CHILDREN, lstat, filler |
| mknod | GET-BRANCH-ID, GET-WRITEABLE-DISK-PATH, mknod/mkfifo, *lstat*, *SET-STAT* |
| mkdir | GET-BRANCH-ID, GET-WRITEABLE-DISK-PATH, mkdir |
| unlink | GET-BRANCH-ID, GET-STAT, GET-DISK-PATH, *unlink*, DEL |
| rmdir | GET-BRANCH-ID, GET-CHILDREN, GET-DISK-PATH, *rmdir*, *lstat*, DEL |
| symlink | GET-BRANCH-ID, GET-WRITEABLE-DISK-PATH, symlink |
| rename | GET-BRANCH-ID, GET-WRITEABLE-DISK-PATH, GET-DISK-PATH, GET-STAT, SET-DISK-PATH, *SET-STAT*, MOVE, DEL |
| link | GET-BRANCH-ID, GET-STAT, GET-DISK-PATH, *COW, COW-LINKS, SET-STAT*, GET-WRITEABLE-DISK-PATH, link, LINK-IN, *HASH-DUMP-UPDATE* |
| chmod | GET-BRANCH-ID, GET-DISK-PATH, GET-STAT, *GET-WRITEABLE-DISK-PATH, SET-DISK-PATH, COW, SET-STAT, chmod* |

**Continued on Next Page...**

**Table 4.1** – Continued

| FUSE call backs | RadFS operations |
|---|---|
| chown | GET-BRANCH-ID, GET-DISK-PATH, GET-STAT, *GET-WRITEABLE-DISK-PATH, SET-DISK-PATH, COW, SET-STAT, chown* |
| truncate | GET-BRANCH-ID, GET-STAT, *GET-DISK-PATH, COW, COW-LINKS*, truncate, *SET-STAT* |
| ftruncate | ftruncate |
| utimens | GET-BRANCH-ID, GET-DISK-PATH, GET-STAT, *GET-WRITEABLE-DISK-PATH, SET-DISK-PATH, COW, SET-STAT, utimes, HASH-DUMP-UPDATE* |
| open | GET-BRANCH-ID, GET-STAT, *GET-DISK-PATH, COW, COW-LINKS, SET-STAT*, open, *HASH-DUMP-UPDATE* |
| read | pread |
| write | pwrite |
| statfs | GET-BRANCH-ID, GET-DISK-PATH, statvfs |
| flush | close(dup) |
| release | close, *HASH* |
| fsync | fsync |

The branch ID of the target file or directory in each file operation and the type of operation decides which RadFS metadata operations need to be performed to achieve read-only sharing and COW semantics. As can be noticed in Table 4.1, almost all filesystem operations start with a GET-BRANCH-ID operation. This gets the branch ID associated with the target file or directory on which the operation needs to be performed. This branch ID obtained from the metadata server is compared with the branch ID of the virtual filesystem handling the filesystem operation. If the operation is a read operation (*access, readlink*), the GET-DISK-PATH operation gets the disk path, be that on the ondisk read-only path or the read-write path associated with the virtual filesystem. If the operation is a write operation (*mknod, mkdir, symlink, rename, link, truncate, open*), and the branch ID is not

of an ondisk branch that is read-write, the GET-WRITEABLE-DISK-PATH operation creates the heirarchy of directories leading to the target file or directory that needs to be written to or created in the read-write branch associated with the virtual filesystem. This may include copying of the file (a RadFS COW operation) from the read-only disk branch to the read-write disk branch in the case of operations that modify an existing file (*open, truncate, link*). After the copying, the filesystem call is issued to the newly copied file. In the case of filesystem calls that create files (*mknod, mkfifo, mkdir, symlink etc.*), a GET-WRITEABLE-DISK-PATH operation creates the directory structure in the ondisk read-write branch and then the particular filesystem call is issued to create the file or directory in the read-write branch.

RadFS's content hashing mechanism and associated file sharing dictates that the metadata for files that are shared also include the file attributes that get modified - uid, gid, mode, access time, modify time and change time. Thus the stat call for a regular file leads to the GET-STAT RadFS operation that stats the ondisk file, possibly linked to the content hash dump, and then overlays the above mentioned attributes values with that obtained from the metadata node for that particular file. This addition of file attributes to the metadata node is only for regular files as only regular files are hashed and dumped into the content hash dump. The other files are either shared read-only, with changes being made only to the access time file attribute or copied on write to obtain a private copy, and directories are created privately for each branch. The addition of file attributes to the nodes representing regular files entails maintaining/updating those attributes using the SET-STAT operation in those filesystem callbacks that lead to a change in the attributes (see Table 4.1).

The DEL operation and the GET-WRITEABLE-DISK-PATH operation are the opposites of each other - DEL deletes nodes from the radix tree metadata structure while GET-WRITEABLE-DISK-PATH inserts nodes. DEL requires special mention as in some cases it need not be backed up by a filesystem call, be that *unlink* or *rmdir*, and so filesystem errors need to be generated by RadFS when necessary. This happens when a file or directory on the read-only branch is deleted. In this case, only the corresponding metadata node should be deleted, and since a DEL operation always succeeds, permission checking should be done before DEL and

filesystem errors might have to be returned (see Section 4.1.3.1).

The GET-CHILDREN operation is to create a directory listing (*readdir*) and it returns a list of subdirectories and files of a given directory. The FUSE library has a *filler* interface that feeds a buffer that is used by it to build the *readdir* response. Since a RadFS virtual filesystem directory might have files or subdirectories from multiple ondisk branches, each file or subdirectory needs to be separately *stat*'ed and fed to the *filler* function. This involves getting the disk path (GET-DISK-PATH) of each file/sub-directory followed by the *stat* call. Since the *readdir* operation is heavily used, this is optimized so that the metadata operations performed are as unified and streamlined as possible. As much processing as possible is done without having to communicate over sockets unnecessarily. Thus GET-CHILDREN groups the multiple GET-DISK-PATH operation results into one big buffer that is sent as a whole to the FUSE application which parses it and *stat*'s the disk paths obtained to build the *readdir* response.

The SET-DISK-PATH operation sets the disk path of a node. Normally the disk path of a node is set when the node is inserted during a write operation through GET-WRITEABLE-DISK-PATH. But in the case of *rename* of a read-only file or directory, its only the RadFS metadata that needs to be changed and not the disk path as the rename callback is not backed up by a backing filesystem *rename*. In the case of a *rename* operation on a read-only branch, the metadata change is effected by inserting a new node into the read-write branch of the virtual filesystem and then setting its disk path to be the same as that of the read-only branch node's disk path, using SET-DISK-PATH and flagging the node as read-only (even if on the read-write branch). This is followed by a DEL operation that removes the read-only node from the current virtual filesystem and the *rename* operation is complete. *rename*s of virtual filesystem directories that are on the read-only branch need extra processing as the new node created on the read-write branch should also be linked to its files and sub-directories. This is done by the MOVE operation which ensures that the renamed directory has all its sub-directories and files linked in (see MOVE in Section 4.2.2). A *rename* on a read-write branch file/directory involves a filesystem backed *rename* call which does an actual rename on disk from the source path to the destination path, in addition to the usual updates in metadata.

SET-DISK-PATH is also used in the *chmod, chown, and utimes* callbacks when the callback is for an operation on a regular file on the read-only branch of a virtual filesystem. As mentioned earlier, content hashing and the associated sharing happens only for regular files and thus filesystem attributes (*uid, gid, mode, atime, ctime, mtime*) are maintained as part of RadFS metadata only for regular files. For other files, and directories, the callback is backed by the equivalent filesystem call which updates these attributes on disk. To update the file attributes for a read-only regular file, a new node is created using GET-WRITEABLE-DISK-PATH but with the disk path set to the read-only branch node disk path using SET-DISK-PATH. Thus a later GET-STAT operation would *stat* the file from the read-only branch disk path and then overlay the stat information with the attributes saved in the corresponding node on the read-write branch. The *chmod, chown, utimes* filesystem callbacks in RadFS FUSE all perform a COW operation in the case of a non-regular file or a directory.

The LINK-IN and COW-LINKS operations are needed to support hard links in RadFS virtual filesystems. Hard links are difficult to handle in copy-on-write filesystems. A file hard linked to another is distinguishable from the other only by virtue of its absolute path. A hard link is another name for a file, and thus hard linked files have the same attributes, including the same inode number. Thus hard linked files share the same data but without having any easy mechanism to know which other file is sharing data with a particular hard linked file. Thus a copy-on-write operation performed on a hard linked file would lead to problems, an example of which is described by the following scenario.

File *foo* and *bar* are hardlinked to each other in the read-only branch of a particular virtual filesystem. A write operation is performed on file *foo* that leads to it getting copied over to the read-write branch of the virtual filesystem. This breaks the expected behaviour of *bar* following *foo*'s changes as *foo* in the read-write branch is not linked to *bar* anymore. So any application that expects this behaviour (write to *foo* and expect the changes to be reflected in *bar*) breaks.

Solitude [10] handles the problem of hard links in its copy-on-write implementation by having a table that maps Solitude's metadata nodes to each other based on inodes. RadFS handles hard links by having back pointers in hard linked nodes that link them together to form a 'hard link' list. The LINK-IN operation in the

*link* callback adds the newly hard linked file's metadata node to the hard link list. When a hard linked file needs to be copied on write, RadFS copies the particular file and also creates hard links to it in the read-write branch using the hard link list (see Section 4.2.2.4 for more details). This is done by the COW-LINKS operation once it is known that a particular file is hard linked with other files which can be checked by using the *nlink* (number of links) attribute in the stat information of a file.

#### 4.1.3.1 Permission checking and Error handling

Since RadFS maintains virtual filesystems, it also has to deal with virtualizing permission checking and error handling in those cases where the backing filesystem isn't used at all and because a virtual filesystem is made up multiple branches. In some callbacks, for e.g., *mknod*, the filesystem call is re-issued with the actual disk path and any error thrown by the backing filesystem call can be returned. But even in this case, because of multiple branches, the basic EEXIST error has to be generated by RadFS FUSE if the path already exists in the read-write branch or the read-only branches that make up the virtual filesystem. Similarly an ENOENT error has to be returned if the path is not found in any of the virtual filesystem branches. In the case of *unlink*, an EISDIR error should be returned if the path leads to a directory. This as mentioned earlier is necessary as in the case of a read-only branch, the filesystem call *unlink* isn't invoked and so the DEL operation will succeed even if the node is a directory. In the case of *rmdir*, an ENOTEMPTY error should be returned if any of the virtual filesystem branches has an entry that is part of the directory to be removed.

The error that needs to be handled the most is EACCES, or insufficient permission. Each RadFS FUSE application taking care of a virtual filesystem is run with super user privileges. This means that any operation performed within a particular RadFS FUSE application would be as the root user who has almost unlimited privileges. FUSE provides a *fuse_get_context* interface to get the context in which the current filesystem call being handled was executed. This context can be used to extract the user ID (uid) and group ID (gid) of the user performing the operation. Thus every filesystem operation performed within RadFS FUSE needs to

be wrapped in setuid and setgid calls which set the uid and gid to the values obtained from *fuse_get_context*. This drops the super user privileges and delegates permission checks to the backing filesystem.

For regular files, which are shared through the content hash dump, the uids and gids will be that of the super user as its the RadFS hash server, again run with 'super user' privileges, that manipulates the content hash dump linking. Thus for regular files, permissions need to be checked against the uid, gid and mode retrieved from their corresponding metadata nodes. The uid and gid retrieved using GET-STAT is compared with the uid and gid retrieved using *fuse_get_context* and this coupled with the mode retrieved is used to check if the operation is permitted.

In the case of *unlink* and *rename*, POSIX states that if the directory containing the file to be deleted or renamed has the sticky bit set, and if the effective uid (obtained from *fuse_get_context*) doesn't match either the containing directory's uid or of the file/directory to be deleted/renamed, and if the initiator of the rename or delete isn't the super user, then an EACCES or EPERM error has to be returned. This is also handled by RadFS in addition to general permission checking.

In the case of an error, RadFS also has to roll back metadata changes it made prior to performing the filesystem backed call that failed.

#### 4.1.3.2 Content hashing

RadFS FUSE applications initiate the content hashing process for regular files associated with each of their virtual filesystems by sending a request to the RadFS hash server. The content hash request is sent whenever a file is closed which is done in the *release* FUSE callback. After the *close* filesystem call, RadFS FUSE sends a content hash request to the RadFS metadata server with the virtual filesystem path and the virtual filesystem's branch ID. The metadata server checks if the file is regular and if the node representing the file is on the read-write branch of the virtual filesystem and if so, forwards the request with the actual disk path of the file to the RadFS hash server. The RadFS hash server generates a content hash and links the file into the content hash dump. (see Section 4.3)

A file in the content hash dump can be shared by multiple virtual filesystems and any file that is shared should be copied out to the virtual filesystem ondisk read-

write path for modification. The RadFS FUSE application, in addition to checking whether a particular file is present in the read-only branch also checks whether it is being shared by some other virtual filesystem using the content hash dump, in which case the file is also considered read-only and the same read-only COW semantics apply even if the file is present in the read-write branch of the virtual filesystem. Since files end up being shared by being hard linked to each other, a way to check if a file is shared is if its hard link count attribute *nlink* is greater than one. If that is the case, and if the file isn't hard linked within the virtual filesystem (which can be checked by seeing if the node corresponding to the file is part of a hard link list), then the file is shared among more than one virtual filesystem and should be considered read-only. This reasoning is done in the GET-BRANCH-ID operation which returns either the read-write branch ID of the virtual filesystem or a value that indicates if the file is on the read-only branch, is hard linked in the virtual filesystem or is shared using the content hash dump. The virtual filesystem managed by RadFS FUSE then acts accordingly to preserve COW semantics and sharing using the content hash dump.

#### 4.1.3.3 Miscellaneous implementation details

An issue that had to be considered when implementing the *symlink* and *readlink* callbacks was that of absolute paths in the root filesystem escaping the mount point and referring to the backing filesystem's directories and files. The RadFS FUSE application handles all filesystem calls that are directed to the mount point of the virtual filesystem that it maintains. So a virtual filesystem mounted at */mnt/vfs1* works fine when the paths are relative to the mount point. But when a virtual filesystem is populated with a root filesystem image which has symlinks with absolute paths, a *readlink* gives an absolute path which points to outside the virtual filesystem. The basic problem is that a path in the root filesystem mounted at a particular mount point is not absolute with respect to the mount point. But this is easily fixed in RadFS because of the way it exports the virtual filesystem via NFS to Xen virtual machines. This creates a 'chroot jail' kind of environment with the root fixed as the mount point and any path being absolute with respect to the mount point.

Another interesting fact worth mentioning about POSIX filesystem rules is in the implementation of the *rename* callback for hard linked files. When a file *foo* is hard linked to another file *bar* and a *rename(foo, bar)* filesystem call is executed, the file *foo* will still exist even though *rename* returns 0. This is the expected behaviour as set out by POSIX but breaks certain filesystem tests which expect the source file in the *rename* operation to get deleted after the *rename* succeeds with a return value 0. The callback for *rename* in RadFS FUSE checks if the source file exists after the *rename* operation, and if so deletes it as is done by *mv*.

## 4.2 RadFS metadata server

The RadFS metadata server manages the radix tree metadata structure and satisfies requests from the RadFS FUSE application and the CLI using the UNIX socket interface. The metadata server is single threaded and handles multiple connections using *select*. It also handles persistence of the virtual filesystems by keeping a log of all operations that modify the metadata tree and writing the whole radix tree to disk whenever the operation log grows beyond a specific size. In the beginning, when the server is initialized, the base read-only branch is built up from the ondisk root filesystem image, the path of which is specified as a command line argument to the RadFS metadata server application. Thus when the server is initialized, it has one branch with branch ID 0, that forms the base read-only virtual filesystem that other virtual filesystems can use as their read-only branch. If the content hashing mechanism needs to be enabled, a user can instruct the server to initialize the content hash dump through the CLI (see Section 3.5). The RadFS FUSE applications check for the presence of the content hash dump and if present, uses the content hashing mechanism. The server then waits for requests from either the RadFS FUSE or the CLI applications. The following subsections describe how the various radix tree operations supported are implemented in the server along with details about how persistence is handled and how snapshots are created.

### 4.2.1 Radix tree

The radix tree used for storing metadata for the different virtual filesystems is made up of a base 'gold' read-only branch which is initialised at server start-up by pars-

ing the base root filesystem in an EXT3 directory. Other metadata branches are built on this read-only branch and the other branch nodes get added to the radix tree using the read-only branch nodes. This is done by attaching the new branch nodes to the base 'gold' branch node using the branch pointers described in Section 3.2. Whenever there is a file modified or a file or directory created in a read-write branch representing a virtual filesystem, the corresponding node is added to the particular branch, either by adding it as a child to an existing read-write branch node of the same branch ID or attaching it to the corresponding node in the read-only base branch using the branch pointers. Thus the metadata radix tree starts from a single read-only branch that represents initial virtual filesystems. With file modifications and new file creations, new branch nodes get added to the base gold branch to represent the modified virtual filesystem; we call this the 'growth-on-gold' model. Removing a node either results in creation and attachment of a 'white-out' node to a 'gold' node in the case of a node in the read-only branch, or an actual removal of the node if the node is part of a read-write branch. Thus at any point, the radix tree consists of branches representing the various virtual filesystems, and containing nodes that are read-only or read-write (depending on the virtual filesystem branch ID and the node's branch ID) or white-outs. Insertions, deletions and other radix tree operations necessary used by RadFS FUSE applications to update the metadata for the various virtual filesystems are described in the next section.

### 4.2.2 Radix tree operations

The core operations performed on the radix tree metadata structure (described earlier in Section 3.2) are *rtree-search, rtree-insert* and *rtree-delete. rtree-search* takes a virtual filesystem path, parses it using the radix tree and returns the target file/directory node if found. *rtree-insert* is used to insert a file or directory node into the tree and fix up its branch, child, parent, sibling and, if necessary hard link pointers (see Figure 3.2). *rtree-delete* is called when a file or directory is deleted or renamed and deletes the corresponding metadata node with corresponding linkage fix ups or creates a white-out node if the branch is read-only.

#### 4.2.2.1 *rtree-search* operation

Traversing the radix tree for a particular path involves breaking the path into path segments, starting from the root directory '/' and leading up to the target file or directory of a given branch based on the branch ID. When a virtual filesystem is created, only the root directory '/' is created with its child pointer pointing to the child node of the base read-only branch's root directory. Thus a virtual filesystem created would initially be identical to the base read-only filesystem as any look-up beyond the root node leads to the read-only branch nodes. *rtree-search* of a particular path starts at the root node and ends when the whole path has been parsed or the parsing fails at a particular level. The basic algorithm is described in Algorithm 1.

```
Algorithm: rtree-search
Input: Root node (cur), path to be parsed (path), branch ID (br), 'insert or
       look-up' flag
Output: node corresponding to path target or to the last path segment of
        path succesfully parsed
path-seg = get-first-path-segment (path)
while there is a sibling node for cur and cur's path-seg != path-seg do
 | cur = cur's next-sibling
end
if cur's path-seg != path-seg then
 | return cur's parent
else
 | if the search is for insert and branch ID of cur != br then
 |  | return cur's parent
 | end
end
remove path-seg from beginning of path
if path has more path-seg then
 | rtree-search(cur's child, path, br, flag)
else
 | return cur
end
```

**Algorithm 1**: rtree-search

Thus a search involves parsing the path a segment at a time, starting from the

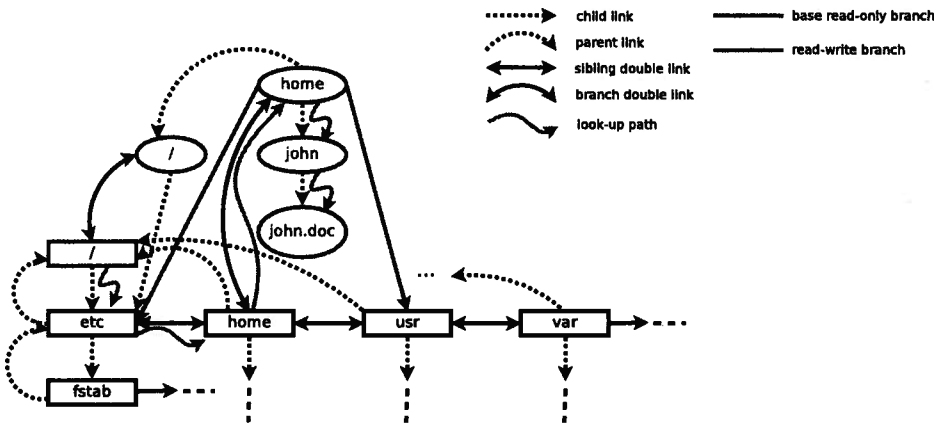


**Figure 4.2:** rtree-search for /home/john/john.doc

root. A search operation can be either for insertion or for checking the existence of a particular path. Levels (directories) in the filesystem hierarchy are traversed through child pointers while the sibling pointers are used to search within a directory level. A node returned in the case of searching for insertion is guaranteed to have the branch ID of the virtual filesystem for which the search is being performed. In this case, if the node corresponding to the target file or directory isn't found, the parent node is returned so that the remaining path can be inserted at the parent node using *rtree-insert*. A node returned in the case of searching for existence of a path can have a branch ID that is of any of the branches that make up the virtual filesystem. Thus a search with flag set for insertion always returns a node associated with the read-write branch of the virtual filesystem. A search with flag set to look-up can return nodes from the read-only branch too. Figure 4.2 shows an example where /home/john/john.doc is looked up.

#### 4.2.2.2 *rtree-insert* operation

The *rtree-insert* function takes a node, a relative path and a branch ID as parameters and inserts the nodes corresponding to the directories leading to the target as well as the target file/directory of the path, recursively. As mentioned before, nodes are inserted into the tree based on a 'growth-on-gold' model wherein *rtree-insert* first tries to find a corresponding node in the virtual filesystem's read-only branch to

which it can attach the new branch node to, using the branch pointers. If no such read-only node is found, then the node is inserted as the child of the parent node with the same branch ID (corresponding to the directory that will contain the new file/directory being created) which is guaranteed to be present in the read-write branch due to the recursive nature of *rtree-insert*. The *rtree-insert* is invoked for inserting the section of a path that is not present in the radix tree for a particular branch at a given node. Algorithm 2 shows the basic logic used in *rtree-insert*.

The fixing up of the pointers of a newly created node is done so that the new node is connected to any virtual branch nodes of the virtual filesystem it is part of. The parent of the newly created node is always from the read-write branch of the virtual filesystem.

The child is always from the read-only branch because the creation of a child always follows the creation of its parent directory node in the read-write branch. This insertion of a node into the read-write branch of the radix tree is mirrored in the actual ondisk path for the read-write branch where the directories that lead to the file or directory corresponding to the newly created node are also created ondisk.

The branch pointer of the newly created node might link it to its corresponding read-only branch node and in that case the virtual paths of the read-only and read-write nodes are the same but backed by different ondisk directories.

The sibling pointers are updated so that they point to the next and previous sibling nodes in the read-write branch if they exist or to read-only branch nodes otherwise. In the case where next and previous read-write branch nodes are found to link to, those nodes' previous and sibling pointers are also updated so that if a look-up gets to a particular read-write branch node, it stays with the read-write branch as long as it can.

A white-out is a node that indicates the absence of a file or directory. It differs from a regular node by the fact that the disk-path field of the white-out node will be set to *NULL*. Updating white-outs during inserts requires changing the disk path from *NULL* to a valid value. Fixing up the updated white-out node's links is unnecessary as the node is already part of the read-write branch and would have been updated as part of the read-write branch updates. Figure 4.3 shows the nodes

```
Algorithm: rtree-insert
Input: Node at which path should be inserted (cur), path to be inserted
       (path), branch ID (br)
path-seg = get-first-path-segment (path)
create new node (node) with path-seg
if cur doesn't have a child then
    cur's child = node
    node's parent = cur
else
    parent = cur
    cur = cur's child
    while there is a sibling node for cur and cur's path-seg != path-seg do
        cur = cur's sibling
    end
    if cur's path segment != path-seg then
        node's next-sibling = parent's child
        replace parent's child with node
        fix up sibling, parent and child pointers for node
    else
        if cur's branch ID == br then
            found cur, a white-out node representing a deleted node
            discard new node and update white-out instead
        else
            found cur, a read-only branch node corresponding to node
            attach node to cur using branch pointers
            fix up sibling, parent and child pointers for node
        end
    end
end
remove path-seg from beginning of path
if path has more path-seg then
    rtree-insert(node, path, br)
end
```

**Algorithm 2**: rtree-insert

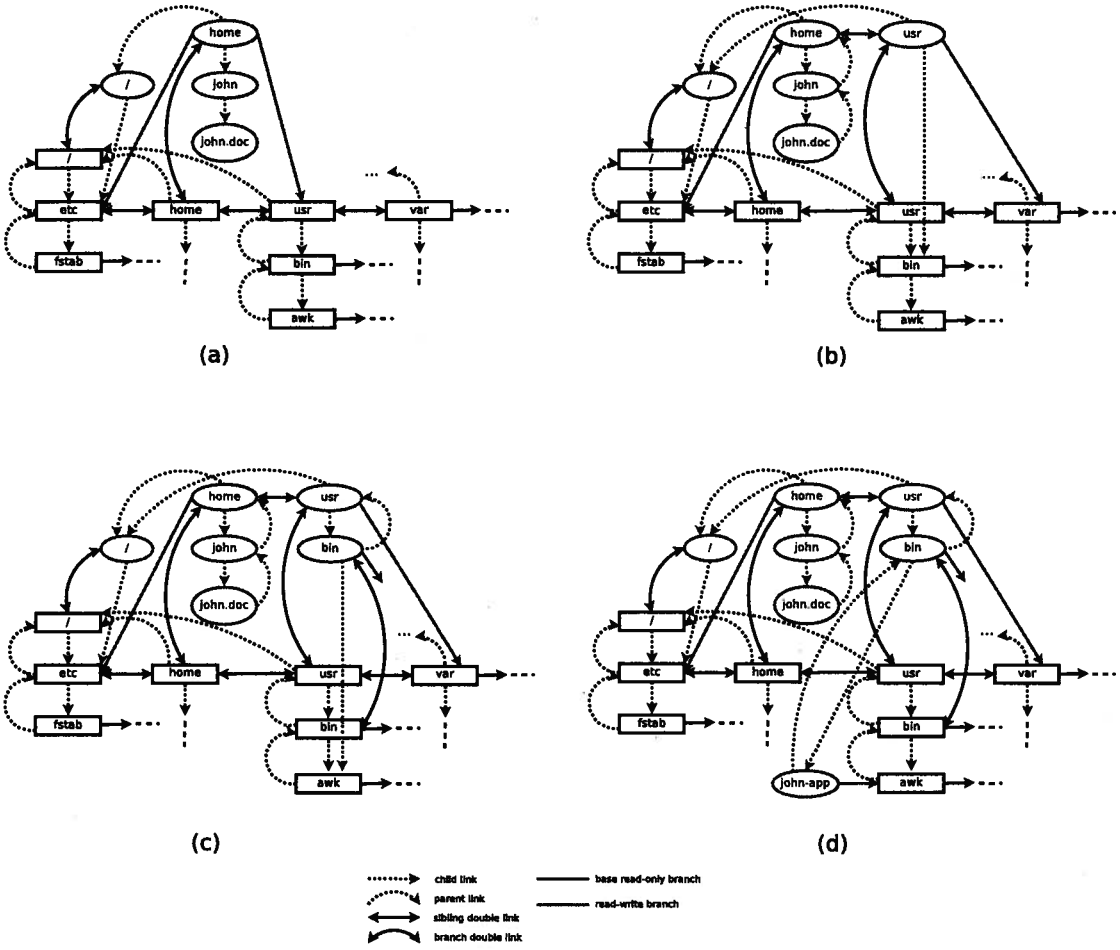


**Figure 4.3:** rtree-insert for /usr/bin/john-app

created in to the radix tree for inserting path */usr/bin/john-app*.

#### 4.2.2.3 *rtree-delete* operation

The *rtree-delete* operation deletes nodes from the radix tree. In the case of deleting a node in the read-only branch, a new white-out node having the branch ID of the read-write branch is attached to the read-only node via branch pointers. Any look-up for the node using the particular branch ID will thus lead to the white-

out signifying that the node has been deleted. In the case of deleting a node on the read-write branch but which is attached to a corresponding read-only branch node, the node is updated as a white-out node to prevent look-ups returning the read-only branch node. In the case where the read-write node does not have a corresponding read-only node in any of the read-only branches that make up the virtual filesystem, an actual deletion of the node is performed. The basic logic is described by Algorithm 3.

```
Algorithm: rtree-delete
Input: path to be deleted(path), branch ID (br)
node = rtree-search(root node of branch br, path, br, look-up)
if node's branch ID != br then
    create white-out with branch ID br fix up parent, child and sibling
    pointers of white-out
else
    if node's branch pointers != NULL then
        update node as white-out
    else
        fix up pointers to node delete node
    end
end
```

**Algorithm 3**: rtree-delete

Fixing up node links in *rtree-delete* is similar to fix ups in *rtree-insert* in the case involving white-out creation. A special case to handle arising from the structure of the radix tree is when creating a white-out for the first child of a particular node. In this case, it is necessary to check for a previous sibling of the read-write branch which won't be linked to the read-only node to be deleted through a previous sibling link. *rtree-delete* also needs to fix up the hard link pointers of a node that is to be deleted. Adding a node to the hard link list need not be done in *rtree-insert* as a specific *rtree-link* operation is implemented and used after the insert operation which adds the newly inserted node to the list of other nodes that it is hard linked to. Figure 4.4 shows rtree-delete for read-only node */usr/bin/awk*, read-write node */usr/bin/john-app*, and read-only node */etc/fstab*.

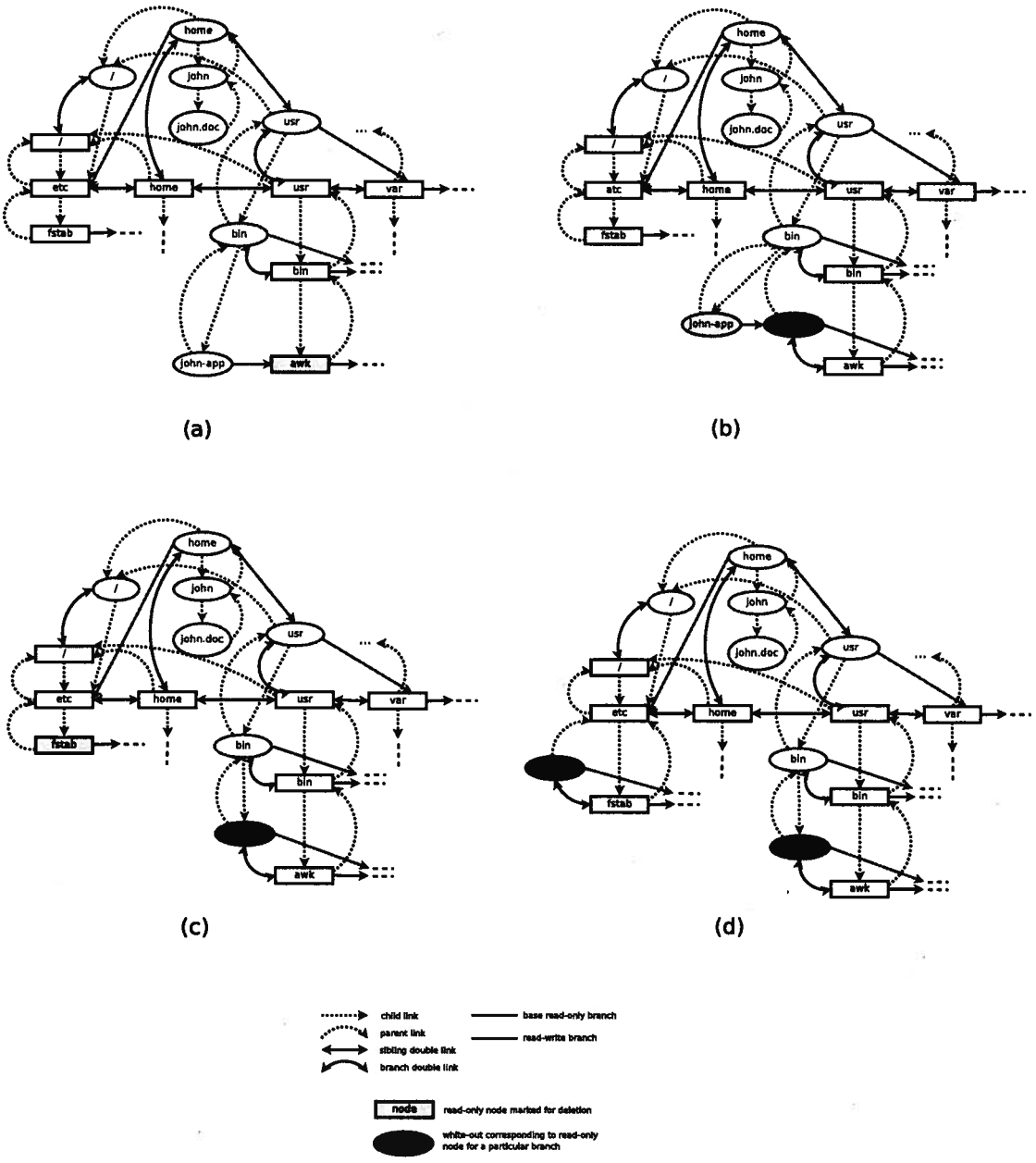


**Figure 4.4:** rtree-delete on read-only and read-write nodes

#### 4.2.2.4 Other radix tree operations

*rtree-link* links metadata nodes that represent files that are hard linked in a virtual filesystem. It is used after a hard link operation to add the node corresponding to the new file that was hard linked, to a list of nodes corresponding to the files hard linked to each other and now the new file. This list is used in copy-on-write operations for the files that are hard linked, to work around the problem of hard-link behaviour breakage described in Section 4.1.3. *create-hard-links* is a helper function that 'copies' hard linked files by copying the initial hard linked file that was written to from the read-only to read-write disk path and then creating the hard links to it using the hard link list associated with the initial file node.

*rtree-move* moves the nodes corresponding to files and sub-directories of a directory from under the corresponding directory metadata node to another directory metadata node. It is used after a *rename* filesystem call to link the nodes corresponding to the sub-directories and files of a directory to the new node representing the renamed directory. It also fixes up the parent pointers for all the file and sub-directory nodes to point to the new node. When a directory is renamed, only the disk path of the directory and the parent pointers of all its children (sub-directories and files) are updated. The disk paths of all the nodes below the renamed directory will still be pointing to the previous read-only or read-write disk location. In the case of read-only directory renames this creates no problem as the disk path should still refer to the read-only disk path as there are no corresponding read-write disk paths for the sub-directories and files as the rename is only in virtual filesystem metadata. But in the case of read-write directory renames, an actual *rename* backing filesystem operation is performed which leaves the disk paths of all nodes below the renamed directory inconsistent. For a read-write branch node, the path parsed to get to the node in the radix tree, suffixed with the ondisk branch root directory, gives the disk path for that particular node. The RadFS metadata server deals with inconsistent disk path in read-write paths lazily by updating the disk paths whenever there is an access to it for a read-write node.

*get-branch-id* searches for a node using *rtree-search* and if found returns the branch ID. It also returns values that specify if the file or directory associated with the node is read-only or hard linked. The hard link status returned is checked for

by the RadFS FUSE application to initiate the content hashing mechanism and will be described in Section 4.3.

*get-children* returns the names and disk paths of the children (sub-directories and files) of a particular node. This is used by the RadFS FUSE *readdir* function to fill a buffer for directory listings. *get-children* takes a path and branch ID as parameters, searches for the corresponding directory node, follows its child pointer to get all its children using the child and its siblings.

The *get-stat* function uses the filesystem call *stat* to get the attributes of a file/directory and then overlays that data with the attributes stored in the corresponding node in the radix tree. The *set-stat* function allocates a stat structure to store the *uid, gid, mode, atime, ctime*, and *mtime* attributes from a stat parameter passed to it. The set-stat function as opposed to the get-stat function is invoked only for regular files as in all other cases, all the attributes are obtained from disk.

The *get-disk-path* function searches for the node corresponding to a particular virtual filesystem path and then returns the disk path stored in the node. It also updates the disk path if the node is on the read-write branch and the path parsed to get to the node in the radix tree, prefixed with the read-write branch root disk path is different from the disk path stored in the node.

The *get-writeable-disk-path* function inserts the part of a path that is not present in a read-write virtual filesystem branch. This involves inserting hierarchically the nodes leading up to and including the target file or directory node and also creating the read-write branch's ondisk directory structure leading up to the target file or directory.

### 4.2.3 Persistence

Persistence of the virtual filesystem metadata is handled by the RadFS metadata server by maintaining a log of operations that modify the radix tree metadata structure. It also writes the radix tree out to disk when the number of log entries reach a predefined count. In the event of a crash and restart, the metadata server builds the base read-only branch again from disk by parsing the root filesystem directory and creating nodes for each file and directory contained in it. It then builds the other virtual filesystem branches by scanning the tree nodes written to the tree log and

then replaying the operations logged in the operation log.

The metadata server when processing requests that modify the radix tree, writes out a log entry for the request which includes a RadFS operation ID, the read-write branch ID of the virtual filesystem issuing the request and other parameters needed to re-execute the RadFS operation.

The tree log stores the nodes of all the virtual filesystem branches except the base read-only branch which can be built up from disk as it is read-only. The only part of a read-only branch node that is modified is the branch pointer which can be set when the corresponding other branch nodes get inserted into the tree during recovery. Each entry in the tree log is a serialized metadata node which includes the file or directory name, the branch ID, the disk path and also flags that indicate whether the node represents a regular file (the node includes file attribute information) or a hard link file (the node is part of a hard link list). Depending on the flags, additional information such as the serialized file attributes structure (for regular files) and the disk path of the previous node in the hard link list (for hard linked files) is also written to the tree log immediately after writing the related serialized node. The tree is written out in a depth first fashion exhausting all the branches of a particular node before moving on to its child node and then finally to its sibling nodes.

During recovery, the file/directory name, the branch ID and disk path obtained from the tree log are enough to insert and link the node into the tree. The flags are then used to do further reads of the tree log to set up the file attributes in the case of a regular file and to link it into the hard link list if it is a hard linked file. After all the nodes written out to the tree log have been restored into the radix tree, the restore procedure moves on to the operations log which would contain all operations performed after the radix tree had been written to disk. These operations when replayed in order is enough to bring the radix tree to the state it was in before the metadata server shutdown or crash.

### 4.2.4 Snapshots

RadFS supports light-weight snapshots. A snapshot of a virtual filesystem *A* makes it read-only and creates a new read-write virtual filesystem *A:timestamp* whose

state is that of *A* at the time of the snapshot. Thus a snapshot of a virtual filesystem in use is created in RadFS by marking the virtual filesystem read-only, creating a new read-write virtual filesystem which then becomes the virtual filesystem in use. In RadFS a snapshot request is initiated by the RadFS CLI application and handled by the RadFS metadata server.

Along with the response to every metadata operation requested by a particular RadFS FUSE application, the metadata server sends the read-write branch ID of the associated virtual filesystem. This is assigned to a virtual filesystem when it is initialized and is the handle used to identify it in all requests sent by it to the metadata server. The virtual filesystem uses its read-write branch ID to decide on copy-on-write operations. If the branch ID returned from a GET-BRANCH-ID operation doesn't match the read-write branch ID, the virtual filesystem assumes the file or directory associated with the path is read-only.

The metadata server processes a snapshot request for a virtual filesystem with a particular branch ID by marking the branch ID as read-only, creating a new virtual filesystem based on the virtual filesystem just marked as read-only and then responding to any further requests from the snapshotted virtual filesystem with the branch ID of the newly created virtual filesystem. This effectively changes the read-write branch ID for the virtual filesystem that was snapshot, making the previous branch ID read-only. The snapshot operation is atomic at the granularity of a RadFS operation as any subsequent RadFS operation leads to a change in the read-write branch ID of the virtual filesystem. The snapshot operation does not close open file descriptors and hence there will be an inconsistency in read-only semantics for those files that are open for writing during the time of the snapshot but this is limited to the time when the file gets closed. In the case of a database application that keeps a file open for long periods of time, this guarantee of read-only semantics on a *close* is unsatisfactory. To work around this, the RadFS FUSE application can check for a read-write branch ID change in each *read* or *write* call-back that does not involve a RadFS metadata operation. If a change in the read-write branch ID is detected, a partial copy-on-write operation of the file being written to can be performed and the file handle in the *read* or *write* updated with a handle to this copied file on the new read-write disk path.

## 4.3 RadFS Hash Server

The RadFS hash server maintains the content hash dump, processes requests for content hashing and is implemented using pthreads. The content hash dump is initialised using the CLI. The initialization involves parsing the read-only root filesystem ondisk and hashing regular files contained in it. This forms the base content hash dump that is shared in read-only mode by virtual filesystems. New files created or existing files that are modified are also hashed and added to the hash dump.

The RadFS hash server spawns a set of worker threads which handle content hash requests received from the RadFS metadata server. Each worker thread receives the disk path of the file to be hashed from the metadata server through a UNIX socket interface. Since a hash request is sent whenever a regular and writeable file in a virtual filesystem is closed, there can sometimes be multiple identical hash requests that are sent to the hash server. To avoid manipulating the hard links to the content hash multiple times unnecessarily, the server buffers requests (essentially disk paths to files that need to be hashed) using a binary search tree. The buffer stores current requests received and also being processed and thus coalesces multiple requests. A worker thread removes a disk path from the tree once the content hash for the corresponding file has been generated.

The SHA-1 hash of a file is generated using the GNU gcrypt library by splitting the file into 4KB blocks and creating a digest of their hashes. When the content hash has been generated without there being any further requests, the hash server thread sees if the content hash dump has a file with the generated hash as its name. If so, it replaces the file in the read-write branch of the virtual filesystem with a hard link to the identical file in the content hash dump. It does so atomically by creating a temporary hard link and then renaming it to the name of the file in the read-write disk branch of the virtual filesystem. If the content hash dump does not have a file with the generated hash as its name, a hard link is created in the content hash dump to the file in the read-write branch with its name as the 20 byte SHA1 content hash.

# Chapter 5

# Evaluation

RadFS is a virtual filesystem that builds on an existing filesystem to provide copy-on-write sharing, fast filesystem deployment with snapshots, and content hashing to save disk space. But this sharing and disk space saving should not be at a cost of significant performance degradation or more importantly, the correctness of operation of the filesystem. The following section shows how RadFS is evaluated for correctness and performance.

RadFS is backed by the ext3 [15] filesystem which is a POSIX compliant filesystem. Hence to check for RadFS POSIX compliance, a POSIX filesystem test suite called pjd [8] is used. The test suite has 1957 regression tests that check for POSIX compliance for the *chmod, chown, link, mkdir, mkfifo, open, rename, rmdir, symlink, truncate*, and *unlink* filesystem calls.

RadFS is built on FUSE, a user level filesystem library. Referring to Figure 1.1, each filesystem call to a virtual filesystem leads to 6 context switches. But since a file operation on average takes an order of magnitude more time than that taken for a context switch, the overhead of context switches associated with a filesystem operation is negligible. In RadFS filesystems are virtualized by using FUSE and by maintaining virtual filesystem metadata. Each file operation involves, in addition to a look-up of the backing filesystem metadata, a look-up of the virtual filesystem metadata too. Data transfer in a FUSE filesystem also involves copying of data more than once from user to kernel space but FUSE uses the kernel page cache by default which minimizes the associated overhead. To quantify RadFS metadata

and throughput overhead, the Bonnie++ filesystem benchmark is used.

RadFS is basically built for creating virtual machine filesystems, with a goal being quick deployment of multiple virtual machines. This necessitates that RadFS scales well when serving an increasing number of virtual filesystems. To evaluate RadFS scaling, an increasing number of virtual filesystems are spawned with each performing a common program compilation. This stresses concurrency and the RadFS metadata server's ability to handle multiple virtual filesystem requests in a timely and correct manner.

The following subsections go into more details about each evaluation with the corresponding results and discussion.

## 5.1 Test environment

All the tests were performed on an AMD Athlon64 X2 Dual Core Processor 3800+, with 2GB of RAM and a Hitachi HDT72503 320GB 7200RPM SATA disk drive. The Linux 2.6.27-rc6 kernel was used with the FUSE 2.8.0-pre1 library. The Linux kernel FUSE module supports NFS exports by default starting from version 2.6.27, but issues still need to be ironed out, one being an NFS stale file handle problem which arises due to the added layer of abstraction in FUSE filesystems. A file handle associated with a particular path can change underneath the NFS client as it is exporting a RadFS virtual filesystem that can switch paths from read-only to read-write. This causes a mismatch between handles expected and found leading to ESTALE errors. There is a solution in the works in FUSE that makes FUSE remember file handles indefinitely but it has not yet been implemented.

## 5.2 POSIX compliance using pjd

The results of running the POSIX pjd test suite on RadFS are shown below:

| Failed Test | Stat | Wstat | Total | Fail | Failed | List of Failed |
|---|---|---|---|---|---|---|
| /pjd/tests/chmod/02.t | | | 5 | 1 | 20.00% | 5 |
| /pjd/tests/chown/00.t | | | 171 | 10 | 5.85% | 36–37 68–69 83–84 141 145 149 153 |
| /pjd/tests/chown/02.t | | | 5 | 1 | 20.00% | 5 |
| /pjd/tests/chown/05.t | | | 15 | 2 | 13.33% | 11–12 |
| /pjd/tests/rename/01.t | | | 8 | 1 | 12.50% | 8 |

```
/pjd/tests/rmdir/02.t                    4    1  25.00%  4
/pjd/tests/truncate/02.t                 5    1  20.00%  5
/pjd/tests/truncate/12.t                 3    1  33.33%  2
/pjd/tests/truncate/13.t                 4    2  50.00%  2-3
/pjd/tests/unlink/02.t                   4    1  25.00%  4
Failed 10/166 test scripts, 93.98% okay. 21/1724 subtests failed, 98.78% okay.
```

The above listing shows the tests that failed to comply with the POSIX filesystem standard. Most of the failed tests are because of not including supplementary group permission checks during file/directory access. While FUSE provides an interface to get the *uid* and *gid* of a user using *fuse-get-context*, it does not include supplementary group information. A mechanism to get supplementary group information for a user (maybe using */proc/tid/task/tid/status*) needs to be built into each FUSE filesystem. This has not been currently implemented in RadFS. Other failures are due to a difference in error handling order between what is expected by pjd-fstest and what is done in RadFS. One example would be the test for ENAMETOOLONG return value for file or directory names which are greater than 255 characters. The test issues a filesystem call with a file name of length greater than 255 characters but it does so for a non-existent file. RadFS checks existence before passing it on to the underlying filesystem and thus these tests return an ENOENT (file not found) error as opposed to ENAMETOOLONG (file/directory name too long).

## 5.3 Throughput and metadata operation benchmark using Bonnie++

To evaluate and compare RadFS with other filesystems, the Bonnie++ filesystem benchmarking tool is used. RadFS is backed by the EXT3 filesystem. Hence to get a measure of RadFS virtualization overhead on EXT3, Bonnie++ results for EXT3 are compared with RadFS using FUSE over EXT3. Since the virtual filesystems would be exported to virtual machines using NFS, it also helps to quantify performance of RadFS over NFS. But with the interface between FUSE and NFS being still unstable, the test parameters for Bonnie++ had to be modified to get results without Bonnie++ failing. Hence the metadata operations benchmark for RadFS over NFS was for a set of 4000 files as opposed to 16000 for other tests. NTFS-3G

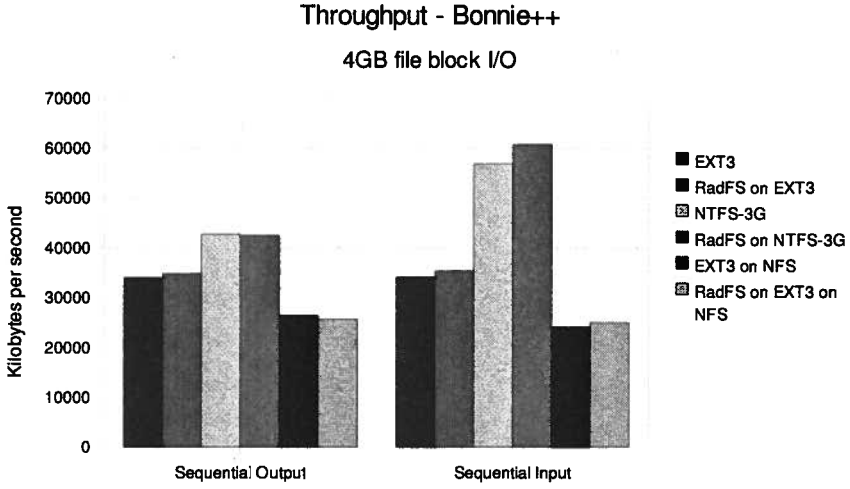


**Figure 5.1:** RadFS throughput

is another FUSE filesystem which provides support for NTFS filesystem mounting in Linux. It has been in development for more than 2 years, is quite stable and can be seen as a target performance base that may be reachable with optimizations to RadFS. Figure 5.1 and Figure 5.2 show throughput as well as metadata operation benchmark results.

As can be seen from Figure 5.1, RadFS throughput closely matches that provided by EXT3, NTFS-3G or any underlying backing filesystem. This is not surprising as FUSE and the latest Linux kernel have been optimized to perform read and write operations between user and kernel space efficiently with an optimized block size per transfer. FUSE also uses the kernel page cache which further eliminates overhead. The EXT3 performance lagging behind NTFS-3G can be attributed to journalling.

RadFS metadata operation benchmarks (Figure 5.2) show that the virtualization done by RadFS, requiring look-ups in the RadFS metadata structure needs to be optimized. Numbers for metadata operations for NTFS-3G and EXT3 are absent because they are too large to quantify according to Bonnie++. A major reason for the low numbers for RadFS in the create, delete and read metadata operations is because of the absence of any caching by RadFS in metadata look-ups. Each

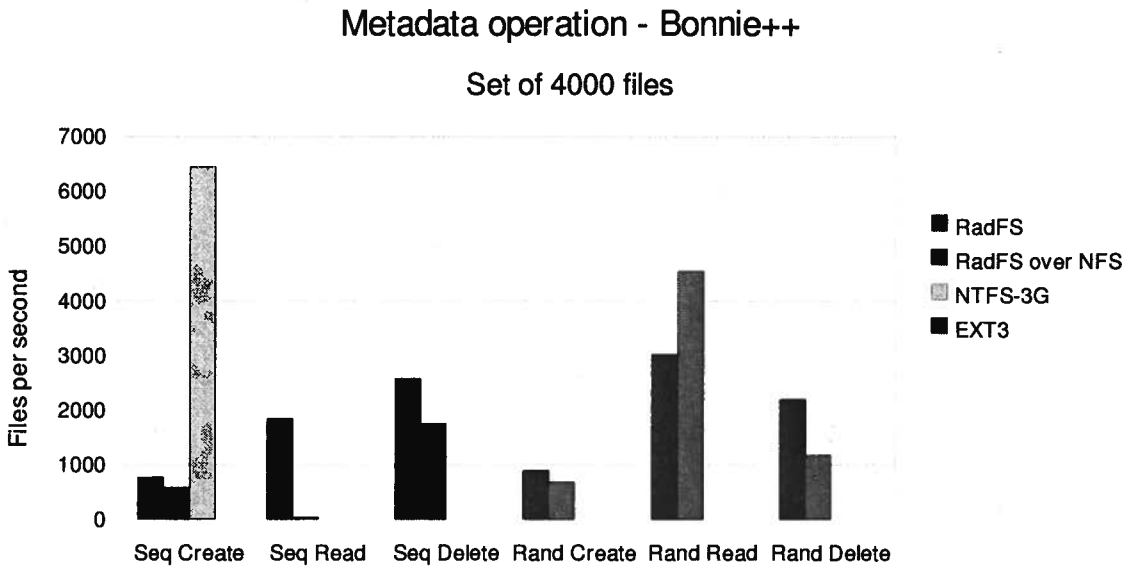


**Figure 5.2:** RadFS metadata operation benchmark

file operation requires the look-up of a disk path. This in turn requires the look-up of a RadFS metadata radix tree node representing the target file or directory over UNIX sockets. Caching needs to be done at the RadFS FUSE side as well as the RadFS metadata server side so that repeated look-ups of recently accessed files or directories are eliminated. The equivalent to a dentry cache should be built at the RadFS FUSE side so that disk paths are cached with cache invalidation following a rename, delete or other metadata modification operations. Since file attributes are accessed regularly, the data returned from the *GET-STAT* operation should also be cached and invalidated as necessary. At the RadFS metadata server side, the equivalent of an inode cache needs to be built so that repeated metadata node look-ups, involving parsing the radix tree can be avoided. This should not only eliminate redundant look-ups but also unnecessary interprocess communication and associated context switching over UNIX sockets between the FUSE application and the metadata server.

## 5.4 RadFS filesystem scaling

RadFS can host multiple copy-on-write virtual filesystems that share a common root filesystem and this sharing is key to saving disk space. The program compile test seeks to demonstrate scalability of RadFS in handling multiple, simultaneous virtual filesystem requests. The test includes compiling the MPlayer video player on a single virtual filesystem and then ramps the load up by having multiple virtual filesystems simultaneously hosting MPlayer compiles. With content hashing, the MPlayer package and extracted source occupies only the space needed by one copy as all the virtual filesystems share it read-only through the content hash store. Any temporary and object files created during compile are created in each virtual filesystem's private read-write branch. But these also get shared in the content hash store after hashing if they are identical. Since the MPlayer source is shared, the overhead of simultaneous compile should be set back by the use of a common page cache by the backing EXT3 filesystem. Figure 5.3 shows the results for the MPlayer compile test.

It should be noted that the compilation process is CPU intensive and there was CPU contention among the various *cc1* processes when the number of virtual filesystems performing simultaneous compile was increased to more than 4. The RadFS metadata server CPU utilization was dominated by the various *cc1* processes's CPU usage. Hence the scalability is CPU limited. For disk intensive or throughput oriented operations involving single files, RadFS performs on par with EXT3 as no metadata operations need to be performed after the file handle is obtained. Thus the only overhead is the context switches between RadFS FUSE and the kernel which can also be minimized by increasing the block size of each transfer in a read/write system call.

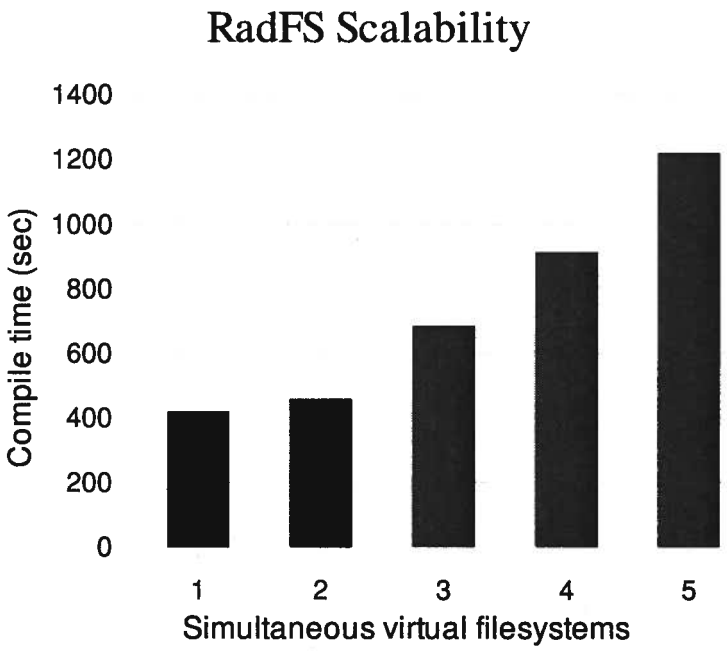


**Figure 5.3:** RadFS scalability

# Chapter 6

# Conclusions

We have built a prototype of a virtual filesystem that saves disk space through persistent sharing of data using a copy-on-write approach and content hashing. It caters to the need in virtual machines for storage that is quickly and easily deployable, supports snapshots and the ability to base one virtual filesystem off another recursively. The copy-on-write (COW) approach aids the creation of multiple virtual filesystems (or virtual machine disks) based on a common root filesystem that is shared among all the virtual filesystems. COW provides very quick deployment and snapshots and is the initial disk space saving mechanism through read-only sharing. Further and continued disk space saving is provided for by a content hashing module that maintains a content hash store which stores a single copy of any number of identical files across virtual filesystems.

RadFS throughput is at par with its backing filesystem EXT3 and since FUSE provides an abstraction at the Linux kernel VFS level, any filesystem supported by VFS can be used as a backing filesystem for RadFS. RadFS thus virtualizes a VFS filesystem and extends it to provide copy-on-write and content hashing mechanisms. Its implementation at the file level with the performance potential of NTFS-3G is proof that its possible to build a filesystem in userspace that provides various useful extensions without the fear of compromising kernel stability. This thesis shows that building a virtual filesystem with FUSE to extend existing filesystems to meet other requirements without too much overhead is definitely viable.

### 6.0.1 Future Work

RadFS is a 98.78% okay POSIX compliant filesystem that has data throughput equivalent to that of EXT3 or as mentioned before the throughput equivalent to that of any backing filesystem supported by the Linux VFS. Metadata operations need more improvement and so future work would involve providing metadata caching at the FUSE application side as well as the metadata server side. More optimization also needs to be performed to streamline the protocol used between the metadata server and the FUSE application. The FUSE-NFS interface is still nascent and once its standardized, more NFS performance oriented optimizations need to be worked out. Making RadFS 100% POSIX compliant requires support for supplementary groups to be built into either FUSE or RadFS.

# Bibliography


[1] Filesystem in userspace. http://fuse.sourceforge.net/.

[2] Gnu libgcrypt reference manual.
http://www.gnupg.org/documentation/manuals/gcrypt/.

[3] Microsoft single instance store.
http://download.microsoft.com/download/0/c/a/0cad7d83-2ef5-498a-af51-7911a10175b0/SIS_TWP.doc,
2008.

[4] qcow. http://www.gnome.org/ markmc/qcow-image-format.html.

[5] Qemu. http://bellard.org/qemu/.

[6] Unionfs-odf. http://www.filesystems.org/unionfs-odf.txt, .

[7] Unionfs - a stackable unification file system.
http://www.filesystems.org/project-unionfs.html, .

[8] P. J. Dawidek. Posix filesystem test suite.
http://www.ntfs-3g.org/pjd-fstest.html.

[9] T. Garfinkel and M. Rosenblum. When virtual is harder than real: Security challenges in virtual machine based computing environments. HotOS, 2005.

[10] S. Jain, F. Shafique, V. Djeric, and A. Goel. Application-level isolation and recovery with solitude. Eurosys, 2008.

[11] D. T. Meyer, G. Aggarwal, B. Cully, G. Lefebvre, M. J. Feeley, N. C. Hutchinson, and A. Warfield. Parallax: Virtual disks for virtual machines. EuroSys, 2008.

[12] B. Pfaff, T. Garfinkel, and M. Rosenblum. Virtualization aware file systems: Getting beyond the limitations of virtual disks. NSDI, 2006.

[13] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. FAST, 2002.

[14] S. Szabolcs. Ntfs-3g. http://www.ntfs-3g.org.

[15] S. Tweedie. Ext3, journaling filesystem, 2000.

[16] A. Warfield and J. Chesterfield. blktap - xen wiki. http://wiki.xensource.com/xenwiki/blktap, June 2006.