João Eduardo Luís

Licenciado em Engenharia Informática

TxBtrfs — A Transactional Snapshot-based File System

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Orientador: Prof. Doutor João Manuel dos Santos Lourenço, Prof. Auxiliar, Universidade Nova de Lisboa

Júri:

Presidente: Prof. Doutor Carlos Augusto Isaac Piló Viegas Damásio

Arguentes: Prof. Doutor Manuel Martins Barata

Vogais: Prof. Doutor João Manuel dos Santos Lourenço

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Novembro, 2011

TxBtrfs — A Transactional Snapshot-based File System

I love deadlines. I like the whooshing sound they make as they fly by.

— Douglas Adams

Writing is easy. All you do is stare at a blank sheet of paper until drops of blood form on your forehead.

— Gene Fowler

Acknowledgements

If it wasn't for a key set of people who provided a great deal of support and insight during the last year, this dissertation probably would not have been finished in the predicted timeframe. Among these stands out my supervisor, Prof. João Lourenço, with whom I've been working for the last four years and who always backed me up during my, ever so often, moments of panic due to some unforeseen situation, even if that meant for him to become sleep deprived.

I'm also obliged to thank my colleagues, and friends, who day-in, day-out, patiently nodded while they listened to me cursing the Linux Kernel's inner-workings, specially Nuno Galvão Martins who, by many times, also shared my pain regarding this particular subject; and to João Miguel Silva, who acted as my interim revisor, in the last weeks before the dissertation's delivery deadline, with an expediency and precision I can only admire.

Furthermore, I would like to thank all my dear friends for their kindness in leaving me alone whenever I had to work, although always supporting me in the background when things went south. Specially to Zé and Rui Jorge, who always had a pep talk ready during my down lows, a full-paid dinner is in order.

Lastly, but not (in any sense or form) least, my deepest gratitude to my father and my mother, who put up with my rather frequent mood-swings, and did it with the utmost comprehension and support; and to my brother, to whom I cannot thank enough for his unconditional predisposition to help me whenever some problem haunted me, and who always tried to help although his field of knowledge largely revolves around rocks.

This work was partially supported by the Centro de Informática e Tecnologias da Informação (CITI), and by the Fundação para a Ciência e Tecnologia (FCT/MCTES) in the research projects PTDC/EIA-EIA/108963/2008, PTDC/EIA-EIA/113613/2009 and Euro-TM COST Action IC1001.

Abstract

Several decades ago, the file system was the container of choice for large bulks of related information, kept in hundreds of files, and relying on applications specifically created to handle them. These configurations weren't scalable and could easily become difficult to maintain, leading to the development and adoption of Database Management Systems (DBMS). These systems, capable of efficiently handling vast amounts of data, allowed heavy concurrency without requiring the programmer to deal with concurrency-control mechanisms, by encapsulating operations within transactions.

The properties of Transactions rapidly became an object of desire by many, and efforts to bring them to general-purpose programming environments began. In recent years there have been breakthroughs in bringing the transactional semantics to memory, using Software Transactional Memory (STM), providing abstractions to concurrency-control on the application-level. However, STM failed to meet some expectations, specially regarding I/O operations, forcing the abstraction to go deeper in the system: directly to the file system.

In this document we shall discuss file systems in general, their properties and common structure, although focusing in those with transactional or versioning capabilities. Later on, we will present our proposed enhancement of an existing Linux file system (Btrfs), in order to offer transactional semantics to applications, while detecting potential conflicts between concurrent flows of execution and reconciling their changes whenever possible.

Keywords: File Systems, Transactional Semantics, Snapshots, Linux, Btrfs

Resumo

Há várias décadas o sistema de ficheiros era responsável por guardar grandes quantidades de informação relacionada, armazenada em centenas de ficheiros, e dependente de aplicações especificamente criadas para as manter. Estas configurações não eram escaláveis e complicadas de manter, levando à criação dos Sistemas de Bases de Dados. Estes, capazes de lidar eficientemente com grandes quantidades de dados, permitiam elevados níveis de concorrência sem que o programador recorresse a mecanismos de controlo de concorrência, encapsulando operações em Transacções.

As propriedades das Transacções rapidamente se tornaram um objecto de desejo, dando-se inicio a esforços para as trazer para os comuns ambientes de programação. Têm existido avanços na demanda por trazer a semântica transaccional para a memória, usando *Memória Transaccional por Software* (STM), e oferecendo abstracções quanto ao controlo da concorrência ao nível da aplicação. Contudo, as STMs não corresponderam a todas as expectativas, especialmente quanto às operações de I/O, levando as abstracções a descerem de nível e directamente para o sistema de ficheiros.

Neste documento discutiremos Sistemas de Ficheiros, as suas estruturas e propriedades, focando-nos naqueles que apresentam capacidades transaccionais ou multi-versão. Apresentaremos também as nossas propostas quanto à extensão de um Sistema de Ficheiros para Linux (Btrfs), de forma a oferecermos semântica transaccional às aplicações, detectando potenciais conflictos entre fluxos de execução concorrentes, e reconciliando as suas alterações sempre que possível.

Palavras-chave: Sistemas de Ficheiros, Semântica Transaccional, Snapshots, Linux, Btrfs

Contents

List of Figures

List of Tables

List of Algorithms

1

Introduction

1.1 The File System

From a user's perspective, the File System is the most visible system component, although the user may not realize its importance. The File System abstracts data storage and provides a logical view, organized in an intuitive way, and is responsible for ensuring correct data manipulation. As the user sees it, the File System is nothing but an aggregation of files and directories. Actually, that's pretty much what the user expects it to be, and the concept of *File System* is taken literally: it's a system to organize files into directories and subdirectories. In reality, there are some file systems that don't even support directories (usually called *Flat File Systems*), but they still keep files. The original Apple's Macintosh computer (Macintosh 128K), introduced in 1984 by its legendary Super Bowl commercial "1984", actually came with a Flat File System: the Macintosh File System. And people loved it. Nowadays, Flat File Systems are rare, as they've been superseded by the Hierarchical File System model, which is the most common file system model, aside from occasional purpose-specific devices that may not need the additional complexities introduced by a more complex file system.

Presenting a couple of files to the user, although it may seem pretty straightforward, is no easy task. There are forces at work behind that curtain that is the File System. The intensity of those forces have been increasing in the last decades, with the advent of modern computing and the increasing capacity of storage devices. The cost per MiB in magnetic storage devices has dropped remarkably in the last 15 years, as a 1 TiB storage device in 2010 is an order of magnitude cheaper than 1 GiB back in 1995 [Res, Iva]. The increasing capacity of storage devices, along with an increased access to data (for instance, the internet), cries for file systems able to meet current requirements, and earlier file systems just

couldn't handle the task. For instance, take Microsoft's FAT32 [Mica], introduced in Windows 95. Although it was a raging improvement on their previous file system (FAT16), it still suffered from severe limitations: files could be, maximum, 4 GiB in size, and the file system would be highly vulnerable to external fragmentation (which would decrease the file system's performance). Nowadays, we require more than 4 GiB to store a single DVD image, and with increased demanding on high-performance systems, file systems like FAT32 just don't meet the needs anymore.

1.2 Objectives

Concurrency is essential to most applications, and the implementation of concurrency control mechanisms is prone to simple mistakes that may cause a great deal of damage. Inspired by databases, mechanisms providing concurrent access to data without requiring the programmer to implement complex mechanisms have been surfacing. These mechanisms, called *Software Transactional Memory (STM)* and discussed later in Section 2.4.2, are usually distributed as libraries which applications may use. Programmers may then be able to enclose memory-related operations in transactions, disregarding concurrency control and focus on developing their applications. However, I/O operations in STM are still discouraged because they usually don't have transactional semantics. For instance, if one writes data to a file inside a transaction and then the transaction aborts, the data may very well stay written. Even if the STM implementation deals with such cases and compensates the operation, another process may have already read the contents of the file, which may lead to incorrect behavior. It becomes essential to support transactional I/O operations, in order to guarantee that file system updates inside a transaction are still run isolated from the remaining system until the transaction finishes.

Furthermore, when modifying the file system within a transaction, it becomes imperative to guarantee the file system is kept consistent from the application's point-of-view. For instance, an application that keeps its state in volatile memory cannot expect to recover it if the system crashes, since everything will be lost on power down. On the other hand, if the application keeps its state in stable storage, and the system fails during the time this state is being saved into one or more files, then the application's state may have been corrupted and impossible to retrieve as expected. This kind of consistency is not trivially obtained, and applications often resort to ad-hoc solutions, such as using temporary files, and later invoking `rename(2)` to replace files, albeit one at a time and still leaving room for failure. However, wrapping all the operations on a transaction working on the file system level, could provide such guarantees as the successful execution of all the transaction's operations, or none at all.

With providing transactional semantics to user-level applications in mind, we created TxBtrfs, an extension to a vanilla Btrfs file system supporting transactions, which allows applications to take advantage of the file system's transactional semantic on an opt-in basis.

1.3 Why Btrfs

Btrfs is one of many responses to the need of better file systems, adequate to modern contexts with high volumes of data while still maintaining good performance. This file system is still under heavy development, in constant change, and has a very active community around it. Where some would find reasons not to use it, we believe using an yet-unfinished file system may help us to introduce some new features into the field, while still taking advantage of an interested community eager to provide assistance and input.

We pondered on using a more stable and reliable file system, such as ext3, but we constantly met with additional features we had to develop in order to support our work. Btrfs, on the hand, already supports explicit snapshot creation, which eases our development by leveraging existing file system properties. However, Btrfs is not the only high-performance file system supporting snapshotting, as we will see further in this document (Section 2.2.3). Sun Microsystems' ZFS also provides snapshot capabilities, but its source code is a tenfold more vast than Btrfs'.

All things considered, we believe Btrfs is the right starting base for our work, and we are confident that by using such a high-profile file system will lead to more awareness on the potential of employing transactional semantics on the file system.

1.4 Contributions of this Dissertation

Following what has been discussed in this chapter, we propose a file system providing transactional semantics using a full-fledged Linux file system at its core, being the main contributions of this work:

- Transactional guarantees for operations over the file system's structure (such as `create(2)`, `mkdir(2)`, `unlink(2)`, among others);
- Transactional guarantees for file operations (such as `read(2)` and `write(2)`);
- A replica-based approach in order to guarantee isolation to transactions;
- An algorithm with the purpose of detecting conflicts between replicas; and
- An algorithm for replica reconciliation providing file system consistency, from the application's point-of-view.
- Adaptation of the *Store Benchmark* [SLL11, Pes11] in order to validate a transactional file system.
- Evaluation of the correctness and performance of an implementation of the proposed file system.

Furthermore, our implementation's prototype should become available to the community as open source.

1.5 Publications

A preliminary version of this work has been published in INFORUM 2011, 3.º Simpósio de Informática [LLL11].

1.6 Document Organization

Chapter 2 gives an overview on file systems and where does the interest in transactional semantics comes from. In detail, we explore several essential file system concepts, discussing the relationship between the operating system and the file system, as well as common data allocation schemes and consistency guarantees, while providing an historical overview of file systems since the 1960's. Later on, we introduce transactional semantics, some concepts, and why these are relevant in a file system context.

Chapter 3 will explore relevant details about Btrfs, namely its organization and capabilities taken advantage by TxBtrfs.

Our work towards a transactional file system based on Btrfs will be presented in Chapter 4, in which we will first briefly describe TxBtrfs, followed by a discussion on the relation between processes and transactions, as well as the relationship between transactions. We will also present our conflict detection scheme, and we will describe how we reconcile the file system in order to obtain a consistent state.

Finally, in Chapter 5 we will present the results of our evaluation, and in Chapter 6 we will provide a summary of this dissertation and present future work to be done on TxBtrfs.

2

The File System

2.1 Basic Concepts

When one speaks of a file system, one thinks of files and directories. Although they may be rebranded from system to system, whether there is a simple Command Line Interface or a full-fledged Graphical User Interface, these are transversal concepts to any file system. One also expects to be able to interact with both files and directories by using some supported operations. These operations may either be specific to the file system itself, or may be defined according to a standardized interface. In the following sections we will detail both file and directory concepts, and we will also discuss operations on different operating systems.

2.1.1 Files

The file system enhances the user-experience by abstracting the physically stored data into *Files*. These may be seen as the essential logical unit of a file system, and is perceived by the user as the only way to keep data in a storage device. Any data written in a file is, generally, directly mapped on a persistent (non-volatile) physical device by the file system, such as flash or magnetic drives. However, there are such file systems, such as *tmpfs*, that are considered volatile or temporary. These are not mapped onto stable storage devices, but instead onto volatile memory (RAM, for instance). As a consequence, any files created by the user in such a file system will be lost once the system either halts or reboots.

A file can contain *anything*. The information a file contains is defined by its creator, or by whoever accesses and changes the file. There is no reason why someone accessing a

file wouldn't be able to write something in binary format, even if the file's creator initially wrote plain ASCII. Of course, if the file was initially intended as a text document, in plain ASCII, and posteriorly some binary data was written to it, then it is possible that reading the file becomes a bit difficult. This is why, usually, files have their very specific purpose: text, graphic images, video, executable programs, etc.

What kind of information is contained by the file roughly is of any concern to the file system, but it is of use to the user. Traditionally, a filename is composed by two distinct parts: the *file's name* (or basename) and an *extension*, divided by a period character. The *extension* in the filename is usually associated with the file's type; for instance, *foo.txt* would be a file named *foo*, which would be a text file. Some operating systems do impose some restrictions based on file types, for both the user's and the system's sake, associating extensions with expected behavior or applications. Microsoft's MS-DOS and early Windows Operating Systems actively enforced the *basename.ext* filename format; for instance, executable files would have to be assigned an extension that would allow them to actually be executed (e.g., *exe* or *com*). On the other hand, on Unix-like operating systems, although allowed, extensions are not mandatory and are simply taken as aids to the user (and applications) determining a file's contents. Inferring a file's type on such a system is made using a *magic number* stored at the beginning of some files (e.g., executables, shell scripts, PostScript files, etc). However, not all files have magic numbers, and it becomes both the users and the application's responsibility to infer the file's type.

2.1.2 Directories

Aside from *Flat File Systems* we previously mentioned in Section 1.1, file systems support organizing files by means of *Directories*. These are merely logical units, as they don't keep any data, as files do. Directories are, in a way, analogous to a *folder*, containing related files (even if this is not mandatory). However, in a file system, these folders may also contain other folders (subdirectories), and so on[1]. Each file or subdirectory within a directory is considered a *directory entry*. In UNIX, directories may be taken as files themselves. However, instead of keeping data as files do, directories keep *directory entries* (*dentries*). Later, in Section 2.2.1 we will discuss this behavior with more detail.

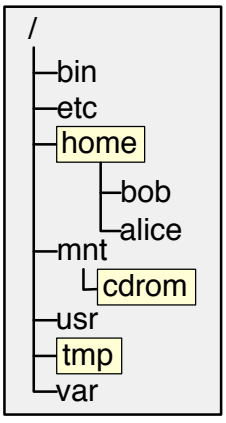


Figure 2.1: Example of a file system directory hierarchy with mounted file systems.

It is not uncommon, however, to use directories for more than simply organizing files. In a way, a file system may contain other file systems.

[1] Although, some file systems may impose restrictions on the number of possible subdirectories. Linux's Third Extended File System, for instance, only allows a maximum of 32 thousand subdirectories within any directory.

In Unix-like operating systems, it is frequent to have one or more file systems mounted in directories within the base file system. In Figure 2.1 we depict such a scenario, where we have the base file system, /, and within which we may have mounted three additional file systems: one in */home*, over NFS [WLS+85], containing the system's users' home directories; another in */mnt/cdrom*, which may be a ISO-9660 file system; and, in */tmp*, we may have mounted a RAM-based *tmpfs* such as the one described back in Section 2.1.1.

2.1.3 Operations over Files and Directories

Operating systems provide system calls to interact with the file system, allowing (at least) to create, write, read, delete and truncate a file. These are the most basic operations required in a file system, although other primitives may be required, or simply exist to facilitate files and directories manipulation — for instance, providing a system call to open a file in either read, write or append mode is common in operating systems.

With the purpose of standardizing the operating system's Application Programming Interface (API), including the file system's API, the Institute of Electrical and Electronics Engineers (IEEE) created the POSIX standards family. In fact, the POSIX (Portable Operating System Interface for Unix) was meant to standardize the APIs among Unix-like operating systems, but it can (and has been) applied to other operating systems as well.

The POSIX.1 standard (IEEE Std 1003.1) [Lew91] defines the API for file and directory manipulation in a POSIX-compliant file system, and this standard currently is either fully or partially supported by most Unix-like operating systems, including Linux, FreeBSD, Mac OS X or Solaris. Some non-Unix-like operating systems also support POSIX, although they may not support it natively; such is the case of Microsoft's Windows operating system, which supports a fully-compliant POSIX interface through *Interix* [Int], a subsystem part of the *Windows Services for Unix* (recently, *Subsystem for Unix-based Applications*) software package. Unlike a POSIX-compliant Unix-like operating system, in an operating system such as Windows, POSIX support is provided in user mode, running atop-the-kernel instead of in-the-kernel.

As our work relies on Btrfs, a Linux File System, we shall not digress and, in this document, we shall focus solely on POSIX-compliant APIs.

2.2 File System Structure

As it has been established by now, a file system is responsible for providing an interface to manipulate files and directories. However, we haven't yet explained how a file system actually works. The file system is very complex, creating the bridge between the user's logical view of data and the storage medium where this data is kept. From the moment a computer is booted, up to the moment it is halted, the file system will be performing write and read operations on and from the physical device it lives on. If the file system fails to perform these tasks graciously data may be lost. If the file system takes too long

to perform these operations performance may be affected. If the file system does not fit the task at hand, or if it is incapable of scaling to the needs of the system, then it may waste too much time reading and writing, definitely affecting performance.

Roughly speaking, implementing a file system is always associated with two main design issues: how the user should logically perceive the file system — files, directories, attributes, allowed operations —, and how to map the logical file system onto a physical device, creating the appropriate data structures and algorithms. Structurally, a file system may be decomposed in three essential components: the basic I/O interface, an organizational component, and the logical file system.

The first one is responsible for the interaction with the physical devices, and will be used to issue write and read requests on and from the device. In reality, the file system doesn't access directly the storage device to read or write data; that responsibility is delegated to a device driver. What the file system actually does is simply ordering the device driver to retrieve, for instance, a given block from disk. A *block* is a one or more disk *sectors*, and these last ones can be of various sizes, depending on the storage device itself, but 512 bytes is recognized as the *de facto* standard for magnetic disks — although optical disks use sectors with 2048 bytes and newer magnetic disks use sectors with 4096 bytes to improve efficiency. Within the file system, both the basic I/O interface and the organization component will be co-dependent, as the last one will be the one responsible for mapping logical file system blocks to physical disk blocks. Last, but not least, we have the logical file system, which actually relies on both the previous components to manage the file system's, and files' metadata. We consider metadata all of the file system's structures and informations, except for the actual file's data — directories, as well as file's attributes, are metadata kept by the file system —, and these informations may be both kept in the device as well as in primary memory, therefore the dependency of the logical file system on the other two components.

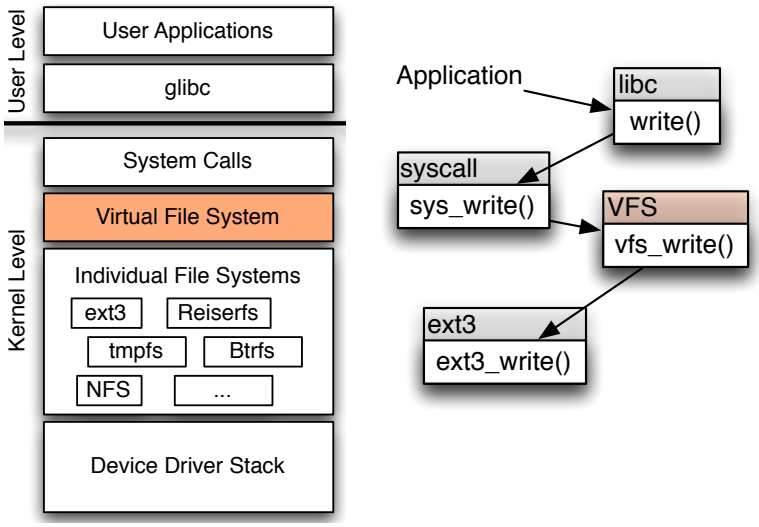
In early operating systems, all these components would be embedded in the file system itself. However, when an operating system supports more than one single file system, integrating them into a single logical file system view and allowing the user to seamlessly access each and every one of them, a more sophisticated approach must be taken to guarantee everything works as expected and without increasing the system's complexity or incurring into performance degradation.

To address this issue, modern operating systems offer two additional layers: a low-level layer for file systems to access device drivers, possibly providing caching mechanisms to avoid performance degradation; and a higher-level layer that abstracts the user from the actual file systems being accessed, providing a single API. This last layer is called the *Virtual File System* (VFS), and has been present in operating systems for more than two decades. Sun Microsystems' SunOS 2.0 (1985) had one of the first VFS implementations [MMS00], allowing to transparently access both UFS and NFS file systems. We will discuss the Virtual File System in Section 2.2.1, although solely focusing on the Linux's VFS as that's what's relevant to our work, and in the following Sections we will

be discussing common approaches to allocate data and guaranteeing file system consistency.

2.2.1 Virtual File System

The Virtual File System, as seen in Figure 2.2a, is a layer between the system calls and the implementations of any number of file systems the OS may support. Being positioned on that level, the VFS is in perfect conditions to receive the operations issued on the system call layer and delegate them to the appropriate file system (Figure 2.2b depicts this behavior).



(a) The Virtual File System in the System Stack

(b) Operation Delegation

Figure 2.2: The Virtual File System and Operation Delegation

With this structure, it is then possible for the VFS to offer a clean and generic interface to the system, despite how each file system is implemented. As long as each file system models their operations according to what the VFS is expecting, it is possible for multiple implementations to coexist and to be transparently accessible from anywhere in the system. From the VFS perspective all file systems look the same, and only the file system itself knows the implementation details. This abstraction is the reason why one given program is able to properly execute using either a local or a network file system. If additional file systems are added to the operating system, programs that may access them will not need to be rewritten.

We have previously discussed that file systems keep both data and metadata, but we did not discuss how these are kept. In a way, if we only had a limited amount of file systems supported by the operating system, both data and metadata could be kept in any form, as long as the operating system and the file system were able to understand the data structures involved. However, once we have a scheme like the VFS, it becomes imperative to standardize the data structures describing any informations the file system

may require. Basically, the VFS needs the file systems to provide files' and directories' metadata in data structures known to both — the *inode*; for data, however, there is no need, as it is accessed only during reads and writes, and those operations are directly delegated to the underlying file system by the VFS.

Inodes are logical aggregates of informations regarding files. From the VFS perspective, each inode will uniquely represent a file in a file system, along with standard informations expected by the operating system — in UNIX, these include the file's mode, access and modification times, and both user and group ids, for instance. In Linux, a VFS inode may also reference file system-specific informations, but that is the file system's choice and responsibility to maintain. Furthermore, and following UNIX's philosophy that "*On a UNIX system, everything is a file; if something is not a file, it is a process*" [Mac07], much of what composes the operating system may be represented by inodes — e.g., directories, block devices (any device supporting random access and seeking, such as physical disks or memory regions) or character devices (usually used for stream communications, as seen in sockets, virtual terminals or peripherals as keyboards and mice).

Accessing files is not a straightforward process. File system operations often rely on path names, which means the system must first translate each component of a path into each inode, before being able to access the desired inode's informations. The VFS will take each component of a path and will create, on-the-fly, an in-memory data structure called *directory entry* (*dentry*). Each dentry will associate a *name* with an inode, a parent dentry and a list of child dentries. Traversing a path will then be summed to obtaining the first dentry, and follow every path component until reaching the last dentry. Basically, by accessing */home/user/foo.txt*, the VFS will create four dentries (i.e., */*, *home*, *user* and *foo.txt*). As path lookup can be a time consuming process, in order to facilitate path based operations, the VFS keeps a cache of *dentry* objects. The cache is then consulted during path name translation, which may improve overall performance because reading an inode from an in-memory cache is faster than looking it up on-disk.

One must not finish this topic without discussing two very important subjects: how does the VFS knows which file system is mounted in a given directory, and how to deal with it. Fortunately, these are both answered with one single concept: the *Super Block*.

In a file system, the Super Block may be seen as the point-of-entry into the file system itself. It is kept in a predefined location within a volume, guaranteeing the file system's driver knows where to look for it once the VFS instructs it to, and it contains essential informations to use the file system as a whole. For instance, in the Second Extended File System (ext2), the Super Block is kept 1024 bytes from the volume's beginning; in Btrfs, however, it is kept 64 KB from the volume's beginning. The on-disk Super Block will provide the file system's driver with the essential informations about the file system's configuration (e.g., in Btrfs, sector size, node and leaf sizes) and the whereabouts of the file system's on-disk data structures (e.g., free blocks, inode's tables or system trees).

After being loaded by the file system's driver into memory during the mount process, the file system will be let known to the VFS. Although each file system may keep its Super

Block in a dedicated data structure, fitting its private needs, it will also register a generic Super Block data structure with the VFS. This data structure will contain non-file-system-specific informations required by the VFS to properly interact with the file system: which are the file system-specific methods to be called for each VFS operation, where is the file system mounted, the file system type, name and mount options, as well as a file system-specific data structure, much like what happens with the inode data structure. From this moment on, the system is totally capable of transparently issuing operations on the available, mounted file systems.

2.2.2 Data Allocation Methods

From a user's perspective, when we have a file system on a storage device we want to handle its data as fast as possible, but we don't want any storage space waste — in other words, we want time and space efficiency. Aside from device-specific characteristics such as magnetic disks' *rotations-per-minute*, data transfer rates and on-device data caches, of which we do not speak on this work, most of a file system's efficiency is obtained from careful and smart on-disk data handling.

Earlier file systems suffered from fragmentation and poor data handling, leading to severe performance degradation due to both waste of storage space and inefficient raw disk bandwidth usage. For instance, the first UNIX file system, initially called simply the *FS* but nowadays commonly known as *System V FS* (circa 1974), could only take full advantage of approximately 5% of raw disk bandwidth. Since then, file systems evolved and, as their internal data structures were adapted to different workloads and different amounts of data, became more powerful.

The biggest challenge in defining a suitable on-disk data structure refers to how the data and metadata are going to be allocated on the storage device. Several methods have been employed throughout the decades, and some are best suited to some tasks than others. We shall discuss those we consider the most relevant allocation methods up until now, contextualizing their uses and the performance impacts the file systems implementing them suffered.

From a performance perspective, the theoretically optimal allocation method would be *Contiguous Allocation* [MvRT+90], guaranteeing that each file occupies a contiguous set of on-disk blocks, requiring the minimal amount of seeks, which may severely impact performance on magnetic disks. However, this allocation method translates into external fragmentation of the file system, due the free space being broken into pieces, as files are created and deleted. Although Contiguous Allocation is optimal to read operations, this allocation method may become troublesome to write and to expand a file, because one must find enough contiguous free space to write the file. However, one should account for the eventual need to expand the file, unless one is willing to reallocate the file in the future if needed. Even then, upon reallocation, one may be unable to find any contiguous free space big enough to store the file, even if there is a lot of free space on disk. Ideally,

this type of data allocation is good for file systems with small, mainly immutable files. The only way around the external fragmentation problem imposed by this method allocation would be to defragment the disk, by compacting the files. Basically, one could reallocate every file, guaranteeing they would be as contiguous on-disk as possible, leaving all the free space either in the beginning or the end of the storage device.

Linked Allocation solves the problems raised by Contiguous Allocation, by considering a file nothing more than a list of allocated block segments. In this allocation method, one may think of the file as a singly-linked list: the file's first block segment is the head of the list, keeping a pointer to the next block segment if any, and so forth. This method avoids the file system's external fragmentation, as well as the issues with expanding the a file when there are no free contiguous blocks available. However, it does introduce a problem the first method avoided completely: Linked Allocation may introduce any number of seeks while reading a file, as the file may be scattered throughout the disk. Eventually, with the increased file creation and deletion, this problems tends to become a severe cause for performance degradation. This is a common issue with FAT file systems (FAT12, FAT16 and FAT32), and it may be fixed by defragmenting the disk, much like what was previously discussed for Contiguous Allocation. A simple approach to reduce the impact of this problem, avoiding the necessity of disk fragmentation, is to allocate *clusters of blocks*. Upon each allocation, instead of using the *block* as a unit, the file system allocates and reserves a *cluster*, which may be (for instance) composed by four blocks. This allows the file system to expand a file in the future, if any space is still left in the cluster, and the use of clusters also enforce higher locality of the file's on-disk data, thus avoiding further seeks.

Block clustering, though, introduces yet another problem: internal fragmentation. In fact, internal fragmentation was already present in both the discussed models, but then it was far less relevant. Using the block as the allocation unit, even if the file didn't occupy the whole block, the wasted space would be minimal. However, using block clusters may lead to an allocation of several blocks, of which the file only uses a small portion. In this case, internal fragmentation is much more severe, as it wastes much more storage space. A technique used to avoid the impact of internal fragmentation on a storage device is the implementation of *Block Suballocation* (also known as *Tail-Packing*), which consists in aggregating the tail (i.e., the last blocks) of multiple files into one single block, reducing the wasted space. Efficient heuristics may be employed, in order to guarantee that the blocks containing tails from multiple files are kept closely enough to the remaining blocks, avoiding further disk seeks to retrieve them. This technique is employed by several file systems in used, such as Btrfs and ReiserFS [Rob].

The simplicity of Linked Allocation fails to perform efficiently when direct access is required, since the pointers to the blocks are scattered throughout the disk and there is no direct way to retrieve them without accessing all the blocks, starting on the head. One solution to this problem is by using *Indexed Allocation*, which literally keeps an index on all the blocks used by each file. The concept is pretty straightforward: the file system

keeps a directory of files, each file associated with an *index block*; each index block will then keep an array of disk addresses for each block belonging to a file, each position corresponding to a block and kept ordered (i.e., position i corresponds to the i^{th} block of the file). Such simplistic indexation, however, limits the maximum size of the a file to the number of blocks that may be kept by each index block — using a 4 KB index block, containing 32-bit addresses, one would be limited to indexing 1024 blocks, which is equivalent to 4 MB files.

A common solution to increasing the maximum allowed file size is to keep a certain amount of *direct pointers* on the index block, which directly address some of the file's data blocks (useful for small files), and some *indirect pointers*, which refer to other index blocks. However, one should be aware that increasing the level of indirection does not infinitely increase the file size, as it is always bounded by the amount of bytes addressable by the architecture and the file system. For instance, with a 32-bit address size and 4 KB blocks, one will easily run out of addresses after the third indirection level, depending on the amount of direct pointers each indirection level holds. In any case, this is a common approach taken by file systems such as the Unix File System or the Second Extended File System.

Performance-wise, Indexed Allocation still suffers from the *scattered blocks* problem as Linked Allocation. Even though the index blocks may be cached in memory, avoiding seeks to retrieve them, this allocation method does not provide any characteristic to avoid spreading a file's blocks all over a volume. This is why file systems such as Btrfs, which aim at providing a reasonable solution to High-Performance systems, take a different approach, using trees [Rod08]. In Btrfs's case, all metadata is kept in trees, including informations regarding the location of a file's data. The nature of trees, along with the file system's policy, tries to keep all the pointers regarding a single file in the same tree leaf; if not, they will probably be in contiguous leaves. Each of these pointers will refer to an *Extent*, which is a group of contiguous blocks much like the *clusters* we have discussed with Linked Allocation. These extents, however, do not have a fixed size, and the file system may even allocate extents of one single block for a file that does not require more than that. The main objective of using extents is to guarantee that we actually allocate as much contiguous space as available, and no more that strictly necessary; if the file happens to grow, another extent will be allocated, although not necessarily contiguously to the previous. With this approach, Btrfs aims at reducing disk seeks, while keeping both external and internal fragmentation to a minimum. The internal fragmentations is also addressed with the usage of *Block Suballocation*, as previously discussed.

2.2.3 Consistency Guarantees

So far, we have discussed how file systems are handled by the operating system, and how the file system handles its data. We haven't discussed, however, how the file system guarantees its own consistency. Machines are prone to failure, either mechanical or

due to external factors, such as power outages. Even if we were 100% sure such issues would not arise, we would still be bound by Murphy's Law: "*If anything can go wrong, it will*" [Roe52]. Therefore, the file system must guarantee that, if anything goes wrong while performing an operation, it is kept in a consistent state.

A popular approach to avoid inconsistencies in a file system is by using a *Journal* (ext3, XFS, ReiserFS), kept on non-volatile storage, such as a magnetic disk, and onto which a set of changes to the file system (*Transaction*) issued by any operation are sequentially appended. Once a transaction is fully written onto the journal it is considered to be committed. At this moment, no file system data structures have been changed, but they are safe in storage. If the system happens to crash at this point, no changes have been totally lost: they may have not been applied to the file system, but it is possible to read them from the log. Eventually, the file system will replay the log, sequentially applying the changes and removing them from the journal. In fact, the log is nothing more than a circular buffer, and what the file system actually does is to update a pointer that reflects the last applied change. Log replaying after a crash may, however, raise issues when the system crashed before a transaction has been effectively committed. In such a case, when the system reaches the end of the log without finding an hint that the transaction has been committed, then it has to undo all the changes made while replaying the log — but it still guarantees that, although some information has been lost, the file system is kept in a consistent state. Journaling file systems usually provide three different types of journaling, depending on how tight one wishes to keep the consistency guarantees while considering the associated performance degradation:

Writeback Mode Only the metadata is journaled, while the data blocks are directly written to disk, without any order preservation. If the system crashes after the metadata has been journaled but before the data is fully written, unless the metadata has some way of validating the data (e.g., checksums), the file system can incur in silent data corruption — which is avoided by using *Ordered Mode*.

Ordered Mode The same as *Writeback Mode*, but metadata is only written to the log *after* the data has been safely written to disk. This method avoids silent data corruption, but it may not be able to fully write the metadata onto the log if the system crashes meanwhile. However, even if such happens, the file system is kept consistent.

Data Mode Both data and metadata are journaled. This guarantees full consistency, but may have a severe impact on performance, as each change is written twice: once in the journal, and another into the file system during log replay.

There are other alternatives to guaranteeing file system consistency besides journaling (soft-updates [MG99], for instance). However, aside from journaling, the technique that is most relevant to our work is *Copy-On-Write*, which basically keeps consistency by creating copies of blocks whenever these are changed. Take a transaction such as the ones

used in journaled file systems, but, instead of applying each change onto a log, blocks aren't really changed. Instead, each block that is supposed to be changed will, first and foremost, be copied into a new location on disk and only then will it be changed. This, by itself, does not guarantee consistency. However, if we are able to guarantee that all pointers in the file system are atomically updated to reflect the changed blocks once the transaction commits, we are able to guarantee full file system consistency. One can imagine the difficulty of guaranteeing atomic updates on several on-disk pointers, possibly scattered throughout the disk, as we would have if we were using an *Indexed Allocation* method — however, associating *Copy-On-Write* with a journaling file system can be a solution, as it was taken by *ext3cow* [PB05], an implementation of the Third Extended File System with *Copy-On-Write* properties.

Copy-On-Write is far more useful if we are dealing with a tree-based file system, such as WAFL [HLM94], ZFS [BAH+] or Btrfs [Rod08]. All of these file systems employ *Copy-On-Write* in order to guarantee full file system consistency. As they are based on trees, upon changing a block, they create copies of each block pointing to the changed blocks, and so forth up until the tree's root. Once the algorithm reaches the root, we have a shadow of a tree branch, with all the changed blocks. It is easier, then, to atomically change a single pointer, effectively changing the tree's root. As the algorithm only changes the pointers related to the changed blocks, unchanged blocks are never copied and their pointers are never changed. Basically, creating a shadow branch does not involve copying the whole tree. Once the tree's root is atomically changed, the file system may either free the old blocks, or keep them *free of charge* as state snapshot. Figure 2.3 illustrates how Btrfs handles *Copy-On-Write*, and should be noted that ZFS' works very similarly to what is presented in this figure.

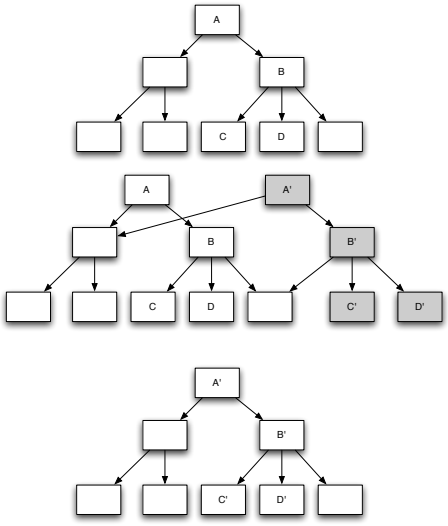


Figure 2.3: *Copy-On-Write* on Btrfs.

2.3 Historical Overview

Common sense dictates that people make mistakes. In a file system, mistakes can lead to unexpected results, such as data-losses. Nowadays, operating systems are usually packed with some kind of tool allowing users to safely delete their files without losing them by mistake. However, these utilities don't usually work on the overall file system level: on an *Apple* computer, deleted files using the Graphical User Interface will be moved to the *Trash*, but if one issues a *rm -fr* of a directory, on the terminal, it will be permanently deleted. This behavior can turn out to be catastrophic if one has *administrator* privileges on a system.

Over all these years, the best that most mainstream operating systems could do to mitigate these problems was some backup utilities. These may be rather useful for a user, providing timed and automatic file system snapshots, but any changes made in between snapshots will still be irreversibly lost. Backup utilities existed in the *server world* way before their introduction into the desktop. For instance, *rsync* [Way] is a differential backup tool that synchronizes files and directories between two locations, introduced in the early 90's and still in common use, and *rsnapshot* [Nat] is another tool, built on top of *rsync*, that provides snapshots instead of full backups. However, these tools lack both transparency and full-time tracking of changes.

Data loss can be mitigated by backup frequency. However, keeping up-to-date backups for a common user is not an easy task. Regular people simply don't backup data. Most believe their computer to be a flawless machine, able to sustain any kind of injury and completely immune to the world around it, but this is not the case. For the layman user, backing up data is either a superfluous or tedious process of machine-like behavior: put disk into tray, burn disk, get disk off tray, repeating the process as many times as needed. Nowadays we start to acknowledge some automatic backup tools in mainstream operating systems [App, Micb], but even their configuration may become an issue for the common user — lack of transparency takes its toll. When on a *server*, however, we must change the approach: backups are required, imperative; snapshots should be frequent in order to guarantee minimal losses; and transparency does not take part in the decision when it comes to the necessity of backing up data, although it may ease up configuration and maintenance.

Throughout the years, it became clear that the best way to avoid data loss due to either common mistakes or catastrophic failures, while providing as much transparency as possible, would be to embed the backup tools into the file system. Currently, there are several file systems supporting snapshotting and file versioning, in several forms and shapes, but all of them with one basic idea: guaranteeing minimal to no information loss, and as much transparency as possible. In the next sections we present the evolution of file systems, hoping to establish a clear relation between the demands of the World facing data handling, and the responses regarding file system implementation.

2.3.1 1960's – 1990's

The first spoken occurrence of a versioned file system comes with ITS [Wik], an operating system from MIT dating back to the early 1960's. Around 1969, the Digital Equipment Corporation (DEC) developed the TENEX [BBMT72] operating system, later becoming the TOPS-20, also having an early versioned file system implementation. Accordingly to the cited paper, in TENEX, *"each time a file is written a new version is automatically created by making its version number be one greater than the highest existing version"*. Although, it wouldn't be until the 1980's that versioned file systems got their spotlight in the academic community. Circa 1983, the Cedar File System was introduced [GNS88], relying on NFS to provide access to remote, *immutable* (read-only) versions of files, which guarantees cache consistency between clients at reading time. Cedar keeps versions side-by-side with the files, using a specific name convention to differentiate among them. This approach to version-keeping was first seen in ITS, and would still be used in other implementations, following Cedar.

2.3.2 1990's – 2000's

In 1992, NetApp (formerly Network Appliance) developed the Write Anywhere File Layout (WAFL) [HLM94], which provides files to multiple clients while using multiple file systems. Among WAFL's capabilities, we are interested in its snapshot capabilities, backed by *Copy-On-Write* policies. Basically, WAFL creates snapshots on a *near per-write* basis, and keeps previous states of the system using snapshots. As we mentioned before, in Section 2.2.3, using *Copy-On-Write* it is possible to create snapshots with low, to no impact on the system's performance. In the late 1990's, the Elephant File System was introduced to the community [SFH+99], bearing much similarity to Cedar when it comes to version handling — versioning was managed by the file system, and version numbers were appended to the filename. By using *Copy-On-Write* protected blocks, the Elephant File System is able to keep a log of changes to inodes, while keeping earlier versions. This allows higher concurrency, while guaranteeing that each application keeps seeing the same file version it would see if there were no other updates to the inode.

2.3.3 2000's – present

The last decade brought an increased availability of implementations on this subject: in 2005, Zachary Peterson and Randal Burns proposed the *ext3COW* file system [PB05], based on a vanilla ext3 but with additional *Copy-On-Write* capabilities; also circa 2005, Sun Microsystems introduced *ZFS* [BAH+], considered by many the most fantastic file system ever developed; and circa 2008, the *Btrfs* [Mai] project was started by Oracle. These three projects had one thing is common: they provided block protection using *Copy-On-Write* policies, therefore allowing costless snapshots. However, ext3COW, at its time of release, only would create snapshots if explicitly requested by the user. Currently, this project is dead and, being a purely academical, its creators show no interest

on resuscitating it.

Both ZFS and Btrfs have different objectives than ext3COW. These are full-fledged file systems, and they aim at being used in High-Performance systems, both allowing the creation of multi-device spanning file systems, disk pooling, RAID or even supporting online file system resize, using file system-specific administration tools. The level of abstraction created simplifies the administration and the file system's maintenance. In reality, Btrfs comes as an alternative to ZFS for Linux-based systems, as until recently there was no implementation of ZFS in the Linux platform [Ric, Inf].

As stated before, both these file systems provide snapshot capabilities, although taking different approaches. On ZFS, snapshots are implicit, and a snapshot is created every time the tree root is atomically updated. Theoretically, this would happen on every write, but, to optimize the file system, the tree is only updated after applying batches of writes. Btrfs follows a similar approach on updating the tree, but does not implicitly create snapshots: if the user wants a snapshot, it must explicitly create it — even so, snapshot creation has barely any cost.

2.3.4 Summary

Table 2.1 presents some of the discussed file systems, some of which we consider to be the most relevant in our work's context.

We include both UFS and ext3 as they have been used as the file systems of choice in several platforms for some time now. The Unix File System (UFS) was developed with the intent of replacing the then used System V FS in UNIX, which had several issues leading to bad performance. UFS is still used in many UNIX-based operating systems, such as Solaris (which is the native operating system of ZFS). The Third Extended File System (ext3) is commonly used in Linux operating systems, still being used as the default file system by many Linux distributions. The stability provided by these file systems, and their widespread use, make them a great base of comparison for the remaining file systems presented in the table.

Although some interest in versioning schemes have been around for a long time, as it was established before back in this section (§ 2.3.1), only with WAFL, in 1992, we have the first full-fledged versioned file system beyond the academic circle. From this point on, several versioned file systems were introduced, and *Copy-On-Write* became widely used, in detriment of journaling policies. WAFL, ZFS and Btrfs are, among the presented file systems, those created for high-performance systems, thus their capabilities on resizing the file system while being used (Online Resizing), and all the three come with versioning capabilities. Therefore, we believe it becomes obvious that versioned file systems have been on their way to adoption on non-academic circles for a while, and may very well become a de-facto standard (considering what is expected from both ZFS and Btrfs) in scenarios where both availability and scalability are required.

Year	File System	Versioning	Consistency	Online Resizing	Academic
1983	Cedar	Yes	On-read consistency	?	Yes
1984	UFS	No	Journaling	No	No
1992	WAFL	Snapshots	Yes, using COW	Yes	No
1999	Elephant	Yes	Yes, using COW	?	Yes
2001	ext3	No	Journaling	No	No
2005	ext3COW	Snapshots	Yes, using COW; also using Journaling	No	Yes
2005	ZFS	Snapshots	Yes, using COW	Yes	No
2008	Btrfs	Snapshots	Yes, using COW	Yes	No

Table 2.1: Comparison of different file systems' properties.

2.4 Transactional Semantics

Several decades ago, the file system was the container of choice for rather large bulks of commercial information. An enterprise that required its customer's data to be stored, would store it in one or more files. If they also needed to keep their orders, payments, or debts, then other files would be created. Managing this amount of information was delegated to programs, that would access the required files and create, delete or update whatever data was needed. Such scheme would rapidly become unsustainable nowadays, since keeping this kind of information on a general-purpose file system has several disadvantages:

Data Redundancy and Inconsistency One may have the same information replicated in different files — e.g., the same address for the same costumer may be stored in both his orders' file and his debts' file —, or different files could hold different informations related to the same subject — the orders' file keeps a given phone number for the customer, while the debts' file keeps a different number.

Difficult Access Considering that, in these systems, informations are accessed by existing computer programs, if one wants to interpret the available information in a new and revolutionary way, one has to write a program to access it. However, as different files may be in different formats, one will have to take that into account and the complexity of accessing the information sky-rockets. It may be possible, but it isn't trivial.

Isolation and Concurrent Accesses Unless the correct measures are employed, file systems allow concurrent accesses to files. A program performing update operations may, in a first run, read some data in order to update it; however, upon update

the previously read data is no longer correct because some other program already changed it. Insuring that each program executes isolated from the rest of the system is not an easy task, and it increases the programming complexity — and overall propensity for mistakes.

Atomicity We have discussed, in Section 2.2.3 how fallible computer systems are and the kind of inconsistencies they may create if they crash. When changing crucial data, such as banking statements for instance, one has to make sure either all the changes are made or none are, and using general-purpose file systems, accomplishing this may be quite complex.

This reasons lead to the development of database systems, able to efficiently retrieve data while easily allowing the creation of relationships between complex data structures. With the formulation of the *Relational Model*, proposed in 1969 by E. F. Codd [Cod09], and the subsequent advent of Relational Databases and query languages, these became the method of choice to keep large amounts of related data.

Aside from the above mentioned capabilities, one of the main reasons for using databases is the simplicity of creating concurrent programs that access a vast quantity of data, without needing to care about implementing concurrency mechanisms, such as locks or monitors. The programmer can easily aggregate operations in *Transactions*, while the database itself will deal with the concurrency, guaranteeing everything works correctly.

2.4.1 Transactions

A Transaction is a logical unit of execution in a database. Much like the transactions discussed in Section 2.2.3, a database transaction is a set of operations that are made on the database's data, both read and write, while guaranteeing that either all operations are executed or none are, making sure the transaction transforms the database from one consistent state to another consistent state, whether the transaction fails or succeeds.

Once created, a transaction may be in one of five states (see Figure 2.4): *Active*, while the transaction is executing; *Partially Committed*, after it executes the last operation, but before it may actually be considered as successfully finished; *Committed*, as soon as all its changes are successfully written on disk; *Failed*, if any problem was raised while executing the transaction, such as conflicts between transactions; and *Aborted*, once all the changes made by the transaction have been rolled back and the database is restored to its previous state.

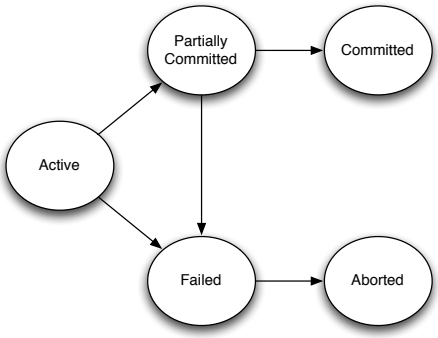


Figure 2.4: A Transaction's State Diagram (taken from [SKS06])

Database transactions are bounded by a set of four properties, known as *ACID* — *Atomicity*, *Consistency*, *Isolation* and *Durability*:

Atomicity Either all operations belonging to the transaction are applied on the database, or none are. If they are successfully executed and applied in the database, from the user's perspective, it appears as if they were atomically executed. If any of the transaction's constituent operations fails, then the transaction does not succeed and the transaction leaves no traces from its execution on the database.

Consistency The database is always kept consistent. Whenever a transaction commits, the database state switches from one consistent state to another consistent state.

Isolation From the transaction's point-of-view, it appears as if it is executing alone in the system. The behavior of a transaction does not affect another transaction's execution, regardless of the number of transactions executing concurrently.

Durability Once a transaction commits, its results are permanent and available to all subsequent transactions, even if there are system failures.

All the above properties have their own objectives and compliance with each one of them guarantees transactional semantics. However, there is a co-dependance between them. For instance, if a transaction is unable to fully comply with the Atomicity property, it may cause an inconsistent database state. The same is true for the Isolation property. Even if the Atomicity property is ensured for each transaction executing concurrently, their operations may interleave in an undesirable way, and if the Isolation property isn't fully complied with, then we may incur in an inconsistent state. A method to enforce Isolation, avoiding concurrency-related problems, is by forcing full transaction serialization; i.e., transactions are executed one after the other. This approach, though guaranteeing full Isolation for each transaction, diminishes concurrency and may cause performance degradation on the overall system.

Although it may be easier to run transactions serially, there are several benefits from allowing them to run concurrently, somehow similar to the benefits of allowing concurrent programs to execute in an operating system, such as higher resource utilization and reduced waiting time. For instance, if we execute transactions serially, the system is able to schedule a transaction while another is blocked waiting for some disk I/O, thus increasing CPU utilization and the system's throughput. Also, allowing concurrency between transactions, smaller transactions do not have to wait for a longer transaction to finish in order to execute. We will not get into specifics regarding the various isolation levels contemplated by the ANSI SQL standard [Ame92], and will rather focus on allowing concurrency while guaranteeing transaction serialization — i.e., the result of executing transactions concurrently is the same as if they were executed one after another.

A common method to achieve transaction serializability is by employing *Multi-Version Concurrency Control (MVCC)* mechanisms. Using MVCC, when a transaction T_1 commits and writes its data, the database will not overwrite the existing values; instead, a new

version of the values will be created. This approach will guarantee that a transaction T_2, that began before T_1 committed, will not have to abort if it reads a value changed by T_1, because the database will make sure T_2 reads the previous version of the data. In this case, T_2 may be serialized before T_1, as if it happened before T_1 committed thus ensuring serializability.

Snapshot Isolation [BBG+95] is a type of Multi-Version Concurrency Control. By executing each transaction on a snapshot of the database, taken at the start of the transaction, it is possible to ensure full concurrency with other transactions. As the transaction will be run on its own *copy* of the database, it runs in full Isolation mode, and all the reads it makes will be fully consistent with the time the transaction began. However, the transaction is not immune to conflicts with other transactions. The transaction will successfully commit if, and only if, the values changed by the transaction on the snapshot have not been externally changed on the database (i.e., committed by another transaction). If so, the transaction must be aborted. With Snapshot Isolation, read-only transactions never abort.

2.4.2 Bringing Transactional Semantics to the File System

Following the dissidence of high volumes of data to database systems, programmers started lusting the amazing guarantees these systems provided and how easy it became to create concurrent applications that would access the data. Concurrency control mechanisms are tricky to implement correctly, susceptible to misbehaviors such as deadlocks, and may considerably limit concurrency in an application. In databases, on the other hand, the programmer does not have to implement concurrency control mechanisms, delegating that task to the database. Nevertheless, in certain situations, using a database is overkill. In order to provide such guarantees, databases actually impose some overhead, although it is frequently ignored because the benefits are frequently much greater than the performance loss. Take an application that only requires keeping some configuration properties for a set of system programs, each of them scattered throughout the file system in different files. If one wants to guarantee that either all configuration changes are updated, or none are, guaranteeing that the configuration as a whole is kept consistent at all times, one may automatically ponder using a database for the task. However, in this case, using a database may not be viable: the system programs may be of different nature, one may not be able to change how they read their configurations, and the simplicity of the task at hand may not justify imposing the usage of a full-fledge database system. Using a simpler database system, such as SQLite [Hip] or the Berkeley Database (BDB) [OBM99], one would avoid the overhead of more complex database systems, but as both these implementations are software libraries, applications must be crafted to use them.

Soon started to became clear that transactional semantics could be useful in other

contexts besides databases. Back in 1977, D.B. Lomet [Lom77] proposed a programming mechanism to ensure the consistency of data shared between several processes, by using a construct to ensure atomicity in the system, analogous to the atomic property of database transactions. Apparently, the idea was forgotten for a couple of decades, probably because Lomet's paper explicitly disclaimed providing an implementation. Although, in 1993, Maurice Herlihy proposed the *Transactional Memory* model, which allowed for lock-free mechanisms with the efficiency of traditional lock-based mechanisms, guaranteeing some of the database transactions semantics. However, this proposal required specific hardware to fulfill the task. Not long after, in 1995, Nir Shavit and Dan Touitou published the paper that coined the concept of *Software Transactional Memory* [ST95] (STM), describing the first implementation of Transactional Memory, which provided an abstraction to access memory locations without requiring blocking mechanisms to ensure correctness, but incurring in significant memory overhead. Since then, there has been a growing interest in the community to develop non-intrusive methods to bring transactional semantics from the databases into general programming contexts [DSS06, Cun07], borrowing the ACI semantics, dropping Durability as it is not applicable in volatile memory. Furthermore, database mechanisms have been a source of inspiration for implementation decisions on other aspects of Software Transactional Memory, such as concurrency control methods based on multi-versioning schemes [Cac05].

Although STM makes the programmer's life easier by handling the required concurrency-control mechanisms, it isn't free of problems. One of them is the inherent incorrectness when using I/O primitives. The first property in ACID — Atomicity — relies on guaranteeing that if a transaction aborts its effects are never visible. However, using I/O inside a transaction is tricky because, generally, I/O is permanent. One may try to delay the I/O operations until the transaction commits, but there are situations in which that behavior is not acceptable (e.g., interactive application requiring user input) or may lead to unexpected states (e.g., one may eternally wait to read or write to a socket or IPC channel). It is possible, however, to implement file system I/O using a STM framework, and there are works proving just that, such as Artur Martins' *Transactional File System* (TFS) [Mar08], which relies on the Consistent Transactional Layer [Cun07] to provide file system transactions. This approach, relying on linking the target application with a STM library, although proving the possibility of applying transactional semantics on the file system, lacks in performance and transparency to applications.

Bringing transactional semantics to the file system has been a focus of great interest in the community, which is reflected by several works in the area in the last few years. For instance, back in 1993, Michael Olson developed the *Inversion File System* [OA93], which would be part of the Postgres DBMS up until 1995. This file system relied on a Postgres database, using it to keep both data and metadata, providing transactional semantics and fine-grained versioning. However, programmers would have to link their applications with a special-purpose library in order to use the file system and harvest

its capabilities. Later on, circa 2002, the Database File System [MTV02] (DBFS) was presented, proposing to improve what had been accomplished with Inversion, by not only offering a special-purpose library providing an interface to the programmer, but also allowing to transparently mount the file system over NFS. Using the Berkeley Database as backend, the authors of the DBFS concluded that programming complexity could be reduced by delegating the concurrency control to the database, while providing transactional semantics (specially regarding atomically changing the file system). In the end, DBFS demonstrated to perform worse than a native file system, but not as bad as to immediately discard the idea of effectively bringing transactions to the file system.

Subsequent implementations of transactional file systems include Nuno Luís' *Transactional File System Over Fuse* [Luí09] (c. 2008), which is inspired in the TFS but aiming at implementing it using Linux's File System in User Space (FUSE) [FUS], therefore providing full abstraction to applications while supporting implicit transactions (i.e., every I/O operation is regarded as belonging to a transaction), thus avoiding the need to rewrite an application to support transactions.

A different approach has been taken by the creators of the *TxOS* [PHR+09], a modified Linux 2.6.22 kernel, providing serializable system transactions with strong isolation. The authors modified 150 system calls, most of them totally supporting transactional semantics, which totals about half of the grand total of existing system calls in the modified version of the kernel.

In TxOS, system transactions may be considered as an abstraction allowing the programmer to update several system resources with ACID guarantees (enforced by the operating system). With the transactional semantics guarantees provided by system transactions, one is able to solve several issues in current operating systems whose solutions, although not impossible, may require some complex programming. For instance, the ability of running operations in full isolation will avoid time-of-check-to-time-of-use vulnerabilities, and also provides the mechanisms to easily roll back failed software installs. In addition, if system calls have transactional semantics, one will be able to safely use them inside transactions, avoiding traditional issues regarding I/O — although, communication channels' system calls don't support transactional semantics.

3

Btrfs — A New Generation File System for Linux

3.1 Introduction

Our work may be seen as yet another attempt at bringing transactional semantics to the file system. As discussed in Section 2.4.2, implementations range from the application-level to the operating system itself. Most of these approaches are purely academical, and they either don't perform reasonably well to be used in real-life applications, or they are too pessimistic and limit concurrency. We expect to contradict this tendency by extending a full-fledged file system, used in high-performance applications, and guaranteeing transactional semantics without compromising concurrency.

3.2 Introduction to Btrfs

Btrfs is a Linux file system from Oracle, aimed at being the next-generation file system for Linux systems and to suppress the gap between Linux and other operating systems with high-availability, high-performance file systems (such as Solaris with ZFS). Among its objectives, Btrfs introduces disk pooling, integral multi-device spanning file systems, checksumming and snapshots, as these are some of the required features that will allow Linux to easily scale into larger storage configurations as demanded nowadays. Furthermore, similarly to ZFS, Btrfs intends to deliver these features with a clean management and administration interface, reducing the burden imposed on the user.

Internally, all file system metadata is kept in modified B-Trees [Rod08], similar to B^+-Trees, hence the name *B-tree file system* from which *Btrfs* derives[1]. Any tree in Btrfs may store various data types, by using generic items identified by 136-bit keys. These keys are internally read as triplets of concatenated values in the form [`ObjectID`, `Type`, `Offset`], providing insight to the item `Type` they represent (predefined 8-bit value), and to which entity the item belongs to (64-bit `ObjectID`); the `Offset` is a 64-bit value which is type specific and may represent many different things. This key format allows each tree to hold multiple item types for each `ObjectID`, while keeping them close to each other. Performance-wise, this becomes very useful since the file system may leverage the contiguous nature of related keys and items within tree leaves.

In Figure 3.1 we depict the relation between keys and items within a Btrfs tree. Considering that a leaf is mapped onto a disk block (by default with a size of 4 KB), in the block's beginning we find the leaf's header, which describes (among other things) the amount of items within the leaf. However, items have a variable size, which makes them hard to correctly obtain and manipulate. Therefore, Btrfs puts the items in the end of the leaf, and for each one keeps a fixed-size *Tree Item* in the beginning. These tree items are responsible for providing the offset (within the leaf) where the item resides, while being tightly kept with a key, which uniquely identifies (and describes) the item.

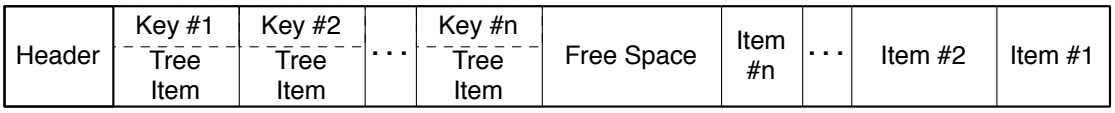


Figure 3.1: Depiction of a Tree leaf

This approach, however, suffers from internal fragmentation. Both the tree items and the keys have a fixed size, but they may refer to any kind of item, which are represented by different data structures with different sizes. In the end, the *Free Space* in the leaf may not be enough to allocate another pair (key, tree item) and its respective item, resulting in wasted space. Nonetheless, this side-effect is an acceptable trade-off to achieve performance over storage efficiency.

3.3 Btrfs's Trees

By default, and at any time since the file system is created, Btrfs has seven core trees, in which all information required by the file system is kept compartmentalized according to their specific purpose, as described in Table 3.1. However, additional trees, corresponding to Subvolumes and Snapshots, may be dynamically created using a user-level file system administration tool.

In the next sections we shall provide further details on three of the trees of Table 3.1 — *Roots Tree*, *Extents Tree* and *FS Tree* — as well as on subvolumes and snapshots, since our

[1] Alternate meanings such as *Better FS* are used.

Name	Description	Stored Information
Roots Tree	Keeps pointers to the remaining trees in the file system	Metadata
Extents Tree	Keeps pointers to the currently allocated extents	Data and Metadata
Chunk Tree	Upon dividing the disk into chunks, keeps track of their information	FS Data
Devices Tree	Each physical disk belonging to this file system will have an entry in here	FS Data
File System's Tree	Default file system tree	Data and Metadata
Checksum Tree	Keeps the checksums for all existing extents	Metadata
Log Tree	Tracks the pending operations to be applied to disk	Data and Metadata

Table 3.1: Btrfs's Trees and their purpose

work heavily depends on both these two last concepts.

3.3.1 Roots Tree

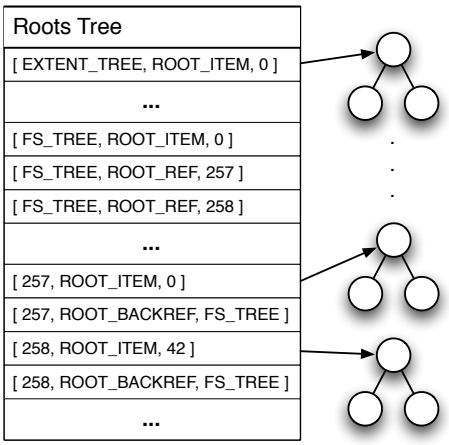


Figure 3.2: Roots Tree directory-like behavior

The *Roots Tree* comes as a solution to the dynamic nature of Btrfs's subvolumes and snapshots. This tree is little more than a directory of tree roots, making the *Roots Tree* also known as the *Tree of Tree Roots*, associating names to locations. Each tree in the file system is assigned an unique identifier (an `ObjectID`), and is mapped in the *Roots Tree* with three different entry types: `ROOT_REF`, which associates two trees in the sense that one can be reached (within the file system) through the other; `ROOT_BACKREF`, referencing a tree's name; and `ROOT_ITEM`, which pinpoints a tree's on-disk location. Depending on the tree to which the item belongs, so depends the key's `ObjectID`. For instance, all the trees described in Table 3.1 have predefined identifiers, known to the file system; on the other hand, any Subvolume or Snapshot will be assigned sequential identifiers with values greater than 256.

Up until now, we have spoken of subvolumes and snapshots as two related concepts, albeit distinct; however, from Btrfs's point-of-view, they are just the same. In the *Roots Tree*, both subvolumes and snapshots are simply considered *Roots* and are mapped with the same item types as other trees (i.e., items of type `ROOT_REF`, `ROOT_ITEM` and

ROOT_BACKREF).

In Figure 3.2 we illustrate how the *Roots Tree* is populated by keys. For instance, take those with type ROOT_ITEM. As we just mentioned, these will keep the on-disk location of the tree they refer to. The key's first value is the ObjectID of the tree the key refers to, whether it is a core file system tree (EXTENT_TREE or FS_TREE, in this example), or a subvolume or snapshot tree (ObjectID's equal to 257 and 258, respectively). From the file system's point-of-view, there is no difference whatsoever between a snapshot and subvolume; however, in the *Roots Tree* we are able to distinguish them through the key's Offset field: when its value is zero, then it is a subvolume; otherwise, it is a snapshot.

Regarding the items of type ROOT_REF, these are used to easily determine where to look for the trees in case of path based operations. For instance, the key [FS_TREE, ROOT_REF, 257] indicates that a reference to the tree with ObjectID = 257 may be found within the *FS Tree*. Analogously, the items of type ROOT_BACKREF indicate within which tree is a given root contained — the key [257, ROOT_BACKREF, FS_TREE] states that the root with ObjectID = 257 may be found within the *FS Tree*.

3.3.2 Extents Tree

Back in Section 2.2.2 we explained that an *Extent* is a group of contiguous blocks, holding related data, with the purpose of leveraging its proximity in order to reduce disk seeks. In Btrfs, extents are used with several purposes, let it either be holding tree nodes/leaves, or file's data. In fact, the file system aggregates extents into distinct *block groups*, which are defined as either holding data — such as file contents — or metadata — such as trees and their nodes/leaves. By taking advantage of these block groups, a file is able to maintain the on-disk locality of its contents.

The *Extents Tree* is thus responsible for mapping the existing extents in a straightforward manner, tracking the space usage, along with the available extents, in order to facilitate the allocation of new extents or their removal. Each entry in the *Extents Tree* will keep, among other information, the on-disk location of the extent it represents, along with its length and a reference count. This reference count is essential when taking advantage of the *Copy-on-Write* nature of snapshots, since it makes it possible to share extents between different file system trees. This capability to share the same extents, among multiple distinct file system trees, is quite useful for our work, as shall be discussed later on Section 4.6.

3.3.3 File System Tree

The *FS Tree* is where the files and directories live by default in a Btrfs file system. This tree will hold items referring to inodes, directories and their names and associated metadata, ordered in a fashion similar to that described when we discussed the *Roots Tree*.

Within the *FS Tree*, inodes are mapped using INODE_ITEMs, which hold inode-specific informations, such as the inode's mode and link count. The inode's data may also be kept

embedded in the item if its size so justifies it. However, if the inode has larger amounts of data, one or more EXTENT_DATA items will exist nearby the INODE_ITEM, pointing to the extent where the data lies. Optional metadata, such as Access Control Lists (ACLs), will be held in specific items, also leveraging the locality principle of Btrfs' trees. Additionally, each inode has one (or more) INODE_REF items, which associate a name to the inode, while allowing upward traversal of the file system hierarchy, since it keeps a reference to the directory holding the name (i.e., its parent directory) — in case of multiple hard links, there will be multiple INODE_REFs.

Each INODE_ITEM may refer to either a file or a directory. In case of a directory, the *FS Tree* will also contain a DIR_ITEM and a DIR_INDEX item for each entry in that directory. Both these items will hold the location of the INODE_ITEM associated with the entry, as well as the entry's name. The difference between these two items is that the DIR_ITEM will have its key's Offset set with the entry's name hash, allowing for faster path-based operations, while the DIR_INDEX will have the entry's index within the directory, with the purpose of providing predictable directory reads. Additionally, both the DIR_ITEM and the DIR_INDEX may refer to a ROOT_ITEM instead of an INODE_ITEM. This is the verified case when a subvolume or a snapshot is created within the tree's hierarchy, and serves as an indication to the file system that further lookups should be made to the root referred by *location* field in both these two items, as illustrated in Figure 3.3.

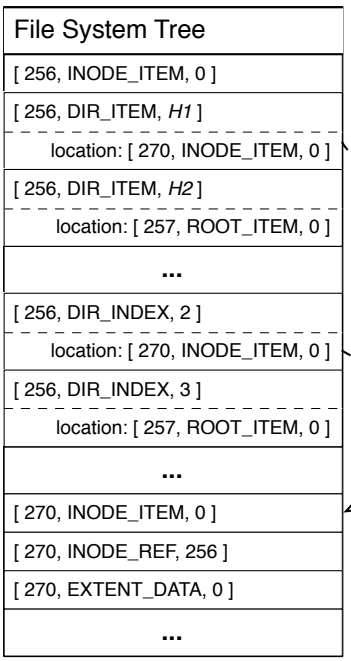


Figure 3.3: Contents of the File System Tree

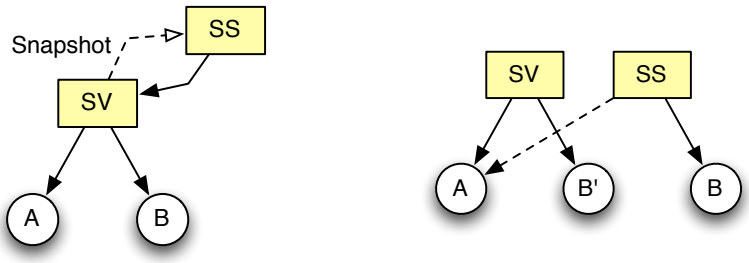
Regarding this figure, taking the first key in the tree, notice how it represents an item of type INODE_ITEM, which lets us assume it refers to a directory, with an inode value of 256 (the ObjectID holds the inode's value). In this case, the inode also corresponds to the tree's root directory — each file tree, let it either be the *FS Tree* or any subvolume or snapshot tree, defines its root as having the inode 256; this inode is used to represent the *parent* of all that is added onto the first level of the tree's hierarchy.

This leads us to the next keys in the tree, of type DIR_ITEM. Both have their ObjectID = 256, meaning they still refer to the INODE_ITEM just described; in a nutshell, both these items describe directory entries within the *FS Tree*'s root directory. In this case, one of the directory items refers to a file belonging to the *FS Tree*, while the other refers to a directory used as an entry point into another tree. We have omitted the file's names in both items since they are not relevant for the task at hand.

Following the INODE_ITEM with ObjectID = 270, we find ourselves in the presence of the inode's metadata. First, we encounter an INODE_REF, with an Offset = 256, meaning there is a name for this inode held by a directory with inode value 256 (which we have just seen is the tree's root directory). Additionally, we find an EXTENT_DATA item, which

will pinpoint the location of the inode's data.

3.3.4 Subvolumes and Snapshots



(a) Creation of snapshot `SS` based on subvolume `SV`

(b) Effect of changing element `B` on `SV`

Figure 3.4: Depiction on how Btrfs' Snapshots and Subvolumes work.

In Btrfs, Subvolumes and Snapshots are full fledged *File Trees*, just like the *FS Tree*: they have on-disk trees, hold files and directories and have an independent inode allocation policy, which basically means that between trees we may have different files with the same inode value, just as if they were different volumes of the same file system.

However, a subvolume or a snapshot are nothing but a user-level abstraction. Technically, from the file system's point-of-view, they are just about the same. The only difference is that a subvolume is created empty, while snapshots are logical copies of subvolumes. This means that we may create a subvolume at any time, for any purpose we intend, but a snapshot must be taken from an existing file system tree, which typically means taking it from a subvolume, although it may also be taken from the *FS Tree*.

Being a logical copy, from the user's perspective, the snapshot contains everything that the subvolume contained when the snapshot was taken; for the file system this is not quite so. In fact, when the snapshot is taken, the only thing that is actually created is its tree's root, which will point to the same extent where the subvolume's root node is kept. Using a policy of *Copy-on-Write*, Btrfs is able to share unmodified data (both tree metadata and file data); as soon as a write on one of the trees targets a shared extent, the said extent will be copied and will become local to the tree through which it was modified, as illustrated by Figure 3.4, where we represent both the creation of a snapshot from a subvolume, as well as the effects of changing a file (initially shared) in the snapshot.

By supporting subvolumes and snapshots, and by intermixing these with the file system's default file tree (*FS Tree*), Btrfs allows one to create a hierarchy based on the one's needs, by opening the door to hierarchies where subvolumes and snapshots coexist with common files and directories. As an example, take a file system hierarchy commonly used in *nix systems, where the users' home directories are kept in `/home`'s and system resources in `/usr`. If we used Btrfs as the system's root file system, and being the *FS Tree* the volume's default tree responsible for keeping files and directories, we would assume

that from the / downwards, all files and directories would be created within the *FS Tree*. Nonetheless, we could also create two additional subvolumes in Btrfs: one for `/usr` and another for the skeleton home directory (say, `/etc/skel`).

Such an hierarchy would be useful for two different reasons: first, being the skeleton home directory a subvolume, upon creating new users we could simply snapshot `/etc/skel`, which would be useful (storage-wise) if the users shared a lot of resources on their home directories but didn't actually change them much; secondly, by using a subvolume as the `/usr` directory, we could snapshot it each time before upgrading some software, and revert back to the snapshot if needed — Red Hat's Fedora 13 has this kind of support in the *yum* package management utility [Fed].

Furthermore, Btrfs allows both subvolumes and snapshots to be mounted directly, which becomes useful if one does not want Btrfs's default file system tree (*FS Tree*) as the system's root, but does want to mount specific subvolumes and snapshots throughout the file system hierarchy.

3.4 Summary

Btrfs has a large code base compared to other popular linux file systems. Out of curiosity, Btrfs has four times the (real) lines of code that ext3 has, and twice as much as ext4, rounding a grand total of roughly 46 thousand lines. What we have presented during this chapter is roughly equivalent to a fourth of the whole file system code. It is then easy to conclude that we focused on a limited part of the file system.

All the concepts, namely keys and items, in Section 3.2, and the trees presented in Section 3.3, were lightly explored. There is much more to them, but their details are outside the scope of this dissertation. Our objective with this chapter was not to give a crash-course on Btrfs internals, but to lay the path for Chapter 4, in which we shall describe our work towards a transactional file system.

Despite the fact that we resorted to all these concepts, the truth is that we barely handled them directly. Aside from items and keys, we almost were not required to directly handle any of them, thanks to the large set of methods, available in Btrfs, to manipulate the most varied data structures, mainly trees. Therefore, we focused on our work, and were able to keep our implementation apart from Btrfs's as much as possible, even avoiding to touch Btrfs's code unless it was otherwise imperative.

4

TxBtrfs — A Transactional File System Based on Btrfs

4.1 Introduction

By taking advantage of Btrfs' snapshot capabilities, we expected to provide transactional semantics in the file system so applications could benefit from the isolation guaranteed by executing the operations on a private file system copy. However, this meant to associate each transaction with a private snapshot, and Btrfs does not support this behavior straight out of the box. Furthermore, given the nature of Btrfs and its support for multiple subvolumes, we believed that forcing the whole file system to be transactional would be stepping over our bounds and constraining the user beyond his needs.

These beliefs steered us into defining a new concept of subvolume: the *Transactional Subvolume*, or TxSv. A Transactional Subvolume is pretty much the same as a regular Btrfs subvolume, but marked as expecting all operations as being part of some transaction, so they may take advantage of the file system's transactional semantics. At any point in time, any given existing subvolume may be marked as a TxSv, although only one TxSv will be allowed at a time, either by using a Btrfs-specific user-level management tool, or by directly issuing an IOCTL system call.

Currently, TxBtrfs assumes all accesses should be transactional, although any non-transactional operations that are issued directly onto the transactional subvolume will always succeed. Nonetheless, any operations performed will be logged as well, and the next committing transaction will be forced to validate against those changes. However, full support for non-transactional accesses is still a subject under study, and currently

may lead to some unforeseen, catastrophic, consequences (such as file system corruption), and are highly discouraged.

4.2 TxBtrfs in a Nutshell

Creating a transactional file system raises several questions: what is a transaction from the file system point-of-view, when and how do they start and finish, and in what state is the file system left in after a transaction completes (either successfully or not).

From TxBtrfs' point-of-view, a transaction is composed by all operations between a *Start Transaction* and a *Commit Transaction*[1]. These operations are specific to TxBtrfs and take the form of IOCTL system calls, although they may be mapped onto a user-level library for simplicity's sake. When defining how the file system should behave regarding transaction creation, we pondered supporting only implicit transactions, starting a new transaction on `open(2)` and committing it on `close(2)`. However, several problems may arise from implicitly starting and committing transactions, mainly due to the nature of the POSIX standard, which assumes each operation to be independent and any erroneous behavior strictly affects the operation raising the error. One simple example is the case where a single operation, from a set of read/write operations issued between an `open` and a `close`, conflicts with another transaction. If the conflict is detected only at `close(2)` time, one would have then to inform the application of such conflict, and hope the programmer checked for errors on `close(2)`, an operation usually assumed to succeed.

In order to avoid problems raised by unsuspecting users, and to avoid clashing against the POSIX standard, we chose instead to define IOCTLs to be called from the user-level, which would in turn start and commit a transaction on-demand. During this document, these operations take the form of `TxStart` and `TxCommit`. Both will take an argument, *TxSv path*, which is the location of the Transactional Subvolume on which to run the transaction.

The following sections roughly describe the impact of the `TxStart` and `TxCommit` operations on the file system, as well as how applications and processes are associated with transactions. Later on in this chapter we shall dwell deeply into this subjects, giving further details on both their behavior and implementation.

4.2.1 Starting a Transaction

Issuing a `TxStart` operation may be seen as an admission of intention, by the application, to associate the issuing process to a transaction — transactions are associated with processes, not with the application itself.

Once a `TxStart` is invoked, TxBtrfs will create a copy of the TxSv and assign it to the process starting the transaction. This copy is merely logical, as it is in fact a snapshot of

[1] At the moment, TxBtrfs does not support an *Abort Transaction* operation (see Section 6.2).

the Transactional Subvolume. In reality, the snapshot may not even be created, as long as the process starting the transaction is already part of a transaction by itself, as we shall discuss later in Section 4.4, since TxBtrfs allows a certain degree of transaction nesting.

The application is oblivious to the snapshot creation phase, and even though its processes are mapped onto a snapshot of TxSv the application will still believe to be accessing the Transactional Subvolume directly (the mapping process shall be explained in further detail in Section 4.3). After the transaction is created, every operation issued during its course of execution targeting the Transactional Subvolume will be transparently mapped into the transaction's private snapshot, while being guaranteed that only processes participating in a given transaction will be able to modify that transaction's snapshot. This strict association between processes and transactions allows us to guarantee that each transaction runs fully isolated from other concurrent transactions.

4.2.2 Committing a Transaction

Committing a transaction is a three-phase process: Conflict detection, Reconciliation and updating the Transactional Subvolume's (TxSv) root.

Conflict detection is common in distributed and replicated file systems, such as the Coda file system, which detects conflicts when two replicas perform operations (on the file system) that may not be compatible [KS93], or even in file system synchronizers, such as Unison [BP98]. Our work, although being kept local to a single file system volume, may be seen as replicated for concurrent transactions: we do replicate the TxSv each time a transaction starts, and when the transaction ends we must guarantee its modifications do not clash with other operations that may have happened in the meantime. For this reason, when a process invokes a `TxCommit`, TxBtrfs will attempt to detect any potential conflict between the process's transaction and the current state of the TxSv by performing a symbolic replay of a log containing the transaction's operations. This symbolic replay happens over in-memory data structures only, since we do not actually require to access the disk in order to obtain the relevant informations to perform it, which will avoid imposing additional overhead on the system. The symbolic replay, which we will describe later in Section 4.5, focuses on creating an in-memory representation of the current file system hierarchy, against which will replay all the operations performed by the transaction. If any of the operations conflicts during the symbolic replay, then the transaction will be aborted; otherwise, TxBtrfs will assume the transaction to be valid regarding the current TxSv's state and may be freely applied.

Once a transaction is considered as being valid, we may then proceed with reconciling it with the TxSv (Section 4.6). In the context of this document, we call *Reconciling* to the process of merging two diverging states (the TxSv's and the transaction's). During this phase, all updates made to the TxSv since the committing transaction began will pulled into the transaction's snapshot. Once this process is finished we may consider that the snapshot is in a state consistent with all the previously committed transactions.

We may then proceed to next and final stage, *Committing the Snapshot*, which may be seen as changing a pointer from the TxSv's root and pointing it to the snapshot's root, making the transaction's snapshot the new Transactional Subvolume (also explored in Section 4.6).

4.3 Processes & Transactions

TxBtrfs main goal when it comes to transactional accesses to the Transactional Subvolume (TxSv) is to map those accesses in a snapshot associated with the process' transaction, as transparently as possible. This means creating an abstraction so, by accessing the TxSv, the process unknowingly accesses the transaction's snapshot.

When we started our work on TxBtrfs, we assumed that doing this mapping would be a question of changing a path on-`open`.However, we realized this would not work since the VFS does not work this way. In fact, path translation happens way before a file system specific `open` is issued, and by that time the whole path has already been looked-up and there is little we can to do.

This lookup process relies on the file system holding the file, and there is a method which is called once the VFS wishes to translate a component of the path. We thought about shifting the mapping to the file system's `lookup` method, returning inodes from the snapshot instead of inodes from the TxSv, but such approach wouldn't work as well. This results from the fact that the VFS lookup does not rely solely on the file system's `lookup` method, but also on the *dentry cache*, or *dcache*.

The dentry cache aims at improving overall system performance by reducing the required amount of lookups within the file system, and their more-than-likely disk accesses. Basically, once a dentry is looked up during path translation, it is put into the dcache for subsequent usage. As the dentry is kept in-memory, lookups will first look for the dentry in the cache, querying the file system only as a last resort. As each entry in the dcache is both path and name based, trying to abstract accesses by returning different inodes for the paths during lookup poses significant problems:

- Keeping the name of the transactional subvolume, changing only the inode it would point to, and because the dcache is an open hash table, we would end up with every single snapshot cached in the same bucket as the transactional subvolume. This would probably become a bottleneck and a cause for overall performance degradation.

- Even if the previous problem was not an actually a problem, we would still have an issue: comparing two different entries in the dcache is a string based operation (i.e., if string A is equal to string B, then entry A equals entry B). From the moment we put all the snapshot entries in the same bucket as the TxSv, all sharing the same name with the TxSv, then whichever was the first to be encountered in the list

would be the one returned, regardless the inode it would refer to. This would not do if we were to get our job done.

One of our initial objectives was to provide a shared entry point for all transactions, each accessing the TxSv and being internally mapped onto their snapshots, as if all the snapshots were mounted in the same place but each transaction could only access its own snapshot. However, due to the lookup method used by the VFS and to the inner-workings of the dcache, we decided that each snapshot should be allowed to have a specific name and be accessible by path, although only accessible by its own transaction and nobody else. This way we would be able to use the underlying filesystem capabilities regarding lookups, since we would be providing different names and paths to the dcache in order to represent different inodes.

4.3.1 Mapping Processes onto Snapshots

Upon path translation, the VFS jumps through a lot of hoops before getting to the filesystem's `lookup` method, and, as stated before, by that time it would not be feasible to change paths. However, prior to reaching the file system's `lookup`, the VFS calls a filesystem specific `hash` method, in case the filesystem wants to hash the entry in a different way. Although we contemplated the possibility to assign different hashes to different dentries based on the snapshot they belonged to, we would be betting a whole lot of work on the assumption that no hash collisions would ever, ever happen. Therefore, once the VFS reaches the hash method we simply change paths from the TxSv to the process' snapshot.

Once we are translating a path on TxBtrfs, we may encounter ourselves in one of four distinct scenarios:

i) The path refers to a common, non-transactional, subvolume or snapshot.

ii) The path indeed refers to the TxSv, but the process looking it up does not have an on-going transaction.

iii) The path refers to the TxSv and the process looking it up has an on-going transaction.

iv) The path refers to a dentry somewhere below the root of either the TxSv or a snapshot of the TxSv.

If the path being translated happens to be on the first scenario, from TxBtrfs' point-of-view it is irrelevant, and there is no action to take since Btrfs itself will take care of all non-transactional accesses. On the other hand, if it does belong to any of the remaining scenarios we have to handle it.

Take second scenario, in which the path refers to TxSv's root's dentry. Such access usually happens when a new transaction is being started, since the `TxStart` method requires the TxSv root as an argument, and thus it must be translated by the VFS. We

must not discard it, but we also must not map it to a snapshot as there is none where to map the access on. Therefore, we simply let it slide and no path is actually changed.

If the process does have a running transaction, it means we are on the third scenario and we must map the process onto its transaction's snapshot. The mapping process is quite simple, requiring us to determine to which transaction the process belongs to, and changing the requested path from that of the TxSv to the path of the transaction's snapshot. Determining if the process belongs to a transaction, and if so to which, is a matter of querying a tree within TxBtrfs' in-memory data structures, which uniquely maps Process IDs (PIDs) to transactions, which keeps a record to the snapshot the transaction is mapped to — if the process does not have an entry, then it does not belong to an on-going transaction.

Finally, the fourth scenario is partly conceptual. If accesses are being performed down TxSv's tree, we assume they must be part of a transaction. Such assumption relies on the fact that we do not guarantee file system consistency when non-transactional accesses to TxSv occur. If these accesses are in fact part of a transaction, then their lookups are not made on the TxSv tree, but on the transaction's snapshot. Such happens to be true because path translation is composed by multiple translations of each one of the path's components. Therefore, if we are to lookup a given component B in the path */A/B*, we know we have already successfully translated the component A. If A were to initially be the TxSv, and the process looking up is part of an on-going transaction, by the time we lookup B we already have A translated to the transaction's snapshot, meaning we have nothing else to do and we will let Btrfs handle the request without changing any more paths.

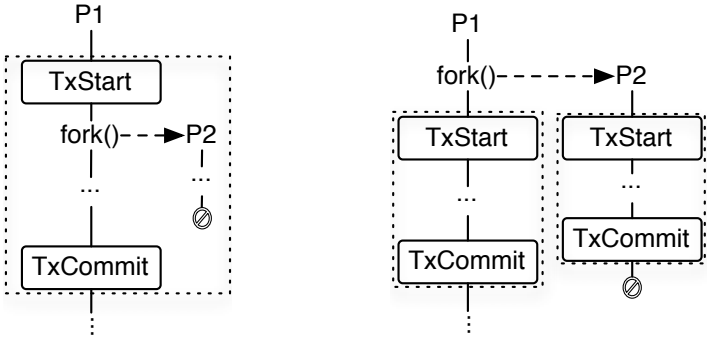
4.4 Transaction Hierarchy

As it has been established in past sections, whenever a transaction is started both the transaction and the process starting it are assigned to a private snapshot. For instance, take any two processes, P_1 and P_2, and let us assume that each one of them are independently executed and each of them starts a transaction. According to everything that has been discussed so far, this means that TxBtrfs will create two different transactions, mapping P_1 onto one of the transactions' snapshot, and P_2 onto the other transaction's snapshot.

Despite the fact that TxBtrfs enforces each process to be associated with one, and only one, transaction (and therefore a single snapshot), it is allowed that a single transaction be associated with many different processes. However, such situation is only possible in three distinct scenarios, all of them involving a process and its children, aiming to keep the expected resource sharing semantics.

In Figure 4.1, we present one of such scenarios. Take Figure 4.1a, in which P_1 starts a transaction and then issues a `fork(2)` leading to the creation of P_2. In this case, P_2 will inherit its parent's (i.e., P_1's) transaction, and all transactional accesses P_2 may issue on

the TxSv will be mapped onto P_1's transaction's snapshot until P_2 finishes, thus the child never conflicts with its parent. On the other hand, if P_1 issued a `fork(2)` before starting the transaction, then P_2 would not inherit P_1's transaction as the transaction would not exist at the time of P_2's creation. This behavior is illustrated in Figure 4.1b, and would require for P_2 to start its own transaction if it were to access TxSv, and now P_1 and P_2 may conflict.



(a) `fork(2)` issued during a transaction

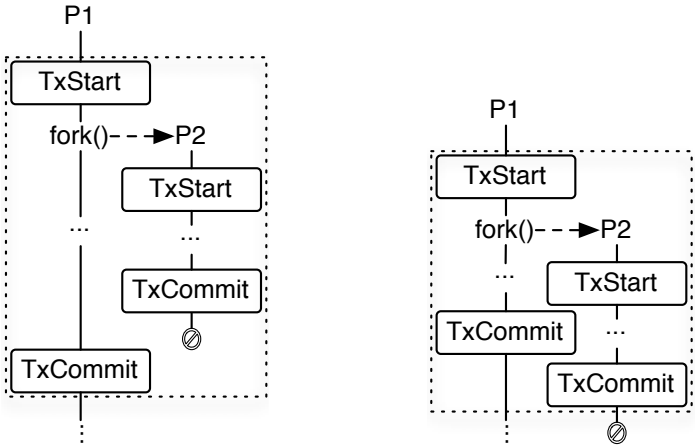
(b) Parallel execution of P_1's and P_2's transactions

Figure 4.1: `fork(2)` issued in different contexts

Let us focus on yet another scenario, similar to that of Figure 4.1a, in which P_1 starts a transaction and then forks P_2. However, in this case P_2 starts its own transaction while within P_1's transaction. In Figure 4.2a we illustrate P_2's transaction nested within P_1's. Since the child process shares its parent's transaction, we hereby establish that any transaction started by the child or any of its descendants is considered as *nested* following a *flat nesting* model [ALS06]. This means that any operation made during the nested transaction shall be considered as belonging to the outer-most transaction. As we can see, the scenario presented back in Figure 4.1a is a particular case of the scenario in Figure 4.2a, on which the child process does not start a transactions.

Finally, take Figure 4.2b, which presents the third and last scenario, in which the parent's transaction commits before the child commits its transaction. In this case we are faced with an interleaving that could result in serious problems if we allowed the parent to take the final word on when to commit the transaction, disregarding its child inner-transaction. Therefore, we established that only the last commit will actually commit the transaction. Using Figure 4.2b as an example, we have P_2 starting its inner-transaction during P_1's transaction, which may be seen as an increment on a counter within the transaction. While this counter is greater than zero, any issued commit will be void besides its decrement of the said counter; only when P_2's commit is issued, and the counter finally reaches zero, will the transaction be actually committed. wait for P_2 to complete. Once P_1's commit returns, even

Since TxBtrfs assumes every transactional access made by a child process being part of the parent's transaction (as long as the child has been created after the transaction start),



(a) Nested Transactions

(b) Interleaved Transactions

Figure 4.2: Different scopes for inner-transactions

and because we do support both transaction nesting and interleaving, we consider a bad approach all transactional accesses made by the child without being wrapped within a transaction. In Figure 4.3 we illustrate how a transactional access by a child process P_2 may become problematic. This case, similar to that of Figure 4.1a, begins with P_2 being created within P_1's transaction and performing any number of transactional accesses to TxSv during its execution. However, at some point before P_2 finishes, P_1 commits its transaction. From this moment on, any transactional access made by P_2 won't be part of P_1's transaction as it no longer exists. Such example sustains our premise that all transactional accesses should be wrapped in a transaction. If P_2 had started a transaction, then its behavior would have fallen into either one of the scenarios in Figure 4.2: either P_2's transaction would have been nested within P_1's, or it would have been interleaved with P_1's; regardless, both scenarios are correctly handled by TxBtrfs without compromising the expected behavior when accessing TxSv. However, if the developer does not wishes to create a transaction on P_2 for a particular reason — for instance, in order to perform a *fork-exec* for another application, and if that application is to perform accesses to TxSv — then the developer should guarantee that the parent process waits for completion on P_2 before committing the outer-most transaction.

4.5 Transaction Validation/Conflict Detection

When a transaction starts, it becomes isolated from the remaining file system, performing its operations on a private snapshot of TxSv. During its execution, other transactions may be started and committed, thus pushing forward the Transactional Subvolume's state. Upon commit, a transaction T_1 may find the TxSv in one of two states:

- The TxSv is still in the same state as the one T_1 inherited when it started, or

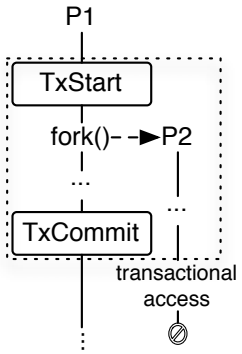


Figure 4.3: Erroneous transactional access by P_2

- At least one other transaction T_2 committed, and the TxSv is now in a different state than the one T_1 inherited when it started.

In case there were no changes to the TxSv, then T_1 is in perfectly good conditions to commit. However, if that is not true, the transaction must go through a validation phase, in order to check for potential conflicts between the transaction's state and the TxSv's new state.

To validate the correctness of a transaction's modifications to the file system, we must assess which changes made in the TxSv are worthy of validating the transaction against. With this purpose, when we start a transaction we assign it a *generation value*, which represents the last TxSv state known by the transaction. A transaction's generation is always equal to the generation of the TxSv the transaction started from. Take, for instance, two different transactions (T_1 and T_2) which have started from the same known state of TxSv with generation zero ($generation = 0$). Let us now assume that T_1 commits before T_2. In this case T_1 commits freely, simply because there were no other commits between T_1's start and its commit, or, as we also may put it, because TxSv's generation was still equal to T_1's generation.

Committing a transaction such as T_1 leads the TxSv onto a different state, which is represented by increasing its generation. Thus, when T_2 tries to commit, the state T_2 will find the TxSv in will not be the same as the one T_2's snapshot initially inherited (TxSv's generation $\neq$ T_2's snapshot generation). In this case, we must determine which modifications were made between the last TxSv's state known by T_2 (generation $= 0$), and the TxSv's current state (generation $= 1$).

Hereinafter, and before we proceed, to a transaction's last known state of TxSv we shall call the transaction's *Master Copy*, and to the set of modifications made on the Master Copy leading to the TxSv's current state we shall call ΔM. Lastly, to the set of modifications made by the transaction onto its private snapshot we shall call ΔS.

4.5.1 Operation Deltas

During a process execution, any number of operations may be performed over the file system, either creating, removing, reading or writing, anything is possible as long as supported by the VFS. While executing, if the process is associated with a transaction and it accesses the transaction's snapshot[2], we will log the performed operations so we may use them during the transaction's validation phase. These logged operations will be the transaction's ΔS, kept private to the transaction, and represent every action that made the transaction's snapshot to evolve from its initial state up to the state at commit time, along with any operations performed that may result in a semantical conflict with other transactions. For instance, a `read(2)` does not modify the snapshot, but does present a potential cause for a semantical conflict, and for such conflict to arise it only takes one transaction to write onto the same place that another transaction reads from.

Additionally there is the ΔM, which (as previously described) is the set of all modifications performed on the transaction's Master Copy since the transaction began until it is ready to commit. If we imagine a transaction that has committed onto the TxSv, we can extrapolate the ΔM that took the TxSv onto the transaction's state is the transactions ΔS, stripped of its non-modifying operations (i.e., reads), since these do not actually alter the file system. Generalizing, if we have a transaction T_1 with generation $gen = n$, and if the TxSv has generation $gen = m$, we may consider the ΔM as nothing more than the union of all the (stripped) ΔS of all transactions that modified (i.e., committed on) the TxSv since generation n until generation m:

$$\Delta M = \bigcup \Delta S_i \, , n < i \leq m$$

It is important to correctly understand these concepts in order to understand the validation and conflict detection process, which we shall describe in sections to follow.

4.5.2 Initial Validation/Conflict Detection Process

At this point we have established the existence of two different sets, ΔS and ΔM, which represent the operations performed in order to obtain two different states: with ΔS we obtain the current state of the transaction T trying to commit, while with ΔM we obtain the current state of TxSv from the point when T started up until the point when T tries to commit.

We are then in conditions to evaluate if T performed any operation, during the time it executed isolated from the remaining system, that may be in conflict with TxSv's current state. For instance, if T created a given file F_1 in TxSv's root directory, which was non-existent at the time T started, and by some chance another transaction (now committed) evolved the TxSv state up to a point where F_1 also exists in the TxSv's root, then we may

[2] As discussed in Section 4.3, although believing to be accessing TxSv, the process is mapped onto the transaction snapshot.

conclude that T performed a modification that conflicts with the current TxSv state.

It is thus necessary to assess if the transaction trying to commit may actually commit. Since we know both ΔS and ΔM, we may then compare the operations both these sets hold. If we find the slightest possibility of a conflict, then the transaction's commit should not proceed.

When we began our work on TxBtrfs, we naively assumed that by simply logging the operations and extrapolating both the ΔS and the ΔM we would be able to straightforwardly detect conflicts. Our approach at the time was to traverse ΔM and, for each operation in ΔM, traverse ΔS looking for any operation that could trigger a conflict. It didn't take long before we realized how little thought we had put into such an approach. The flaw with traversing both logs is not only the overhead imposed by an $O(m \times s)$ complexity[3], but also the inability to infer the relationship between operations. For instance, if ΔM contains a `create(`F_1`)`, followed by an `unlink(`F_1`)`, from an external observer point-of-view F_1 is as good as non-existent in the final file system state. However, with the approach we were taking, we would detect a conflict if ΔS contained a `create(`F_1`)`, when in fact there would be no reason for such conflict to ever arise, assuming the same parent directory, of course.

Even if we could gladly accept such false positives as a mean to obtain a correct (although highly pessimistic) validation, we believed this approach to be too constraining when it came to the next step of transaction commit: the reconciliation process. Although the validation and conflict detection process is independent from the reconciliation process (which we shall describe in detail later on) we have always believed there was no point in wasting the precious work done during the log traversals. However, since our initial approach kept no information whatsoever, we would have to traverse the logs once more just to reconcile the transaction's snapshot with the TxSv. All these issues led us to define a different method to validate the correctness of the transaction's operations and perform conflict detection, while still keeping valuable information for later use during the reconciliation process. We call this method *Symbolic Log Replaying*.

The way we see it, our *Symbolic Log Replaying* is much the same as any other log replaying technique, except for the fact that we do not actually apply any of the log elements being replayed onto the file system. Essentially, we take ΔM and ΔS and we create an in-memory representation of the final state of the file system — i.e., a glimpse of what the state would be if the transaction was successfully committed. This representation is obtained by replaying all the operations on both ΔM and ΔS, one by one, and applying their effects only on in-memory data structures, allowing us to detect conflicts more easily and still be able to (later on) use the information for the reconciliation process, as shall be explained in Section 4.6.

[3] m and s being the number of elements on ΔM and ΔS, respectively.

4.5.3 Symbolic Log Replay

Our Symbolic Log Replay will not recreate the whole file system state. Since the transaction's initial state contained any operation performed on TxSv prior to its beginning, which stems from the fact that the transaction's snapshot is a logical copy of the TxSv at the time the transaction was started, we can safely assume that a transaction T_i does not conflict with any transaction T_n that have successfully committed before T_i started. Therefore, there is no reason to replay all operations ever performed on TxSv, as such would not provide any advantage while validating the transaction and would introduce unnecessary overhead.

However we do need to replay all the operations of all the transactions that may have committed after T_i started (ΔM), since T_i did not see them happening and does not know in which state these operations left the TxSv. Thus, the Symbolic Log Replaying algorithm is divided in two different phases:

1. Creates an in-memory representation of all operations that led the TxSv from the last known state by T_i (i.e., from T_i's Master Copy) up to the TxSv's current state; and

2. Will attempt to apply each and every operation in the transaction's operation log (ΔS), detecting any conflict that may arise in the process.

The first phase, which we shall describe in Section 4.5.4 in detail, will have the additional responsibility of mapping each and every file creation onto the transaction's snapshot. This necessity results from a peculiarity of Btrfs' subvolumes and snapshots: being autonomous entities from Btrfs' point-of-view, each one has its own inode allocation scheme, which eventually leads to the same inode values being used for different files, much like what happens between different file system volumes.

In Table 4.1 we represent the sets used during the Symbolic Log Replaying algorithm, what is expected from each one of them, and how they are composed. Among these sets we include both ΔM and ΔS, and we define six new sets: *M2S*, which maps TxSv's inode values onto their assigned snapshot inode values; *Dir*, which is the actual representation of the effects of each operation belonging to ΔM, as reproduced by the algorithm; *REM*, the set of files, originally from the transaction's Master Copy, that are removed during the log replaying; *NLink*, associating an inode value to its *links* count number, which is to be maintained during the log replay; *Blocks*, keeping all the block intervals written and the file they were written to; and *DirtyDirs*, which simply holds the inode value of each directory modified during the log replay. All these sets will be actively used during the Symbolic Log Replay algorithm, and some of them will also be used later on, if the transaction reaches the reconciliation phase.

Populating these sets, however, is not a simple matter of traversing ΔM and ΔS, adding each and every operation onto a given set, specially considering that TxBtrfs currently supports over twelve different file system operations. However, most of the

pi	Parent Inode Value (unsigned 64-bit)
fn	Filename (string)
fi	File's Inode Value (unsigned 64-bit)
nl	Inode's Number of Links (Integer)
origin	{ Local, Global }
op	Operation issued
data	Specific operation data
Used solely while validating ΔM:	
lpi	Local Parent Inode
lfi	Local File Inode

(a) Legend

ΔM =	(op × (pi × fn × fi × nl × data))
ΔS =	(op × (pi × fn × fi × nl × data))
Inode =	unsigned 64-bit value
M2S =	Inode × Inode
Dir =	pi → (fn × fi × origin)
REM =	{ (pi × fn × fi × origin) }
NLink =	Inode → Integer
Blocks =	(pi × fn × fi × [start, end] × origin)
DirtyDirs =	{ pi }

(b) Sets

Table 4.1: Sets used during the Symbolic Log Replay

supported operations have similar effects when being applied during the Symbolic Log Replay, which allow us to decompose them in a few sets of elementary operations. For instance, in Table 4.2a and in Table 4.2b, we present how we decompose all of the operations that actively modify the file system structure — i.e., creation, removal and rename operations — and those accessing files and directories while applying the Symbolic Log Replay algorithm.

	Create	Unlink	Link
Create	✓		
Unlink		✓	
Link			✓
mkdir	✓		
rmdir		✓	
Rename		✓	✓
Symlink	✓		
mknod	✓		

(a) Operations over the FS structure

	Read	Write	Unchanged
Read	✓		
Write		✓	
mmap[4]	✓	✓	
Truncate[5]		✓	
Read Dir			✓

(b) Operations over file & directory contents

Table 4.2: Decomposition of operations that modify the FS.

In the next sections we shall describe in further detail how the Symbolic Log Replay algorithm works, focusing on its two main steps: processing the ΔM and applying its operations onto the the sets from Table 4.1b; and processing, validating and applying each operation in ΔS against the sets resulting from the previous step. Traversing both ΔM and ΔS in this fashion allows us to process both logs with a far more reduced overhead

[4] In reality, a `mmap` operation is not decomposed into a Read and a Write. In fact, a `mmap` is always a Read operation, though we use this notation for simplicity's sake, since its pages are written through an auxiliary, non-POSIX method. We log both operations.

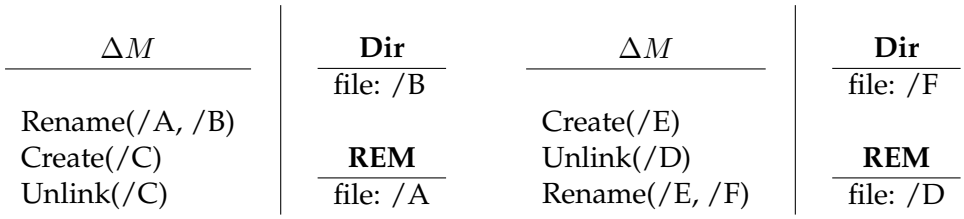
[5] How to deal with `truncate(2)` is still under study, and should be taken as future work.

than the approach initially considered (as described in Section 4.5.2), while creating the necessary conditions to validate each of ΔS's operations against the file system state they should be performed on and detect any potential conflict that may arise.

4.5.4 Replaying ΔM

The objective of the Symbolic Log Replay algorithm when processing ΔM is to obtain a representation of all the actual modifications sustained by the TxSv, while T_i executed in isolation (T_i being the transaction that is now trying to commit). Obtaining such a representation is essential to avoid conflicting either due to mock modifications during ΔM or due to operations that could trigger a conflict but, being intermediate, their real effects don't actually conflict with T_i (in Figure 4.4 we represent simplified ΔM logs and the effects of the operations it contains on the different sets).

For instance, take Figure 4.4a, in which we exemplify a mock operation on `/C`, where ΔM is represented as containing a `rename(2)` operation, which is decomposed in a *Link/Unlink* sequence, and both a `create(2)` and an `unlink(2)` over a file `/C`. The rename works over `/A` which was not created during ΔM, thus it must already exist in T_i's Master Copy; this means the algorithm must create a representation in which `/B` now exists (*Link*) and `/A` has been removed (*Unlink*). Similarly, the create operation over `/C` will add an entry to *Dir*, representing a creation T_i does not know about. However, ΔM also issues an unlink over `/C` and, because `/C` was created during ΔM, being unlinked during ΔM is the same as if it never existed at all. If we were using a simple list traversal as described back in Section 4.5.2, we could have triggered a conflict on `/C` that the Symbolic Log Replay algorithm is able to avoid.



(a) Mock operation on **/C** (b) Intermediate operation on **/E**

Figure 4.4: Mapping ΔM onto Symbolic Log Replay's sets

Another trivial example on the benefits of Symbolic Log Replay may be seen in Figure 4.4b. Since we are only interested in representing the state a transaction T_i is trying to commit against, we may ignore intermediate operations that could trigger a conflict if they were checked one by one. For instance, the final state represented in this figure only contains a new creation of `/F` and a removal of `/D` (which must have originated in T_i's Master Copy). The creation of `/E` is ignored, because the `rename(2)` operation will perform an unlink on `/E`, and since `/E` was created during ΔM both its creation and deletion may be gracefully ignored — neither are part of the TxSv's state the transaction

is trying to commit against.

When committing a transaction in TxBtrfs, it is not the ΔM's state that must be compatible with the transaction's state, but rather the other way around. Therefore, during the Symbolic Log Replay algorithm we always assume that processing ΔM is a straight-to-the-point kind of operation, with nothing to be validated: ΔM is consistent with the file system, as it represents (at least) part of the operations already applied onto the TxSv. All there is left to do, while processing ΔM, is to make sure that all the operations are added to the sets defined in Table 4.1b, so we may validate the transaction and detect any conflict that may arise when the time for replaying ΔS comes (Section 4.5.5).

Replaying ΔM then becomes straightforward, without requiring the algorithm to make any more decisions than those regarding which sets should be manipulated, depending on the entry being processed. As established, each entry in ΔM represents an operation performed on TxSv; however, each entry will not only include the performed operation, but it will also provide all the informations that may be necessary for their correct in-memory representation. For each entry in ΔM, the Symbolic Log Replay algorithm will check which kind of operation is being processed (following the operation decomposition as described in Table 4.2) and perform the according set-manipulation steps (Algorithm 1).

Algorithm 1: Symbolic Log Replay algorithm's first step: ΔM Replaying

```
1  forall the entry ∈ ΔM do
2  |  // lpi is the local representation of the entry's parent inode
3  |  // lfi is the local representation of the entry's file inode
4  |  if (entry.pi, i) ∈ M2S then
5  |  |  lpi ← i
6  |  else
7  |  |  lpi ← entry.pi
8  |  end
9  |  switch entry.op do
10 |  |  case Create
11 |  |  |  call Create_M(entry, lpi) // Algorithm 2
12 |  |  endsw
13 |  |  if (entry.fi, i) ∈ M2S then
14 |  |  |  lfi ← i
15 |  |  else
16 |  |  |  lfi ← entry.fi
17 |  |  end
18 |  |  case Unlink
19 |  |  |  call Unlink_M(entry, lpi, lfi) // Algorithm 4
20 |  |  case Link
21 |  |  |  call Link_M(entry, lpi, lfi) // Algorithm 3
22 |  |  case Write
23 |  |  |  call Write_M(entry, lpi, lfi) // Algorithm 5
24 |  |  case Truncate
25 |  |  |  // Still Missing!!
26 |  |  endsw
27 |  endsw
28 end
```

In Algorithm 1 we define a couple of local variables, `lpi` and `lfi`, representing a file's parent inode and the file's inode value, respectively. We need these variables because a given file's inode provided in a ΔM's entry may not be directly mapped to the same value during Symbolic Log Replay's execution. This behavior is quite specific to any operation decomposed to a Create, and results from Btrfs' inode allocation policy among snapshots or subvolumes, which allows them to allocate the same inode values as if they were different file system volumes. Therefore, whenever a Create operation is processed (Algorithm 2), the algorithm will define a new inode value for the entry being created, and will map the original inode value (provided in the ΔM entry) to the newly allocated value, using the set `M2S` for that purpose. Both `lpi` and `lfi` will then keep the inode value expected during Symbolic Log Replay execution: if the entry's inode exists in `M2S`, then the inode must have been allocated during the algorithm's execution; otherwise, the inode must originate in the Master Copy — using `M2S` to assess the origin of an inode is common practice on the Symbolic Log Replay algorithm, both in ΔM and ΔS replay.

When a Create operation is replayed during ΔM, besides having a new inode value being allocated and inserted in `M2S`, the replayed entry must also be inserted into the *Dir* set, associating the parent inode to the entry being added, in order to obtain a representation of the created file within the file system hierarchy. However, the entry added into `Dir` is marked as being a *Global* entry. We believe this nomenclature to be more straightforward when describing where the entry originated: the global file system state, instead of the transaction's private snapshot (*Local*). This distinction between local and global entries will be rather helpful when detecting conflicts during ΔS's replay (Section 4.5.5).

Algorithm 2: Method Create$_M$ (ΔM Replay)

Input: entry, lpi
Result: **void**

```
1 lfi ← new Inode // generates a new inode value belonging to the
      snapshot
2 Dir ← Dir ∪ {lpi → (entry.fn, lfi, global)}
3 M2S ← M2S ∪ {entry.fi → lfi}
4 NLink ← NLink ∪ {lfi → 1}
5 DirtyDirs ← DirtyDirs ∪ { lpi }
```

Much like processing a create operation, when it comes replaying a *Link* (Algorithm 3), the algorithm will add a new entry to `Dir`. Although, since a `link(2)` (or any operation decomposed that may act as such) is issued over an existing inode, we do not require to allocate a new inode value: we rely on `lfi` as the inode value, which should have been determined *a priori*.

However, the *Unlink* operation is, by far, the most complex of them all. Replaying an unlink (Algorithm 4) does not only affect the `Dir` set, but may also affect the *REM* set, depending on the file being unlinked. If the unlink being replayed was issued on a file created during ΔM, then all the Symbolic Log Replay algorithm must do is to remove the entry previously added to `Dir` (during the replay of the create operation). On the other

Algorithm 3: Method Link $_{\mathsf{M}}$ (ΔM Replay)

Input: entry, lpi, lfi
Result: **void**

```
1 Dir ← Dir ∪ {lpi → (entry.fn, lfi, global)}
2 DirtyDirs ← DirtyDirs ∪ { lpi }
3 if (entry.fi, _) ∈ M2S then
4 |   nlinks ← NLink[lfi]
5 |   NLink[lfi] ← nlinks + 1
6 else
7 |   NLink ← NLink ∪ { lfi → entry.nl }
8 end
```

hand, if the file originates in the transaction's Master Copy, then we must keep track of this removal, in order to enable the algorithm to detect any conflict that may arise if this file was used during the transaction's execution. Keeping track of this kind of removals is made by inserting an entry, referring to the removed file, in the `REM` set for posterior use by the ΔS replay phase.

Algorithm 4: Method Unlink $_{\mathsf{M}}$ (ΔM Replay)

Input: entry, lpi, lfi
Result: **void**

```
1 if (entry.fi, i) ∈ M2S then
2 |   Dir ← Dir \ {lpi → (entry.fn, lfi, global)}
3 |   nlinks ← NLink[lfi]
4 |   NLink[lfi] ← nlinks - 1
5 else
6 |   REM ← REM ∪ {(entry.pi, entry.fn, lfi, global)}
7 |   NLink ← NLink ∪ { lfi → entry.nl }
8 end
9 DirtyDirs ← DirtyDirs ∪ { lpi }
```

Besides the operations that modify the file system's structure, ΔM also contains *Write* operations over files. These are the only operations over files that we keep. The reason is simple: we do not need the reads issued during the time-frame ΔM represents, since those were associated with transactions that have already committed and pose no danger for the transaction now trying to commit.

In reality, we do not log only issued `writes(2)`. A write operation, when processing ΔM, may be any operation that can be decomposed to a write (Table 4.2b), such as a `mmap(2)` or a `truncate(2)`. That aside, what is relevant from the ΔM replay's point-of-view is to determine if the file being written to was created during ΔM's replay. If so, then there is no point replaying the write for two reasons: first, being a file created during ΔM's replay, there is no way possible for the current transaction to access this file, thus never triggering any conflict; secondly, when reconciling the file system (which shall only be explained in Section 4.6), all blocks from a file created during ΔM will be reconciled when the file creation is reconciled. However, if the file being written to exists originally on the transaction's Master Copy, then there is the chance for a conflict, since it is possible

for the transaction to access the file during execution. That being said, the Symbolic Log Replay algorithm needs to keep the access (in REM) to the given block interval, specified by the entry being replayed, so it may be validated during ΔS replay.

Algorithm 5: Method Write $_{\mathsf{M}}$ (ΔM Replay)

Input: entry, lpi, lfi
Result: **void**

```
1 if (entry.fi, i) ∉ M2S then
2 |   if entry.fi ∉ NLink then
3 |   |   NLink ← NLink ∪ { lfi → entry.nl }
4 |   end
5 |   Blocks ← Blocks ∪ {(lpi, entry.nf, lfi, [entry.start, entry.end], global)}
6 end
```

4.5.5 Replaying ΔS

Once ΔM is fully replayed, the Symbolic Log Replay algorithm has enough information about the current state of the TxSv in order to process the transaction's ΔS. The algorithm then proceeds to replay ΔS with the objective of assessing if there is a conflict between any of the operations performed by the transaction and current state of the TxSv, now represented by the sets presented in Table 4.1b and which have been populated during ΔM's replay.

Conflict detection relies on a set of checks highly dependent on the type of the operation performed, as summarized by Table 4.3. Each operation has a specific semantics and some of them work on a different grain than others, which forces us to take a different approach for each operation and dealing with them individually. For instance, the *Create* operation works on a *Directory Entry* basis, along with the *Link* and *Unlink* operations. This means they will trigger a conflict if their effects on a directory clash with some operation replayed during ΔM replay. On the other hand, *Read* and *Write* work mainly on the file's block level, although both operations have strong ties to the directory entry they operate on. We shall describe in greater detail how each operation is validated, since each one presents some corner cases not easily described in a single table.

The Symbolic Log Replay algorithm processes ΔS in a similar fashion as it does for ΔM, handling each entry in ΔS one at a time (Algorithm 6). Nevertheless, ΔS will also contain read operations both over files and directories, which ΔM did not. Keeping such operations serves the purpose of detecting potential conflicts between what the transaction read during its execution, and what actually exists in the TxSv at the time of its commit.

Replaying an operation from ΔS means we have to try to replay its effects on the sets populated during ΔM's replay (Section 4.5.4). This is how we will detect conflicts or declare an operation as *valid* according to the transactional semantics provided by TxBtrfs. Besides conflict detection, which by itself makes the replay of ΔS not as straightforward as ΔM's, replaying each operation also imposes an additional complexity when trying to

		ΔM				
		Create	Link	Unlink	Write	Truncate
ΔS	Create	×	×	—	—	—
	Link	×	×	×	—	—
	Unlink	✓	✓	×	—	—
	Read Dir	×	×	×	—	—
	Read	—	—	×	×	*
	Write	—	—	×	—	*
	Truncate	—	—	×	*	*

× = Conflict ✓ = No Conflict — = Does not apply
* = Under Study

Table 4.3: Conflict Detection when ΔS operations are validated against ΔM's

Algorithm 6: Symbolic Log Replay algorithm's second step: ΔS Replaying

```
1  forall the entry ∈ ΔS do
2      switch entry.op do
3          case Create
4              call Create_S(entry) // Algorithm 7
5          case Unlink
6              call Unlink_S(entry) // Algorithm 8
7          case Link
8              call Link_S(entry) // Algorithm 9
9          case ReadDir
10             call ReadDir_S(entry) // Algorithm 10
11         case Read
12             call Read_S(entry) // Algorithm 11
13         case Write
14             call Write_S(entry) // Algorithm 12
15         case Truncate
16             call Truncate_S(entry) // Algorithm 13
17         endsw
18     endsw
19 end
```

maintain a certain degree of compatibility between the transactional semantics and what is expected from executing a group of operations on a common file system.

Take Algorithm 7, where we present the conflict condition for a *Create* operation replayed while processing ΔS. In this case we only manipulate the `Dir` set, checking if (after ΔM's replay) exists any entry with the same name, and in the same parent directory, as the one we are trying to create. Since we only log operations that actually affected the transaction's snapshot, we can be assured that there is no clashing with an entry created during ΔS replay, thus our attempt to locate a *global* entry (in line 1). Even if the algorithm previously replayed an operation from ΔS that created a clashing entry, we know that it must have been unlinked in the meantime. Following the same reasoning, imagine that (by some reason) the algorithm is trying to replay the creation of a file that existed in `REM` (assuming the same parent), having been removed during ΔM's replay.

This would mean that the file we are trying to create must have originated in the transaction's Master Copy and removed during ΔM. However, if the transaction was able to create a file with the same name, in the same parent, as a file that had originated in the Master Copy and removed in ΔM, the only explanation possible is that the this same file had been removed in ΔS some time prior to this create we are now replaying. Therefore, those verifications should have been made during the unlink validation and are not meant to be done during a create.

Algorithm 7: Method Create$_S$ (ΔS Replay)

```
   Input: entry
   Result: Conflict ∨ Nothing, iff valid)
1  if entry.pi → (entry.fn, _, global, _) ∈ Dir then
2  |  return Conflict
3  else
4  |  Dir ← Dir ∪ { entry.pi → (entry.fn, entry.fi, local, entry.nl) }
5  |  NLink ← NLink ∪ { entry.fi → 1 }
6  end
```

Replaying an *Unlink* operation does exactly what has just been discussed: it checks `REM` for a record of a prior unlink of the file during ΔM, thus freeing any subsequent *Create* of that task. If `REM` does contain an entry referring to the same file the unlink operation refers to, then the operation from ΔS leaves the file system in the same state as the one left by ΔM, thus we could assume no conflict would be triggered. However, and as stated back in Table 4.3, we do assume a conflict in case of an *Unlink-Unlink* clash, and this results from a tradeoff between what is could be expected from a common file system — the unlink would have no impact whatsoever — and what is expected from a transactional file system — the transaction's unlink may have been based on a file system state that considered the file as existent.

Algorithm 8: Method Unlink$_S$ (ΔS Replay)

```
   Input: entry
   Result: Conflict ∨ Nothing, iff valid
1  if (entry.pi, entry.fn, entry.fi, global) ∈ REM then
2  |  return Conflict
3  else
4  |  if entry.pi → (entry.fn, entry.fi, local, _) ∈ Dir then
5  |  |  Dir ← Dir \ {entry.pi → (entry.fn, entry.fi, local)}
6  |  |  nlinks ← NLink[entry.fi]
7  |  |  NLink[entry.fi] ← nlinks - 1
8  |  else
9  |  |  REM ← REM ∪ { (entry.pi, entry.fn, entry.fi, local) }
10 |  |  NLink ← NLink ∪ { entry.fi → entry.nl }
11 |  end
12 end
```

Unlike a *Create*, which creates a new file by creating a new inode and associating a name to it, a *Link* actually takes an existing file and associates a new name to it. This

means that replaying a *Link* (Algorithm 9) may trigger an additional conflict than replaying a *Create*: if the source file, to which the new name should be associated, has been removed during ΔM then we have a conflict. However, similarly to replaying a *Create*, a conflict may also be triggered if the new name clashes with another name created during ΔM's replay.

Algorithm 9: Method Link$_S$ (ΔS Replay)

```
   Input: entry
   Result: Conflict ∨ Nothing, iff valid
1  if (entry.pi_1, entry.fn_1, entry.fi_1, global) ∈ REM then
2  |  // Removed during ΔM.
3  |  return Conflict
4  else if entry.pi_2 → (entry.fn_2, _, _, _) ∈ Dir then
5  |  // The destination file was created during ΔM
6  |  return Conflict
7  else
8  |  Dir ← Dir ∪ {(entry.pi_2, entry.fn_2, entry.fi_1, local) }
9  |  if (entry.fi_1 ∈ NLink) then
10 |  |  nlinks ← NLink[entry.fi_1]
11 |  |  NLink[entry.fi_1] ← nlinks + 1
12 |  else
13 |  |  NLink ← NLink ∪ { entry.fi_1 → entry.nl }
14 |  end
15 end
```

All the operations we have just discussed fall in the category of operations that have their conflicts detected on a directory entry basis. This basically means that we detect whether one of these operations conflicts with some other operation from ΔM based on a clash of names belonging to the same directory. However, they may also clash with another operation that works on a coarser grain: the *Read Dir* (Algorithm 10). In reality, it is the *Read Dir* that will clash with any *Create*, *Link* or *Unlink* that may happen in a given directory during ΔM's replay. The coarser grain of this operation results from the fact that we take it as being an atomic operation — i.e., once the process issues a `readdir(3)` during its transaction, it is as if it read the whole directory — even though this may not be expected from a POSIX file system, and should be considered for further development as future work. In any case, assuming this atomic behavior, it becomes impossible to assess whether the transaction read all the directory's entries or just a few, and a conflict is triggered because there is no way to detect an individual conflict on the directory entry level (for instance, reading an entry that was removed during ΔM's replay).

Regarding conflict detection on operations over files, we consider the grain of access on such operations to be the file's block. The difference between replaying a *Read* operation and any other operation is that the read will conflict with any *Write* replayed during ΔM if, and only if, both operations access the same block(s). Nevertheless, replaying both a *Read* (Algorithm 11) or a *Write* (Algorithm 12) will attempt to detect a conflict with a removal of the file being read or written. In this case, a conflict will only be triggered

Algorithm 10: Method ReadDir $_S$ (ΔS Replay)

Input: entry
Result: **Conflict** $\vee$ **Nothing, iff valid**
if $entry.fi \in DirtyDirs$ **then**
  /* Something was either created or removed from the directory with inode *entry.fi* */
  return *Conflict*
else if $(entry.pi, entry.fn, entry.fi, global) \in REM$ **then**
  // The directory was previously removed
  return *Conflict*
end

if the file being accessed by either one of the operations originates on the transaction's Master Copy, and if it has been removed during ΔM's replay (i.e., if it belongs to `REM`). If the file being read or written was created during ΔS, then no conflict should ever arise.

Algorithm 11: Method Read $_S$ (ΔS Replay)

Input: entry
Result: **Conflict** $\vee$ **Nothing, iff valid**
// Validate Reads only against writes from ΔM
if $(entry.pi, entry.fn, entry.fi, [s, e], global) \in Blocks : [s, e] \cap [entry.start, entry.end] \neq \emptyset$
  $\vee\ (entry.pi, entry.fn, entry.fi, global) \in REM$ **then**
  return *Conflict*
end

Algorithm 12: Method Write $_S$ (ΔS Replay)

Input: entry
Result: **Conflict** $\vee$ **Nothing, iff valid**
if $(entry.pi, entry.fn, entry.fi, global) \in REM$ **then**
  return *Conflict*
else
  if $entry.fi \notin NLink$ **then**
    NLink $\leftarrow$ NLink $\cup$ { entry.fi $\rightarrow$ entry.nl }
  end
end

Currently we are not able to guarantee full detection of conflicts in case of a `truncate(2)`. This is both due to all the corner cases inherent to this operation when it comes to conflict detection, as well as some technical issues regarding our implementation. Algorithm 13 describes how far we have gotten when detecting a conflict replayed on ΔS, although it should be taken with a grain of salt, as we have defined the case common to most operations — when the file being truncated is removed during ΔM's replay — and a highly pessimistic case, in which we conflict whenever there is a write operation over the file during ΔM's replay.

Once the Symbolic Log Replay algorithm finishes validating the transaction's ΔS, we will now be able to decide what to do with the transaction: abort it, considering

Algorithm 13: Method Truncate$_S$ (ΔS Replay)

Input: entry
Result: **Conflict** $\vee$ **Nothing, iff valid**
1 **if** $(entry.pi, entry.fn, entry.fi, global) \in REM$ **then**
2 | **return** *Conflict*
3 **else if** $(\_, \_, entry.fi, \_, global) \in Blocks$ **then**
4 | **return** *Conflict*
5 **end**

the commit as failed and discarding both the transaction and its private snapshot, or proceed with the reconciliation phase, which will produce an unified file system state consistent with all the operations that happened before the transaction started, with those that happened while the transaction executed and with all the operations performed by the transaction until it started its commit.

4.6 Reconciliation & Commit

After successfully validating the transactions ΔS against the TxSv's ΔM, we can now use the resulting sets from executing the Symbolic Log Replay algorithm in order to reconcile the transaction's private snapshot (T_s) with the TxSv. This reconciliation will produce an unified state ($TxSv_U$), which shall become the new TxSv when the transaction finishes its commit process ($TxSv'$), as illustrated in Figure 4.5.

Unlike typical transactional file systems, which apply the transaction's changes directly onto the file system, TxBtrfs does just the opposite: it applies the operations executed on the file system (i.e., the TxSv) during the transaction's execution into the transaction's private snapshot, and then makes this one the main copy. This approach appears to be far from usual, and up to now we have not found any work on file systems using a similar approach.

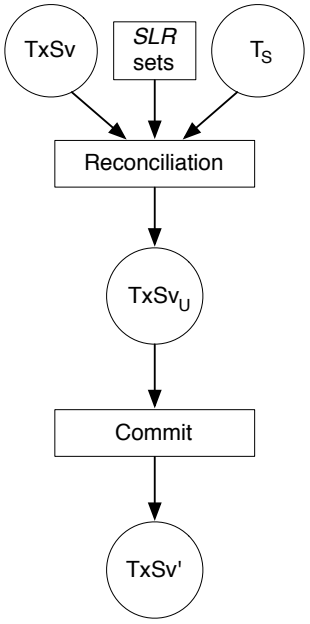


Figure 4.5: Reconciliation & Commit Process

However, one can relate this reconciliation method with what is used by revision control software such as Subversion (SVN) [Apa]. In SVN, a local copy is made from the master repository, stored and modified locally; once it is decided to commit the local changes, one of two scenarios may come up: a) the master repository was changed in the meanwhile, and it is necessary to obtain those changes, validate them and reconcile them with the local copy before committing them into the master; or b) there was no change in the master and it is possible to freely commit the changes. This behavior guarantees the master isn't left in an inconsistent state after a local copy is committed, in case of a conflict between modifications.

Transposing this behavior to TxBtrfs is quite straightforward, as long as we assume the concept of *master repository* as being a *pointer* to the most recent stable file system state. If this assumption was valid on SVN, once the local copy was validated and reconciled with the master repository, then the local copy would become the new master repository. Although this model would clash with the centralized nature of SVN, it fits just perfectly in TxBtrfs.

In TxBtrfs we keep both the TxSv and the transaction's snapshot (T_S) in the same, local, Btrfs volume, and the snapshot follows a *Copy-on-Write* behavior, which avoids unnecessary waste of storage by shared files. This gives us some flexibility when reconciling the TxSv and the T_S, since we only have to manipulate the internal file system structures in order to make, for instance, T_S contain a file created on the TxSv. However, if the TxSv had suffered no change whatsoever, and no reconciliation was required, all TxBtrfs would need to do to commit the transaction would be to state "The new TxSv is T_S", which can be seen as changing a pointer from TxSv to T_S [6].

By taking this approach, using a *pull model* — i.e., applying the modifications made on TxSv, which are unknown to the transaction, onto the transaction's private snapshot — instead of directly modifying the TxSv, we are able to easily guarantee all of the ACID properties we proposed ourselves to provide in TxBtrfs, while still allowing concurrency during the whole commit process (i.e., validation/conflict detection, reconciliation and, finally, the commit).

4.6.1 Providing ACID

We discussed the ACID properties back in Section 2.4.1, but we never clarified how we would guarantee them in TxBtrfs. By now we have detailed quite enough about TxBtrfs in order to explain how this set of properties are guaranteed.

Let us begin with *Durability*, which should enforce that, after a transaction successfully commits, a transaction's effects are durably kept on stable storage. In Section 4.2.1 we explained that each transaction is associated with a private snapshot of TxSv. This snapshot is safely kept on disk, from the moment the transaction starts up to the moment it commits. Each operation performed is kept in the transaction's snapshot in a durable manner as the one provided by Btrfs. If the transaction in given the *green light* to commit, its snapshot will become the new TxSv, becoming now available as being part of the file system, yet without compromising its durability.

The *Isolation* property is guaranteed from the moment the transaction starts and all its accesses are mapped onto its own private snapshot. From the transaction's perspective, there is no one else modifying the file system, simply because that is exactly what is happening: nobody else, outside the transaction, modifies the transaction's private snapshot. Analogously, since the transaction is directly mapped onto its private snapshot, the transaction is unable to perceive any modifications either on TxSv or on other

[6] In reality, this process is a little more complicated than simply assigning a pointer, since it requires modifying internal file system structures in multiple disk locations.

transactions (although it may perceive any modification outside the transactional scope of TxBtrfs).

Guarantees for these two properties are assured by capabilities explained already explained previously in this document. However, the remaining two properties — *Atomicity* and *Consistency* — haven't been directly detailed, albeit the path for their explanation has been laid.

Take the *Consistency* property, which must assure the file system always transitions between consistent states. In a nutshell, this means that every transaction should be able to see all the modifications previously made by a successfully committed transaction, but should not be allowed to commit operations that depend on non-committed operations and data. Mapping transactions onto their private snapshots and reconciling them with the TxSv with a *pull model*, allows us to never actually change the current TxSv. Instead, all the modifications that should be replayed in order to obtain a reconciled state will be applied onto the transaction's private snapshot. If by any chance any of them fails, or if the system fails abruptly during the reconciliation process, no harm was done to the TxSv. This method therefore guarantees that in case of an unexpected problem, the TxSv will be kept in its initial state, and nowhere in-between its initial state and the state in which the transaction is fully reconciled and committed. Nevertheless, guaranteeing the *Consistency* property not only depends on how we deal with unexpected problems during the reconciliation, but also on how we deal with them while committing the reconciled state (i.e., defining the snapshot as the new TxSv). Therefore, once a transaction T finishes reconciling with the TxSv, TxBtrfs will guarantee that no other transaction starts until T finishes committing. Also, before changing the *pointer* from TxSv to T's snapshot, TxBtrfs will store enough on-disk informations to choose the correct file system state (either TxSv's or the T's snapshot) during the next mount, in case a problem occurs (such a system failure) while changing the file system's internal data structures.

Finally, TxBtrfs is able to guarantee *Atomicity* by taking advantage of all the capabilities already described. One of the pillars of this work is to provide atomicity to any application using TxBtrfs, guaranteeing that either all of its operations are successful and successfully applied on disk, or none are. Considering the method used to commit a transaction, just described in the context of *Consistency*, we are able to provide the *Atomicity* property just by making sure that no other transaction will start or commit while we are changing the *pointer* from TxSv to T's snapshot: i) Any transaction already executing will keep on executing on their private snapshots; ii) one transaction commits at a time, thus only when T finishes will any other transaction start its commit (and, by then, will validate and reconcile itself with the modifications made by T); and, iii) Any transaction attempting to start, thus creating a snapshot of the TxSv, will have to wait for the *pointer* to be changed to T's snapshot and will then start off from the new TxSv — as if all of T's operations happened in the same single moment in time.

5

Evaluation

5.1 Introduction

Defining how to evaluate TxBtrfs was a challenge right from the moment we started this work. Unlike most file systems, TxBtrfs provides transactional semantics; unlike databases, TxBtrfs is a file system with a POSIX interface and available as a Linux Kernel module. Due to this mixed paradigm we easily lose our chances of taking a *one-size-fits-all* kind of evaluation: we cannot simply test file system performance and POSIX compliance, as one would for a common file system; and we certainly cannot use an accepted database benchmark to evaluate our transactional semantics, conflict detection and reconciliation.

Due to our limited time timeframe, we decided to keep it simple and divide our evaluation into three different components: POSIX compliance, correctness of transactional semantics, and overhead introduced by TxBtrfs's additional features. Our implementation was evaluated on a system running Debian Linux 6.0, using an Intel dual-core i5 650 at 3.20 GHz processor, 4GB of RAM and with a Samsung P80 SD SATA hard disk.

5.2 POSIX Compliance

Among many specifications, the POSIX standard defines the interface and behavior of I/O calls. Linux vastly supports POSIX regarding I/O, albeit Linux not being fully POSIX compliant.

While designing TxBtrfs, we purposely kept it as POSIX compliant as Btrfs, by keeping our transactional support separated from the file system I/O interface. We did not changed any of the file system's I/O calls in a perceivable way for the application, and

any operation issued by the application during its execution will keep its expected behavior and return status. The only visible difference between TxBtrfs and other file systems is that an application may execute its operations in a transaction, by issuing file system specific IOCTL calls (`TxStart` and `TxCommit`), but, even then, all operations executed in a transaction will maintain its POSIX compliance up until commit.

We evaluated TxBtrfs's POSIX compliance by running the *pjd-fstests* POSIX Test Suite[Tux] inside a single transaction, and identifying which tests failed. Of all the 1957 tests ran by the suite only one failed, regarding the creation time of a file when a `truncate(2)` is issued with the purpose of increasing the file's size. This very same test fails when ran on Btrfs's version 0.19, which is the version we used as the code base to implement TxBtrfs.

It would be have been interesting if we had found a suite that could test the POSIX compliance of the `read(2)` and `write(2)` calls (*pjd-fstests* does not), although we believe there is no reason for them not be compatible: we do not alter the behavior of the operations when accessing the transaction's snapshot. Nevertheless, we think that having access to such a suite could allow us to (modify it and) test the POSIX compliance of `read(2)`'s and `write(2)`'s when running multiple transactions.

5.3 Implementation Evaluation

TxBtrfs introduces a large amount of instructions to Btrfs, summing up to over nine thousand new lines of code. These are the support for all we described during this document, but are mostly regarding snapshot creation on transaction start, operation logging during a transaction execution and the transaction's commit process (conflict detection and reconciliation). Therefore, our evaluation is not focused on assessing if we introduce overhead over a vanilla version of Btrfs, but on how much and why.

Given the components of TxBtrfs, in the course of this section we will produce two kinds of analysis: a broader assessment on the overhead imposed over Btrfs when accessing the file system, and an evaluation on the impact of the reconciliation process on the file system's performance. All the results presented are the average of (at least) five runs of each test, minus both the best and the worst results.

5.3.1 Throughput

We used the IOzone benchmark [IOz] to assess the throughput of TxBtrfs and to measure how it behaves against Btrfs. The benchmark was parametrized in the same way for both file systems, and between each test we made sure the VFS's page cache would be freed. We executed both read-only and write-only workloads, using files of 8 MB, 256 MB and 4 GB, with the intent to represent a rather large range of file sizes and to identify changes of behavior when using record sizes of 4 KB, 1 MB and 4 MB. In order to test TxBtrfs's mechanisms, we ran IOzone in a single transaction, which imposes onto IOzone the overhead resulting from logging the operations, as well as mapping the operations

onto the transaction's snapshots.

In Figure 5.1 we present the results of running IOzone with a read-only workload over TxBtrfs, and it becomes clear that, except for small files and using a small record size, the throughput is quite regular.

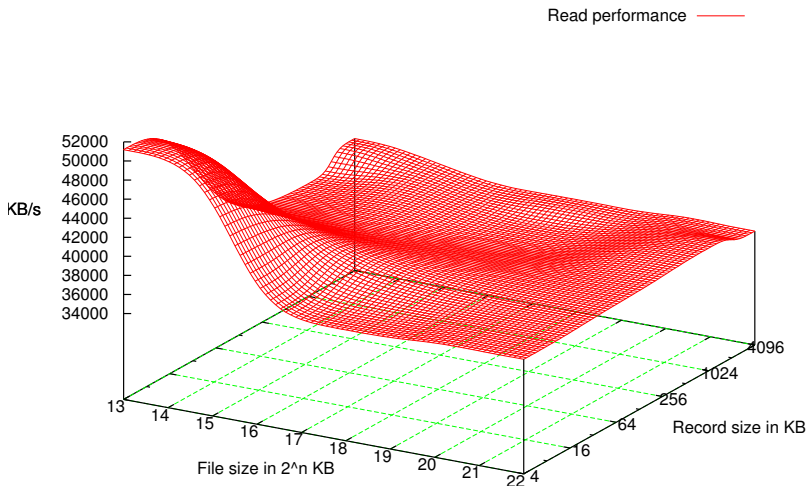


Figure 5.1: TxBtrfs's Throughput for Read operations

When compared to a vanilla version of Btrfs (Figure 5.2), we can conclude the overhead imposed by TxBtrfs is minimal, being quite close to Btrfs. We do have, however, an anomaly which was verified in almost all runs we made, making TxBtrfs considerably faster than Btrfs when reading small files using small record sizes. Since these results refer to 8 MB files, which curiously enough is exactly the size of our disk's buffer memory size, we suppose this strange behavior results from some unforeseen cache affinity created by TxBtrfs, which enables our implementation to leverage the disk's cache.

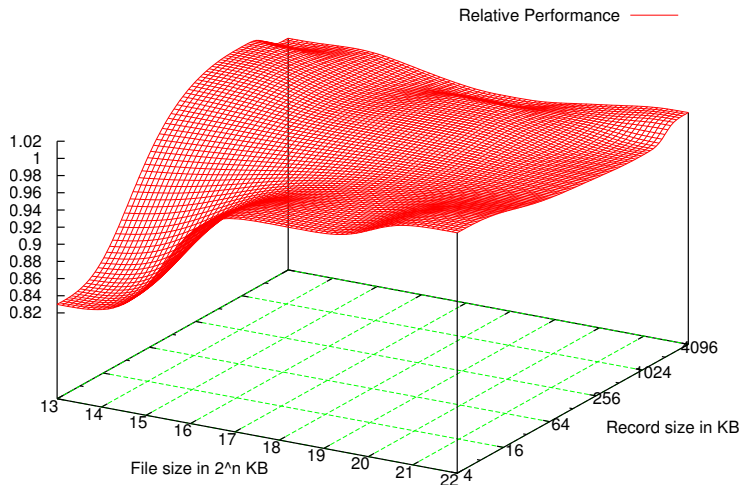


Figure 5.2: TxBtrfs's relative performance to Btrfs's, for Read operations

While the read-only workload's throughput had a maximum value of about 52 MB/s,

when executing write-only workloads we obtained throughputs several orders of magnitude higher, as shown in Figure 5.3. Unlike read operations, which will hang while waiting for the data to be retrieved from disk, write operations may not necessarily block while waiting for the data to reach disk. This is specially true in Btrfs, which caches writes in order to optimize disk accesses, guaranteeing that (eventually) they will reach their place in the disk. Our work simply inherits Btrfs's behavior.

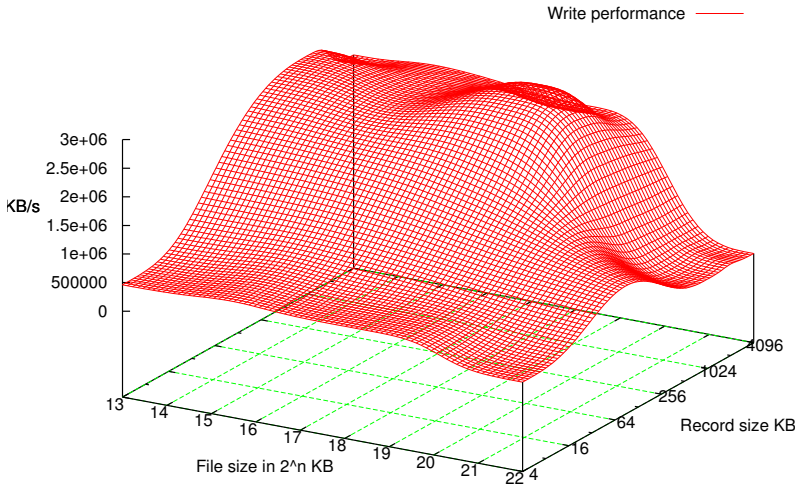


Figure 5.3: TxBtrfs's Throughput for Write operations

However, when comparing our results with Btrfs's, we verified some fluctuations we are not yet able to explain. These fluctuations are presented in Figure 5.4, and although our results are kept relatively near Btrfs's performance, they also spike towards both better and worst performance. These results should be further studied in order to assess the causes of such incongruence.

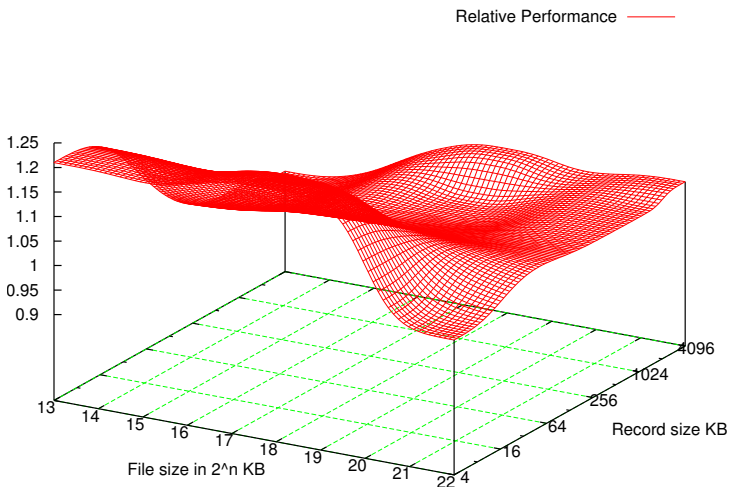


Figure 5.4: TxBtrfs's relative performance to Btrfs's, for Write operations

5.3.2 Conflict Detection and Reconciliation

In order to assess the minimal burden of conflict detection and reconciling a transaction upon commit, we decided to force an empty transaction to commit after the IOzone's transaction committed, guaranteeing that both transactions shared the same generation. Using the same testing scenarios used in Section 5.3.1, we implemented a simple application that would create a thread responsible for running IOzone within a transaction, and we made the application create a concurrent transaction. The application's transaction would share the same generation as the benchmark's transaction, and we guaranteed that it would only finish after benchmark's transaction finished. This way we were able to guarantee that the application's transaction would be reconciled with the new TxSv's state. By keeping the time the benchmark started and finished, and how long it took until the application was able to finish its transaction, we were able to obtain the time the conflict detection and reconciliation took for each test ran.

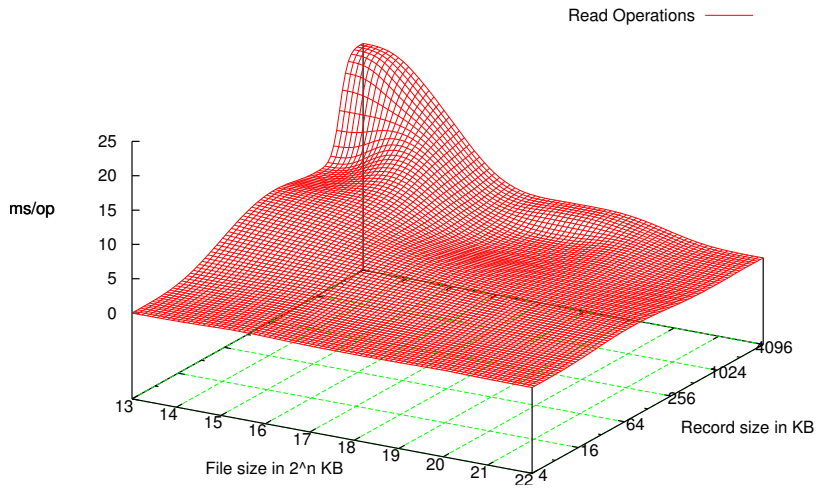
We verified that the time taken by TxBtrfs to reconcile a transaction does not change that much for the workloads used, regardless of the file and record size used. In fact, as shown in both Figure 5.5a and Figure 5.5b, the burden of the reconciliation gets diluted as the number of operations performed grows. The worst case, 25 milliseconds per operation, happens during the read-only workload and while accessing an 8 MB file with a 4 MB record size, which translates in a rather low number of operations and an equivalently small amount of execution time, therefore emphasizing the burden associated with the reconciliation. In reality, this is also the worst case for the write-only workload, but since this workload performs a lot more operations than the read-only workload, the impact of reconciling isn't as profound as it is in the read-only workload. For executions over 4 GB files, the cost of reconciling is nearly null.

5.4 Transactional Semantics Correctness

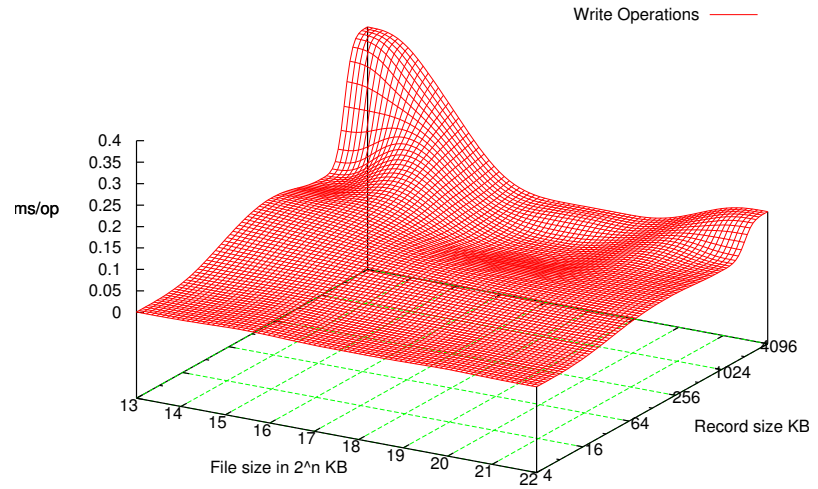
We subjected TxBtrfs to *Store Benchmark* [SLL11, Pes11], a benchmark being developed by the Transactional Systems Research Team (TrxSys) at Centro de Informática e Tecnologias da Informação (CITI). This benchmark mimics a storehouse, where products are kept, and within which three different entities are at work: *Clients*, which will select products and order them; *Workers* responsible of obtaining the ordered products in order to sell them to the Clients; and *Suppliers*, which will resupply the storehouse's shelves.

More precisely, each entity will be one or more threads, executing operations over the file system within the lines of the following model:

- Each product is a file.
- Each product is assigned a *type*.
- A predefined directory `ProductType` will keep all the existing product types.
- A predefined directory `ProductWorld` will keep all the products available for sale.



(a) Read-Only workloads



(b) Write-Only workloads

Figure 5.5: Reconciliation Time (ms) Per Operation

- Within `ProductWorld` there will be a directory for each existing product type.
- Each available product will be kept in the directory according to its type, inside the `ProductWorld` directory.
- Multiple instances of a same product (i.e., two boxes of cereals) within the `ProductWorld` will be assigned different identifiers.

The model's objective is to allow *Client* threads to order products, *Worker* threads to process those orders, and *Supplier* threads to resupply the `ProductWorld`. In a nutshell, Clients will lookup the `ProductWorld` and select random products, each lookup performing a `readdir(3)` without removing the product. Worker threads will then process the Client's order, by removing the requested products from the `ProductWorld` (i.e., `unlink(2)`) after verifying those products are available. Finally, a Supplier will copy or link random amounts of random products from the `ProductType` directory

onto the `ProductWorld`.

In order to verify the correctness of the transactional semantics offered by the file system, upon completion the benchmark checks the file system's state enforcing the following consistency constraint:

$$(InitialPopulation + TotalProductsSupplied) - TotalProductsSold = TotalRemaingProducts$$

Where *InitialPopulation* stands for the amount of products initially created in the file system, *TotalProductsSupplied* being the amount of products the Suppliers added to the `ProductWorld`, *TotalProductsSold* being the amount of products successfully ordered by Clients and sold by Workers, and *TotalRemainingProducts* the final amount of products existing in the file system when the constraint is enforced.

According to this benchmark, TxBtrfs successfully guarantees the expected file system consistency.

6

Conclusion

6.1 Summary

This work has the purpose of providing to applications the possibility of using transactional semantics when accessing the file system, allowing them to take advantage of the ACID properties directly embedded on the file system; these properties could also be found in Database Management Systems, but using them would imply a significant change in paradigm when performing I/O, which in the file system is kept as POSIX.

To provide the application's with this transactional semantics without forcing them to be dependent of external software, we focused our on extending a Linux Kernel file system, and we opted on using Btrfs as our code base. When we decided to extend Btrfs, we did so due to its native support for Snapshots, which we used as the basis to implement and support transactions within the file system, in order to keep the transaction's modifications isolated from the remaining file system.

The result of our work, TxBtrfs, provides transactional support to applications by means of explicit creation and finalization of transactions, to which are guaranteed the ACID properties — Atomicity, Consistency, Isolation and Durability — while imposing a negligible overhead compared to Btrfs. This claim is supported by our results regarding TxBtrfs's performance compared to Btrfs's, when executing the IOzone file system benchmark [IOz] within a transaction. Also, our measurements indicate that when using TxBtrfs for a large volume of disk operations, the overhead generated by detecting conflicts between transactions and reconciling them is practically null. We also partially validated the transactional semantics, using the *Store Benchmark* [SLL11, Pes11], and TxBtrfs was able to meet the consistency constraints imposed by the benchmark.

6.2 Future Work

There is quite a lot of work yet to be done in TxBtrfs. During the course of this dissertation, we created the necessary base to continue the work over our file system, and to push forward where we could not do it ourselves during the available timeframe. As future reference, we can pinpoint several issues that should be worked on, so TxBtrfs can achieve its full potential as a transactional file system.

If we had the time, we would have tackled the implementation regarding the following technical issues:

- Minimization the heavy memory dependence throughout the implementation, namely when mapping a process's operations into the process's transaction's snapshot, as well as when it comes to processing the transaction's operations log during Conflict Detection.
- Purging null operations (i.e., operations whose effects were nullified by a subsequent operation) from the transaction's operations log, which not only would reduce the memory footprint, but also the processor usage during the transaction's Conflict Detection and Reconciliation.
- Support for a `TxAbort` operation, responsible for finishing a transaction and removing its snapshot both from disk and memory, as if it never existed.
- Removal of past copies of the Transactional Subvolume (TxSv) that are no longer required to validate any running transaction. These copies are kept to build the ΔM (see Section 4.5.1), but once they are not required they should be purged from disk and memory in order to free resources.

Further study should also be made regarding the following subjects:

- Analysis on the `truncate(2)` and `fallocate(2)` operations's behavior, in order to support them when detecting conflicts and reconciling.
- Clearer definition of all the potential corner cases when providing file system operations with transactional semantics, taking into account their expected behaviors.
- Further validation of the defined transactional semantics through testing.
- Assessment of the requirements and implications on providing non-transactional accesses to the Transactional Subvolume.
- Support, and definition of the expected behavior, for conflict detection and reconciliation regarding file and directory metadata, such as attributes.
- How to deal with non-transactional accesses to the TxSv.

Bibliography

[ALS06] Kunal Agrawal, Charles E. Leiserson, e Jim Sukha. Memory models for open-nested transactions. In *Proceedings of the 2006 workshop on Memory system performance and correctness*, MSPC '06, pág. 70–81, New York, NY, USA, 2006. ACM.

[Ame92] American National Standard for Information Systems — Database Language — SQL. ANSI X3.135-1992, November 1992.

[Apa] Apache Software Foundation. Apache Subversion. `http://subversion.apache.org`, last checked on 17 September 2011.

[App] Apple Inc. Mac OS X Time Machine. `http://www.apple.com/macosx/what-is-macosx/time-machine.html`, last checked on 25 January 2011.

[BAH+] J. Bonwick, M. Ahrens, V. Henson, M. Maybee, e M. Shellenbaum. The zettabyte file system. Relatório técnico, Sun Microsystems.

[BBG+95] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, e Patrick O'Neil. A critique of ansi sql isolation levels. In *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, SIGMOD '95, pág. 1–10, New York, NY, USA, 1995. ACM.

[BBMT72] Daniel G. Bobrow, Jerry D. Burchfiel, Daniel L. Murphy, e Raymond S. Tomlinson. Tenex, a paged time sharing system for the pdp - 10. *Commun. ACM*, 15(3):135–143, 1972.

[BP98] S. Balasubramaniam e Benjamin C. Pierce. What is a file synchronizer? In *4th Annual ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM'98)*, 1998.

[Cac05] João Cachopo. Versioned boxes as the basis for memory transactions. In *Proceedings of Synchronization and Concurrency in Object-Oriented Languages Workshop*, Outubro 2005.

[Cod09] E. F. Codd. Derivability, redundancy and consistency of relations stored in large data banks. *SIGMOD Rec.*, 38:17–36, June 2009.

[Cun07] Gonçalo Cunha. Consistent state software transactional memory. Tese de Mestrado, Universidade Nova de Lisboa, November 2007.

[DSS06] Dave Dice, Ori Shalev, e Nir Shavit. Transactional locking ii. In *Distributed Computing*, volume 4167, pág. 194–208. Springer Berlin / Heidelberg, October 2006.

[Fed] Fedora Project. System rollback with btrfs. http://fedoraproject.org/wiki/Features/SystemRollbackWithBtrfs, last checked on 19 September 2011.

[FUS] FUSE Project. File System in User Space – FUSE. http://fuse.sourceforge.net/, last checked on 17 September 2011.

[GNS88] David K. Gifford, Roger M. Needham, e Michael D. Schroeder. The cedar file system. *Commun. ACM*, 31(3):288–298, 1988.

[Hip] D. Richard Hipp. SQLite software library. http://sqlite.org/, last checked on 25 January 2011.

[HLM94] Dave Hitz, James Lau, e Michael Malcolm. File system design for an nfs file server appliance, 1994.

[Inf] Knowledge Quest Infotech. ZFS-Linux on Github.com. https://github.com/zfs-linux, last checked on 25 January 2011.

[Int] Interop Systems Inc. SUA Community for Interix SUA & SFU. http://www.suacommunity.com/SUA.aspx, last checked on 17 September 2011.

[IOz] IOzone. IOzone File System Benchmark Home Page. http://www.iozone.org/, last checked on 29 January 2011.

[Iva] Ivan Smith. Nova Scotia's Electric Gleaner. Cost of hard drive storage space. http://ns1758.ca/winch/winchest.html, last checked on 21 January 2011.

[KS93] Puneet Kumar e M. Satyanarayanan. Log-based directory resolution in the coda file system. In *Proceedings of the Second International Conference on Parallel and Distributed Information Systems*, pág. 202–213, 1993.

[Lew91] Donald Lewine. *POSIX Programmer's Guide*. O'Reilly & Associates Inc., April 1991.

[LLL11] João Eduardo Luís, João M. Lourenço, e Paulo A. Lopes. Suporte transaccional para o sistema de ficheiros btrfs. In *INFORUM 2011*, 2011.

[Lom77] D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. *SIGPLAN Not.*, 12:128–137, March 1977.

[LR07] James R. Laurus e Ravi Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, first edition, 2007.

[Luí09] Nuno Lopes Luís. Sistema de Ficheiros Transaccional sobre FUSE. Tese de Mestrado, Universidade Nova de Lisboa, 2009.

[Mac07] Garrels Machtelt. *Introduction to Linux*. Fultus Corporation, 2nd edition, 2007.

[Mai] Community Maintained. Btrfs wiki at kernel.org. https://btrfs.wiki.kernel.org/, last modification on 11 January 2011.

[Mar08] Artur Martins. Transactional Filesystems. Tese de Mestrado, Universidade Nova de Lisboa, 2008.

[MG99] Marshall McKusick e Gregory Ganger. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. In *Proceedings of the Freenix Track: 1999 USENIX Annual Technical Conference*, pág. 1–17, 1999.

[Mica] Microsoft. FAT File System. http://technet.microsoft.com/en-us/library/cc938438.aspx, last checked on 27 January 2011.

[Micb] Microsoft. How Volume Shadow Copy Service Works: Data Recovery. http://technet.microsoft.com/en-us/library/cc785914(WS.10).aspx, last checked on 25 January 2011.

[MMS00] Jim Mauro, Richard McDougall, e Sun Microsystems Press. *Solaris(TM) Internals (Vol 1)*. Prentice Hall, 2000.

[MTV02] Nick Murphy, Mark Tonkelowitz, e Mike Vernal. The design and implementation of the database file system, 2002.

[MvRT+90] S.J. Mullender, G. van Rossum, A.S. Tananbaum, R. van Renesse, e H. van Staveren. Amoeba: a distributed operating system for the 1990s. *Computer*, 23(5):44 –53, Maio 1990.

[Nat] Nathan Rosenquist et al. rsnapshot, a remote filesystem snapshot utility, based on rsync. http://rsnapshot.org/, last checked on 25 January 2011.

[OA93] Michael A. Olson e Michael A. The design and implementation of the inversion file system, 1993.

[OBM99] Michael A. Olson, Keith Bostic, e Seltzer Margo. Berkeley DB. In *USENIX Annual Technical Conference Proceedings*, Monterey, CA, June 1999. USENIX.

[PB05] Z. Peterson e R. Burns. Ext3cow: A time-shifting file system for regulatory compliance. *ACM Transactions on Storage*, 1(2):190–212, 2005.

[Pes11] Vasco Pessanha. Verificação Prática de Anomalias em Programas de Memória Transaccional. Tese de Mestrado, Universidade Nova de Lisboa, 2011.

[PHR+09] Donald E. Porter, Owen S. Hofmann, Christopher J. Rossbach, Alexander Benn, e Emmett Witchel. Operating system transactions. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pág. 161–176, New York, NY, USA, 2009. ACM.

[Res] Jim Handy . Semico Research. Flash Memory vs. HDD costs over the years. http://www.storagesearch.com/semico-art1.html, last checked on 21 January 2011.

[Ric] Ricardo Correia (creator), currently maintained by the community. ZFS on Fuse. http://zfs-fuse.net/, last checked on 25 January 2011.

[Rob] Daniel Robbins. Advanced filesystem implementor's guide, Part 2. http://www.ibm.com/developerworks/library/l-fs2.html, last checked on 24 January 2011.

[Rod08] Ohad Rodeh. B-trees, shadowing, and clones. *Trans. Storage*, 3(4):1–27, 2008.

[Roe52] Anne Roe. *The Making of a Scientist*. Dodd, Mead & Company, 1952.

[SFH+99] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, e Jacob Ofir. Deciding when to forget in the elephant file system, 1999.

[SGG04] Abraham Silberschatz, Peter Baer Galvin, e Greg Gagne. *Operating System Concepts*. John Wiley & Sons, 7th edition, 2004.

[SKS06] Abraham Silberschatz, Henry F. Korth, e S. Sudarshan. *Database System Concepts*. International Edition. McGraw-Hill, 5th edition, 2006.

[SLL11] Daniel Santos, João Lourenço, e João Luís. Store Benchmark. Relatório técnico, Transactional Systems Research Team, Centro de Informática e Tecnologias da Informação, Universidade Nova de Lisboa, 2011.

[ST95] Nir Shavit e Dan Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pág. 204–213, New York, NY, USA, 1995. ACM.

[Tux] Tuxera. Posix test suite. http://www.tuxera.com/community/posix-test-suite/, last checked on 17 June 2011.

[Way] Wayne Davison (maintainer). rsync. http://samba.anu.edu.au/rsync/, last checked on 25 January 2011.

[Wik] Wikipedia. Versioning file systems. http://en.wikipedia.org/wiki/Versioning_file_system, last checked on January 25 2011.

[WLS+85] Dan Walsh, Bob Lyon, Gary Sager, J.M. Chang, D. Goldberg, S. Kleiman, T. Lyon, R. Sandberg, e P. Weiss. Overview of the sun network file system. In *USENIX Conference Proceedings*, pág. 117–124, Dallas, TX, 1985. USENIX.