Copy On Write Based File Systems Performance Analysis And Implementation

Sakis Kasampalis

Kongens Lyngby 2010
IMM-MSC-2010-63

Technical University of Denmark
Department Of Informatics
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

Abstract

In this work I am focusing on Copy On Write based file systems. Copy On Write is used on modern file systems for providing (1) metadata and data consistency using transactional semantics, (2) cheap and instant backups using snapshots and clones.

This thesis is divided into two main parts. The first part focuses on the design and performance of Copy On Write based file systems. Recent efforts aiming at creating a Copy On Write based file system are ZFS, Btrfs, ext3cow, Hammer, and LLFS. My work focuses only on ZFS and Btrfs, since they support the most advanced features. The main goals of ZFS and Btrfs are to offer a scalable, fault tolerant, and easy to administrate file system. I evaluate the performance and scalability of ZFS and Btrfs. The evaluation includes studying their design and testing their performance and scalability against a set of recommended file system benchmarks.

Most computers are already based on multi-core and multiple processor architectures. Because of that, the need for using concurrent programming models has increased. Transactions can be very helpful for supporting concurrent programming models, which ensure that system updates are consistent. Unfortunately, the majority of operating systems and file systems either do not support transactions at all, or they simply do not expose them to the users. The second part of this thesis focuses on using Copy On Write as a basis for designing a transaction oriented file system call interface.

Preface

This thesis was prepared at the Department of Informatics, the Technical University of Denmark in partial fulfilment of the requirements for acquiring the M.Sc. degree in engineering.

The thesis deals with the design and performance of modern Copy On Write based file systems, and the design of a transactional file system call interface using Copy On Write semantics. The main focus is on the higher interface of the Virtual File System to user processes. A minimal transaction oriented system call interface is proposed, to be used as the Virtual File System of a research operating system.

Lyngby, October 2010

Sakis Kasampalis

Acknowledgements

I want to thank my supervisor Sven Karlsson for his help and support regarding many issues related with this thesis. Only a few to mention are finding a relevant topic which fits to my current skills, suggesting research papers and technical books, giving useful feedback, and providing technical assistance.

I also thank my friend Stavros Passas, currently a PhD student at DTU Informatics, for the research papers he provided me concerning operating system reliability and performance, and transactional memory. Among other things, Stavros also helped me with writing the thesis proposal and solving several technical problems.

Special thanks to Andreas Hindborg and Kristoffer Andersen, the BSc students who have started working on FenixOS for the needs of their thesis before me. Andreas and Kristoffer helped me to get familiar with the development tools of FenixOS very fast.

Last but not least, I would like to thank Per Friis, one of the department's system administrators, for providing me the necessary hardware required to do the file system performance analysis.

Contents

Chapter 1

Introduction

The motivation behind this thesis is two folded. The first part of the thesis focuses on testing the performance and scalability of two modern Copy On Write (COW) based file systems: the Zettabyte File System (ZFS) and the B-tree file system or "Butter-eff-ess" (Btrfs). COW based file systems can solve the problems that journaling file systems are facing efficiently. Journaling file systems (1) cannot protect both data and metadata, (2) cannot satisfy the large scaling needs of modern storage systems, (3) are hard to administrate. Recent file systems use COW to address the problems of fault tolerance, scalability, and easy administration. The main reason for doing the performance analysis of ZFS and Btrfs is that we are investigating which file system is the best option to use in our UNIX-like research operating system called FenixOS. We are aware that there are more COW based file systems but we have decided to choose one of ZFS and Btrfs for several reasons. ZFS is already stable, has a very innovative architecture, and introduces many interesting features that no other file system ever had. Btrfs on the other hand is under heavy development, targets on simplicity and performance, and is supported by the large GNU/Linux community.

The second part of the thesis deals with the design of a transaction oriented file system call interface for FenixOS. A transaction is a sequence of operations described by four properties: Atomicity, Isolation, Consistency, and Durability (ACID). Transactions are essential for using concurrent programming models

efficiently. Just a few examples of using concurrent programming based on transactions are for eliminating security vulnerabilities, for rolling back unsuccessful installations, and to do file system recovery. Moreover, only concurrent programming models can take full advantage of today's multi-core and multiple processor architectures. Most UNIX-like operating systems are compatible with the Portable Operating System Interface (POSIX). Unfortunately, POSIX does not offer ACID support through transactions. We believe that transactions should be exposed to programmers as a first class service, using an intuitive Application Programming Interface (API). To achieve that, a modern non POSIX transaction oriented interface is proposed.

1.1 Problem statement

In a sentence, the problem statement of the thesis is to (1) identify the strengths and weaknesses of ZFS and Btrfs and propose one of them as the best choice for the needs of FenixOS, (2) define a transaction oriented file system call interface for FenixOS, and document the design decisions related with it.

1.2 Contributions

The contributions of this thesis include (1) an evaluation of the performance and scalability of ZFS and Btrfs under several workloads and different scenarios using recommended benchmarks, (2) the definition of the file system call interface of FenixOS and a discussion about the design decisions related with it.

1.3 Major results and conclusions

The performance analysis results of the thesis show that at the moment ZFS seems to be the right choice for FenixOS. The main reasons are that (1) ZFS uses full ACID semantics to offer data and metadata consistency while Btrfs uses a more relaxed approach which violates transactional semantics, (2) ZFS can scale more than Btrfs.

Even if Btrfs outperforms ZFS on simple system configurations, the lack of supporting ACID transactions and its poor scaling properties make currently ZFS the preferred file system for FenixOS.

Since I have not implemented the file system call interface, I cannot make conclusions about its performance. What I can do instead is an evaluation of the design decisions that I have taken against the design of other transaction oriented interfaces.

To begin, using COW as a basis for supporting transactions instead of other approaches, like journaling and logging, has several advantages. Journaling file systems can only protect metadata at an acceptable performance. This happens because they need to write all file system operations twice: Once to the journal, and once to the file system. Logging file systems have the same performance issue and are more complex because they need to save all operations on a separate log partition which is located on the hard disk. When COW is used, full ACID semantics provide both data and metadata consistency at an acceptable overhead. Per-block checksumming is used to provide both data and metadata integrity without slowing down the file system extremely. A negative aspect of using COW is that the length of a transaction is limited, because the transaction must fit into the main memory. While journaling faces the same problem as COW regarding the length of a transaction, logging supports long transactions because the transaction is kept on-disk. Researchers argue that the transaction length problem of COW can be solved using techniques like swapping the uncommitted transaction state to the hard disk, but as far as I know there are no available implementations yet.

Cheap and instant snapshots and clones offer online backups and testing environments to the users. When COW is not used, snapshots and clones are expensive because non COW based file systems overwrite data in place. That is why they are usually not supported in file systems which do not use COW.

We use system transactions to protect the operating system state. Protecting the data structures of a specific application is not where we want to focus on. System transactions (1) do not require any special hardware/software, (2) can be isolated and rolled back without violating transactional semantics, (3) perform better than user space solutions, (4) can be optionally integrated with Hardware Transactional Memory (HTM) or Software Transactional Memory (STM) to provide full protection of an application's state.

Exposed transactions work as a facility for supporting concurrent programming. Transactions are exposed to programmers using an intuitive API with the minimum overhead: All the code wrapped inside two simple system calls, `begin` and `end`, is protected by ACID. Logging approaches have a much more complex API and require from the programmer to have a deep understanding of their architecture. Journaling file systems cannot protect data, which means that the usefulness of a transaction is limited.

A strong atomicity model is used to allow the serialisation between transactional and non-transactional activities. A strong atomicity model is more flexible than a weak atomicity model, because it allows transactional and non-transactional activities to access the same resources, ensuring that system updates are consistent. Most transaction oriented interfaces are not strict about the atomicity model which they support. However, it is important to decide about the atomicity model which will be used and notify the programmers, since directly converting lock based programs to transaction based is generally not safe.

Network File System (NFS) Access Control Lists (ACLs) are used as the only file permission model. NFS ACLs are (1) standard, in opposite to "POSIX" ACLs, (2) more fine-grained than traditional UNIX mode style file attributes and "POSIX" ACLs. Furthermore, FenixOS does not need to be compatible with "POSIX" ACLs or traditional UNIX file attributes, thus there are no backward compatibility, performance, and semantic issues which occur when conversions from one format to another and synchronisations are required. A disadvantage of using only NFS ACLs is that they are more complex compared with traditional UNIX mode style file attributes. Users who are not familiar with ACLs will need some time to get used at creating and modifying NFS ACLs. However, users who are already familiar with "POSIX" ACLs will not have any real problems.

Memory mapped resources are a good replacement of the legacy POSIX `read` and `write` system calls since they (1) are optimised when used with a unified cache manager, a feature planned to be supported in FenixOS, (2) can be used in a shared memory model for reducing Interprocess Communication (IPC) costs.

Since our interface is not POSIX compatible (1) it will take some time for POSIX programmers to get used to it, (2) POSIX code becomes inapplicable for FenixOS. In our defence, we believe that the advantages of exposing transactions to programmers using a new interface are more than the disadvantages. File system call interfaces which add transaction support without modifying POSIX end up being slow (user space solutions) or limited: transactions are not exposed to programmers. Interfaces which modify POSIX to support transactions end up being complex to be used (logging approaches) or complex to be developed: transaction support is added to existing complex non transaction oriented kernel code. We believe that it is better to design a transaction oriented system call interface from scratch, instead of modifying an existing non transaction oriented system call interface. To conclude, supporting POSIX in FenixOS is not a problem. If we decide that it is necessary to support POSIX, we can create an emulation layer like many other operating systems have already done. Examples include the ZFS POSIX Layer (ZPL), the ANSI[1] POSIX Environment (APE),

[1] American National Standards Institute

and Interix.

1.4 Outline of the thesis

In this chapter - "Introduction" I have discussed about the motivation behind this thesis, my contributions, and the major conclusions.

Chapter 2 - "Background" introduces you to the concepts and the technical topics related with this thesis.

Chapter 3 - "Copy On Write Based File Systems" describes the problems that current file systems have, and how they can be solved using Copy On Write.

Chapter 4 - "Overview, performance analysis, and scalability test of ZFS and Btrfs" focuses on the ZFS and Btrfs file systems: Their features and design, how well they perform, and how much they can scale.

Chapter 5 - "A file system call interface for FenixOS" includes a discussion about the usefulness of transactions in operating systems and file systems, and presents my contributions to the design of the file system call interface of FenixOS.

Chapter 6 - "Development process, time plan, and risk analysis" is a description of the development process used during this thesis, the completed tasks, and the risks involved.

Chapter 7 - "Related work" presents the work of other people regarding the performance of ZFS and Btrfs and the design of transactional system call interfaces, and discusses how they relate with this work.

Finally, chapter 8 - "Conclusions" summarises the contributions of this thesis and proposes future work.

Chapter 2

Background

This chapter provides definitions for all the important technical terms which will be extensively used during my thesis: operating system, kernel, file system, virtual file system, and more. Apart from that, it (1) describes the most common ways modern operating systems can be structured, (2) introduces you to the research operating system FenixOS, and (3) includes references to several well-known and less known operating systems and file systems.

2.1 Operating systems

It is not easy to provide a single clear definition about what an operating system is. Personally, I like the two roles described in [120, chap. 1]. One role of the software layer which is called the operating system is to manage the hardware resources of a machine efficiently. By machine I mean a personal computer, a mobile phone, a portable media player, a car, a television, etc. Examples of hardware resources are processors, hard disks, printers, etc. The second role of the operating system is to hide the complexity of the hardware by providing simple abstractions to the users.

An example of a key abstraction found in most operating systems is the file. Users create files for saving and restoring data permanently in a uniform way,

which is independent of the device the data are saved. They do not need to know the details about how Universal Serial Bus (USB) sticks, hard disks, or magnetic tapes work. The only fundamental operations users ought to learn are how to create, edit, and save files using their favourite text editor. The same operations apply for all devices, magnetic or electronic, local or remote.

Figure 2.1 shows the placement of the operating system in relation to the rest layers of a personal computer. The operating system is on top of the hardware, managing the resources and providing its abstractions to the applications. Users interact with the operating system through those abstractions, using their favourite applications.

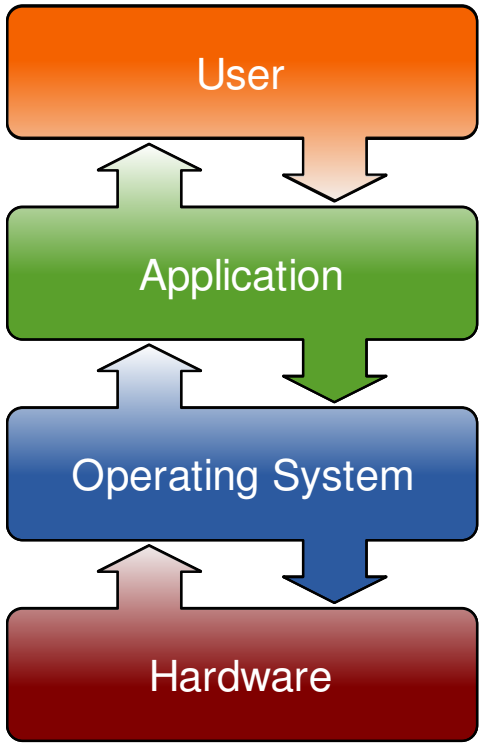

Figure 2.1: The placement of the operating system. The operating system is placed between the hardware and the application layers. It manages the hardware resources efficiently, and provides intuitive abstractions to the user applications

For most operating systems, a service can run either in kernel/supervisor mode/space or in user mode/space. The services which run in kernel mode have

full control of the hardware, which means that a potential bug can cause severe damage to the whole system: A system crash, critical data corruption, etc. The services which run in user mode have restricted access to the system resources, which makes them less dangerous [121]. For example, if the printer driver crashes while running in user mode, the worst case will be the printer to malfunction. If the same thing happens while the printer driver is running in kernel mode, the results can be much more harmful.

Not all operating systems need to support both kernel and user mode. Personal computer operating systems, such as GNU/Linux [116, chap. 5], Windows, and Mac OS X offer both modes. Embedded operating systems on the other hand, may not have kernel mode. Examples of embedded operating systems are QNX, eCos, and Symbian OS. Interpreted systems, such as Java based smart card operating systems use software interpretation to separate the two modes. When Java is used for instance, the separation happens through the Java Virtual Machine (JVM) [120, chap. 1]. Examples of interpreted systems are Java Card and MULTOS.

2.1.1 Operating system structure

The heart of an operating system is its kernel. For operating systems which support two modes, everything inside the kernel is running in supervisor mode, while everything outside it in user mode. Examples of privileged operations which run in supervisor mode are process creation and Input/Output (I/O) operations (writing in a file, asking from the printer to print a document, etc.). Kernels can be very big or extremely small in size, depending on the services which run inside and outside them, respectively. Big is in terms of million lines of code, while small is in terms of thousand lines of code. Up to date, the two most popular commercial operating system structures follow either a monolithic or a microkernel approach [120, chap. 1].

In pure monolithic kernels all system services run in kernel mode. The system is consisted of hundreds or thousands procedures, which can call each other without any restriction [122, chap. 1]. The main advantage of monolithic systems is their high performance. Their disadvantages are that they are insecure, they have poor fault isolation, and low reliability [52, 118, 121]. A typical monolithic organisation is shown in Figure 2.2. Without getting into details about which is the role of each service, we can see that all services run in kernel mode. There is no isolation between the services, which means that a bug in one service can cause troubles not only to itself, but to the whole system. A scenario where a service faces a crash and as a result overwrites important data structures of another service can be disastrous. Examples of monolithic kernels are Linux,

FreeBSD, and OpenSolaris.

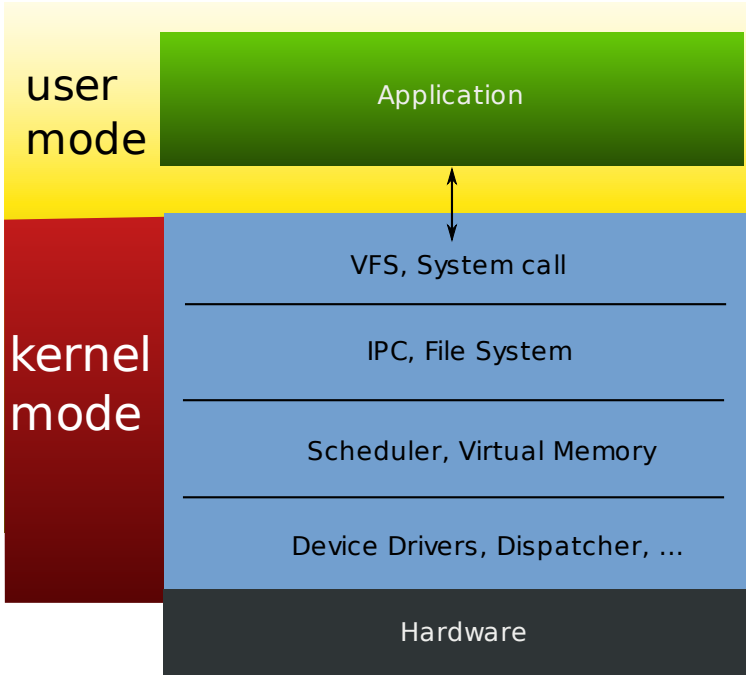

Figure 2.2: A typical monolithic design. On top of the hardware is the operating system. All services (file system, virtual memory, device drivers, etc.) run in kernel mode. Applications run in user mode and execute system calls for switching to kernel mode when they want to perform a privileged operation

Microkernels follow the philosophy that only a small number of services should execute in kernel mode. All the rest are moved to user space. Moving services to user space does not decrease the actual number of service bugs, but it makes them less powerful. Every user space service is fully isolated from the rest, and can access kernel data structures only by issuing special kernel/system calls. The system calls are checked for validity before they are executed. Figure 2.3 demonstrates an example of a microkernel organisation. In the organisation shown in Figure 2.3, only a minimal number of services runs in kernel space. All the rest have been moved to user space. When following a microkernel structure, it is up to the designer to decide which parts will live inside the kernel

and which will be moved outside. Of course, moving services to user space and providing full isolation between them does not come without costs. The way the various user mode services can communicate with each other and the kernel is called Interprocess Communication (IPC). While in monolithic systems IPC is easy and cheap, in microkernels it gets more complex and is more expensive. The extra overhead that IPC introduces in microkernel based systems is their biggest disadvantage. The benefits of microkernels are their enhanced reliability and fault isolation [52, 118, 121]. Examples of microkernel based systems are MINIX 3, Coyotos, and Singularity.

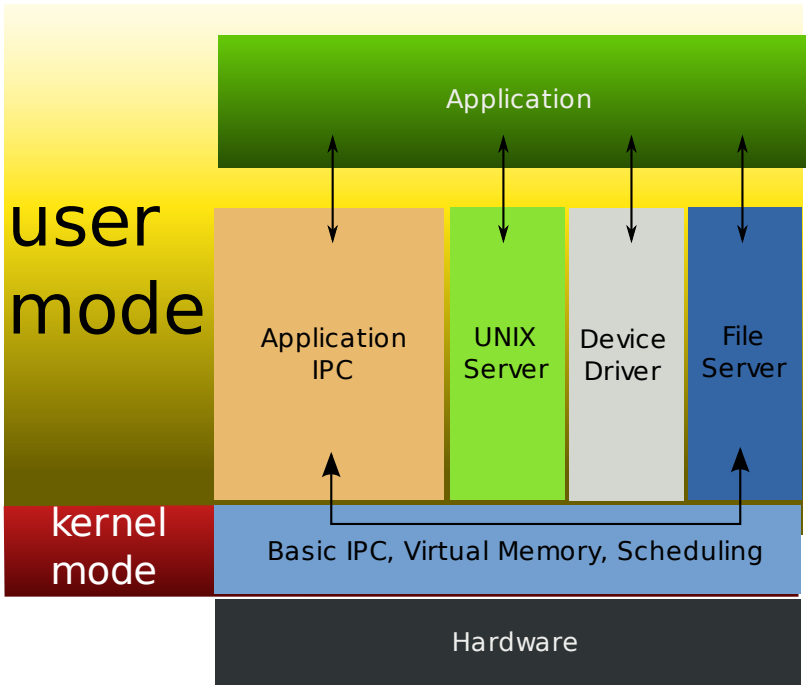

Figure 2.3: A microkernel design. A minimal number of services (basic IPC, virtual memory, and scheduling) runs in kernel mode. All the rest services (file server, device drivers, etc.) execute in user mode. Applications execute services using IPC. The services communicate with each other and the kernel also using IPC

There is a long debate which holds for years about whether modern operating systems should follow a monolithic or a microkernel approach [119]. Researchers

have also proposed many alternative structures [118, 121]. Personally, I like Tanenbaum's idea about creating self-healing systems. And I would not mind to sacrifice a small performance percentage for the sake of reliability and security. However, I am not sure if the performance loss of modern microkernels is acceptable. The developers of MINIX 3 argue that the extra overhead of their system compared to a monolithic system is less than 10%. Yet, MINIX 3 does not currently support multiple processors. Since modern personal computer architectures are based on multi-core processors, operating systems must add support for multiple processors before we can make any conclusions about their performance.

2.1.2 FenixOS

When a program is running in user mode and requires some service from the operating system, it must execute a system call. Through the system call, the program switches from user mode to kernel mode. Only in kernel mode the program is permitted to access the required operating system service [120, chap. 1].

Back in the 70s and the 80s, when the code of UNIX became publicly available, many different distributions (also called flavours) started to appear. Some examples are BSD, SunOS, and SCO UNIX. Unfortunately, the incompatibilities between the different UNIX flavours made it impossible for a programmer to write a single program that could run on all of them without problems. To solve this issue the Institute of Electrical and Electronics Engineers (IEEE) defined POSIX. POSIX is a minimal system call interface that all UNIX-like systems should support, as long as they want to minimise the incompatibilities between them. The systems which support the POSIX interface are called POSIX compatible, POSIX compliant, or POSIX conformant (all names apply) [122].

FenixOS is a research operating system under active development. Most FenixOS developers are students at the Technical University of Denmark (DTU). Being a research project, FenixOS allows us to experiment with things that commercial operating systems cannot. For example, even if FenixOS is UNIX-like, it is not necessarily POSIX compliant. POSIX was something which was defined many years ago to solve a very specific problem, but we believe that the time has come to define a new system call interface, which fits better with modern operating systems. And we are not the only ones who argue on this. Even Plan 9, which is a system that many of the original UNIX developers are involved with, is not POSIX compliant. It only uses an emulation environment called the ANSI/POSIX Environment (APE) for running POSIX commands. But that is all. There are no other relations between Plan 9 and POSIX [95].

Even though the kernel of FenixOS follows a monolithic structure, there is a big difference between our design philosophy and pure monolithic systems. Like other researchers [52, 53, 54, 118, 121], we believe that moving all the buggy components to user space and isolating them can increase the fault tolerance, reliability, and stability of a system. By buggy components, I mean those which include most of the programming bugs and cause the biggest troubles in commercial operating systems. An example of such a component is the device driver. A device driver is the software part which makes possible the communication between the operating system and a peripheral device [120, chap. 1]. If you have ever used Windows 98, you were probably annoyed by the fact that you needed to manually install a device driver for most of your computer's devices: the modem, the sound card, the graphics adaptor, etc. It is also likely that you have experienced small or serious problems at least once during your Windows 98 adventures. A small problem is an unrecognised or not properly installed device driver. A serious problem is ending up with an unstable operating system which crashes or reboots unexpectedly after the "proper" (according to the operating system) installation of a device driver.

An example of the FenixOS structure is shown in Figure 2.4. Buggy components like device drivers run in user mode. Applications never communicate directly with the user mode services. They execute system calls to make use of a system service. The kernel decides whether a user space service should be consulted or not. Moving components like device drivers in user space allows us to use services like driver wrappers. A device driver wrapper can be used to monitor the activity of a driver and check whether it is privileged to do a particular job or not. Some people use the term hybrid kernel to describe structures similar to the one of FenixOS [92].

To conclude this section, I should mention that apart from fault tolerance, reliability, and stability, FenixOS also focuses on scalability. As a proof, FenixOS already supports multiple processors. And we are not investigating COW based file systems with no reason. It is important that scalability is one of the main features that COW based file systems are focusing.

2.2 File systems

In a previous section I have introduced one of the most important abstractions of an operating system: the file. There are many key topics related with files, such as protection, structure, management, organisation, and implementation. This fact leads to the creation of a separate operating system software layer which is responsible for dealing exclusively with files: The file management system,

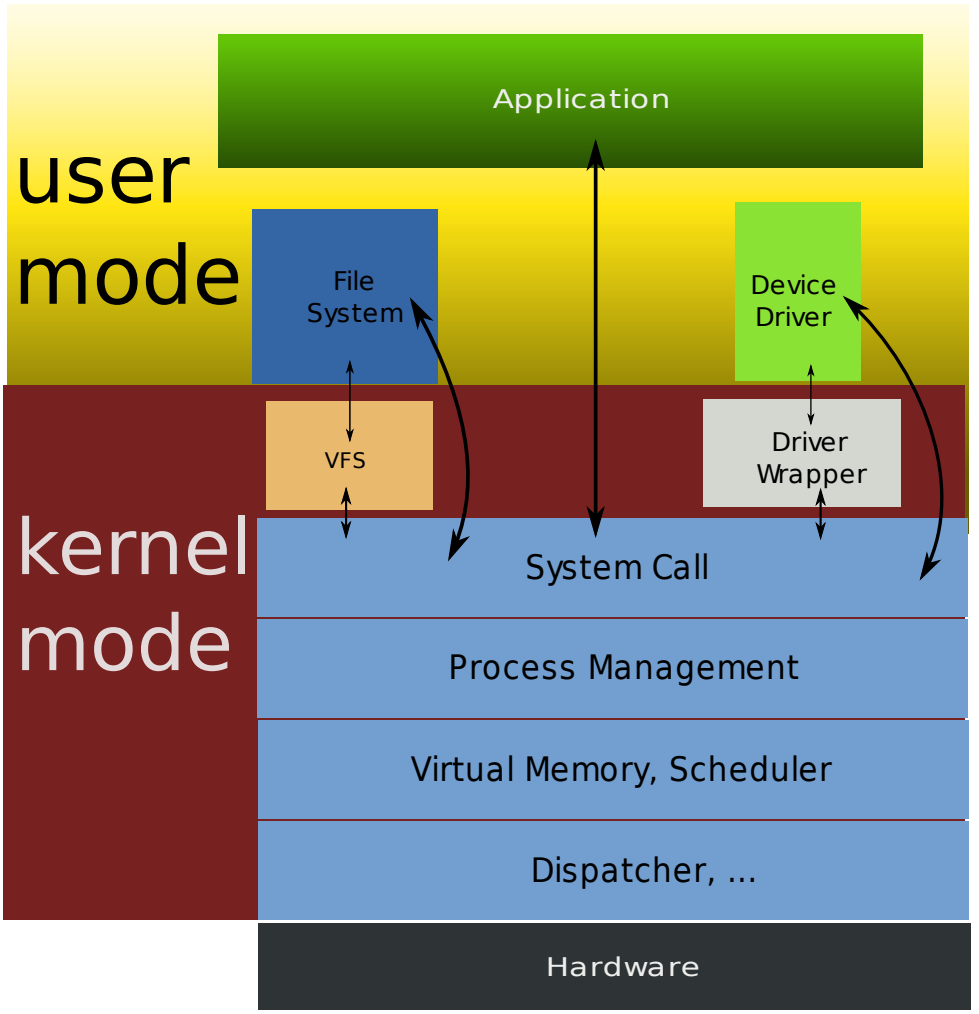

Figure 2.4: The structure of FenixOS. Buggy components like device drivers run in user space. Applications, which also run in user space never contact directly with those components. They execute system calls for asking a service from the kernel. Services like driver wrappers allow the system to monitor and restrict the activities of the device drivers

or simply file system [115, chap. 12], [120, chap. 4]. Examples of common file systems are the New Technology File System (NTFS), the fourth extended file system (ext4), the File Allocation Table (FAT32), and the UNIX File System (UFS).

An operating system usually supports many file systems. This is required because users can share data only if the operating systems that they use support a common file system. For instance, it is not unusual for a GNU/Linux system to use ext4 as its primary file system, but also support FAT32. In this way it allows GNU/Linux and Windows users to exchange files.

2.2.1 The Virtual File System layer

The need of an operating system to support multiple file systems has lead to the concept of the Virtual node (Vnode) layer [63] or Virtual File System (VFS) layer. The VFS is a software layer which provides a single, uniform interface to multiple file systems. The main idea of the VFS is to abstract the common functionality of all file systems on a high level, and leave the specific implementation to the actual file systems. This means that when file system developers wish to add a new file system to an operating system which uses a VFS, they must implement all the functions (more precisely system calls) required by the VFS [22, chap. 12], [47, chap. 10], [115, chap. 12], [120, chap. 4].

File systems in FenixOS will run in user space, while the VFS will run in kernel space. This is demonstrated in Figure 2.4. The VFS will be the mediator between an actual file system and the kernel. A file system which runs in user space, must execute a system call through the VFS for switching to kernel mode and using an operating system service. A typical example is when a program needs to write some data in a file. All I/O operations, including writing in a file are privileged. This means that they are only allowed in kernel mode. As we will see in chapter "A file system call interface for FenixOS", in FenixOS the user space file system (for example FAT32) must execute the `memoryMap` system call to request write access in a file. Note that executing the `memoryMap` system call does not necessarily mean that write access will be granted. The VFS needs to check if the requested file exists, if the user which executes the system call is allowed to write in the requested file (according to the file's permissions), etc.

Chapter 3

Copy On Write based file systems

In the beginning of this chapter I explain why it is not uncommon for modern hard disks to fail pretty often. I then continue with describing why a better solution than journaling file systems is required for solving the problems of data consistency and integrity. Finally, I conclude with an introduction to Copy On Write: what Copy On Write actually means, how it relates with file systems, and how it can provide data consistency and integrity efficiently.

3.1 The need for data integrity and consistency

File system information can be divided into two parts: data and metadata. Data are the actual blocks, records, or any other logical grouping the file system uses, that make up a file. Metadata are pieces of information which describe a file, but are not part of it. If we take an audio file as an example, the music result which can be listened when the file is played using a media player is its data. All the rest information like the last time the file was accessed, the name of the file, and who can access the file, are the file's metadata [47, chap. 2].

Even if modern hard disk capacity is rapidly increasing and is considered rela-

tively cheap, there is one problematic factor which still remains constant: the disk Bit Error Rate (BER). In simple terms, BER is the number of faulty (altered) bits during a data transmission, because of noise, interference, distortion, etc., divided by the total number of bits transmitted [25]. Modern disk manufacturers advertise that there is one uncorrectable error every 10 to 20 terabytes [117].

Both data and metadata are of great importance. On metadata corruption, the file system and therefore the user is completely unable to access the corrupted file. On data corruption, although the user can access the metadata of the file, the most important part of it becomes permanently unavailable. Thus, it is important for modern file systems to provide both data and metadata integrity.

Ensuring that data and metadata have not been altered because of BER solves a number of problems. But there is still one serious problem which needs to be solved: consistent system updates. The most common file systems (and operating systems in general) do not provide to programmers and system administrators a simple way of ensuring that a set of operations will either succeed or fail as a single unit, so that the system will end up in a consistent state. A simple example of an operation which requires this guarantee is adding a new user in a UNIX-like system. In UNIX-like systems, adding a new user requires a couple of files to be modified. Usually modifying two files, `/etc/passwd` and `/etc/shadow` is enough. Now, imagine a scenario where the utility which is responsible for adding a new system user (for example `useradd`) has successfully modified `/etc/passwd` and is ready to begin modifying `/etc/shadow`. Suddenly, the operating system crashes because of a bug or shuts down due to a power failure. The system ends up in an inconsistent state: `/etc/passwd` contains the entry for the new user but `/etc/shadow` is not aware of this modification. In such cases, it makes more sense to ignore the partial modifications and do nothing, instead of applying them and ending up with an inconsistent system. Ignoring the partial modifications is not a bad thing. It keeps the system consistent.

3.2 Why journaling is not enough

Now that we have discussed why it is (1) normal for modern hard disks to fail, (2) important for file systems to enforce both data and metadata integrity and consistency, let's identify what kind of problems do the most popular file systems have.

Most of the file systems which are currently used in operating systems work

by using a technique called journaling. A journal is a special log file in which the file system writes all its actions, before actually performing them. If the operating system crashes while the file system is performing an action, the file system can complete the pending action(s) upon the next system boot, by simply investigating the journal. The family of the file systems which use a journal are called journaling file systems. Examples of popular journaling file systems include NTFS, ext4, XFS, and ReiserFS [120, chap. 4].

Journaling file systems have a big problem: they can only provide metadata integrity and consistency. Keeping both data and metadata changes inside the journal introduces an unacceptable performance overhead, which forces all journaling file systems to log only metadata changes. Since writing the changes twice - once to the journal and once in real - is extremely slow, a better solution is required [47, chap. 7], [72]. The recommended solution must provide both metadata and data integrity and consistency in a cheap way.

3.3 Copy On Write

The most recent solution regarding file system (meta)data consistency and integrity is called Copy On Write (COW). Before getting into details about what COW (sometimes called shadowing [102]) is in respect to file systems, I will give some background information about what COW is in general, and discuss some applications of it.

COW generally follows a simple principle. As long as multiple programs need read only access to a data structure, providing them only pointers [60, chap. 5] which point to the same data structure, instead of copying the whole structure to a new one, is enough. If at least one of the programs needs at some point write access to the data structure, create a private copy for it. The same holds for each one of the programs which will need write access to the data structure. A new private copy will be created for them. Let's assume that we have an array data structure [60, chap. 1] called "foo", and two programs called "Bob" and "Alice" which need to access it in read/write mode. If during the access, none of the programs tries to change the contents of "foo", both "Bob" and "Alice" will actually use exactly the same "foo" array. If at some point "Bob" changes the contents of "foo", a private copy of the changed data of "foo" will automatically be created by the COW system and provided to "Bob". Note that the unchanged data can still be shared between "Bob" and "Alice". This is a powerful feature of COW [93, 102].

If you do not understand some of the terms that are mentioned in the next

paragraph, do not worry. They are given just for completeness. Only one of them is important for the needs of my thesis: snapshots. A snapshot is a consistent image of the data at a specific point in time [2, 55]. By data I mean the contents of a file system, the contents of a database, etc. Snapshots can be read only, or read/write. Writable snapshots are also called clones [102]. Snapshots are extremely useful and have many applications: data recovery, online data protection, undo file system operations, testing new installations and configurations, backup solutions, data mining, and more [72, chap. 1], [102].

COW is used in many areas of computer science. Only a few to mention are (1) optimising process creation in operating systems [79, chap. 5], (2) creating secure, isolated environments, appropriate for running untrusted applications on top of primary file systems [57], (3) designing systems which are based on delimited continuations and support advanced features, such as multitasking and transactions [61], (4) supporting efficient snapshots in database systems [105], (5) improving snapshots to support both undo and redo operations in storage systems [131], and (6) combining snapshots with pipelining for more efficient snapshot creation [127].

3.3.1 Copy On Write and file systems

Now that I have introduced COW to you, I should underline that optimisation is not the main reason for using COW in file systems. I believe that COW's major contributions to file systems are the support for (1) taking cheap snapshots, (2) ensuring both data and metadata consistency and integrity with acceptable performance [72, chap. 1].

Snapshots can be implemented in many ways. An example is through the use of Logical Volume Management (LVM). LVM is a virtualisation technique which supports abstracting many physical disks and partitions as a single virtual disk. LVM makes the life of system administrators easier by supporting snapshots, extending and shrinking file systems dynamically without the need to bring them offline, etc. [62]. Unfortunately, LVM has a big disadvantage regarding snapshots. LVM snapshots (1) have poor performance because every block needs to be written twice, once to the primary and once to the backup logical volume, and (2) add a space penalty because the disk area which is allocated by LVM is reserved and cannot be used by the file system [72, chap. 2], [86].

When using a COW based file system, offering snapshot support is straightforward. COW in file systems means a very simple but at the same time very important thing: do never overwrite any (meta)data [83, chap. 1]. All (meta)data

modifications are written to a new free place on the disk. The old (meta)data cannot be freed until all the changes complete successfully. Since data are never overwritten, it is easy to take a snapshot. You just have to say to the file system "do not to free the old space, even after the successful completion of the changes". Give a name to the old space, and your snapshot is ready. That is what I mean when talking about cheap snapshots.

Consistency in COW based file systems is guaranteed using transactional semantics [83, chap. 1]. A transaction is a sequence of operations which has four properties: Atomicity, Consistency, Isolation, and Durability (ACID). Regarding file systems, (1) Atomicity means that a transaction either succeeds or fails atomically. If it succeeds all the file system operations which belong to it are completed successfully. If it fails or the user aborts it (provided that abort is supported) none operation takes place. Whenever a transaction fails, the result should be the same as it would be if the transaction never happened. (2) Consistency means that partial file system updates are not allowed. Even if the operating system crashes while a transaction is in progress, the file system will remain in the most recent consistent state it was before the beginning of the transaction. (3) Isolation means that even if multiple transactions are happening simultaneously, a transaction in progress behaves as it is the only one which is active. Each transaction has a clean and consistent view of the file system, and file system updates of the rest transactions in progress are not visible to it. (4) Durability means that once a transaction is committed, all file system changes are permanent even if the operating system fails right after the commit [8], [91, chap. 1].

Transactions have been supported in databases for a long time, but the same is not true for file systems and operating systems [96, 114, 130]. Some journaling file systems support transactions but the problem still holds: they can only protect metadata. Using COW and transactional semantics, file systems can provide full data consistency.

Integrity is a different thing. Even if a file system is consistent, there is nothing which can ensure that the contents of a specific file have not been altered because of BER, a malicious program, human intervention, etc. File systems support data integrity using checksums. A checksum is a calculated (usually alphanumeric) identifier of a data portion [129]. There are many checksum algorithms, but all follow the same principle. In file systems, the idea is to produce a unique checksum depending on the contents of a file. Even if a single bit of the file changes, for example a new character is added to a text file, or a bit in the file permissions is activated, the produced checksum will be different. In this way the file system can detect that the contents of a file have been altered. Let me demonstrate checksumming with a simple Example: 3.1. First I will type the `md5sum` command to get the checksum of this LaTeX file using the Message-

Digest algorithm 5 (MD5), and then I will repeat the same process after adding a single character to the file's contents.

Example 3.1 Checking the integrity of a file using MD5

```
$ md5sum Chapter2.tex
8a8f444823633774b9943611e3a9e0e3  Chapter2.tex
$ md5sum Chapter2.tex
9158f26d177c517d459b1151c212999f  Chapter2.tex
```

Even with a so simple change, the MD5 checksum is completely different. This indicates that the file has changed. Note that `md5sum` works only for data, not for metadata, thus changing the metadata of the file - for example its access permissions using `chmod` - will produce exactly the same checksum. File systems which support (meta)data integrity do it on a per-block level, or even better [83, chap. 1]. In this way, they can provide full data integrity.

Not all COW based file systems support (meta)data integrity and consistency. On the next chapter I am focusing on ZFS [51] and Btrfs [9]. ZFS and Btrfs support (meta)data integrity. ZFS also supports (meta)data consistency, while Btrfs is at the moment more relaxed regarding consistency. Other file systems based on COW are LLFS [72], ext3cow [93], Write Anywhere File Layout (WAFL) [55], and Hammer [43].

Chapter 4

Overview, performance analysis, and scalability test of ZFS and Btrfs

In this chapter I begin with an overview of two modern COW based file systems: ZFS and Btrfs. The chapter continues with a performance analysis between the two file systems using recommended benchmarks. At the end of this chapter I am running a small scalability test to see how well can ZFS and Btrfs scale against large multithreaded workloads.

4.1 ZFS and Btrfs overview

ZFS and Btrfs are the latest efforts of the free software [116, chap. 3] community aiming at creating a scalable, fault tolerant, and easy to administrate file system. In the previous chapter I have focused only on fault tolerance (consistency and integrity) because I consider it the most important design goal. However, when the file system designers are designing a new file system, they are trying to solve as many problems as possible. Two additional problems with current file systems are (1) their inability to satisfy the large storage requirements of today's data centres [27], (2) the lack of a simplified administration model [83, chap. 1].

ZFS and Btrfs are very similar regarding the features that they already support or plan to support, but after taking a deeper look one can find many differences in their design, development model, source code license, etc. In the following sections I will try to take a brief tour on ZFS and Btrfs.

4.1.1 Common features

To begin, ZFS and Btrfs are based on COW. They use COW semantics for providing support for snapshots and clones. ZFS and Btrfs use checksums which are 256 bits long on both metadata and data to offer integrity, and support multiple checksumming algorithms. To address the scaling requirements of modern systems, ZFS and Btrfs are designed based on an 128 bit architecture. This means that block addresses are 128 bits long, which adds support for extremely large files, in terms of million terabytes (TB) - 10^{12} [64], or even quadrillion zettabytes (ZB) - 10^{21} [83, chap. 1]. Both file systems have multiple device support integrated through the usage of software Redundant Array Of Inexpensive Disks (RAID). RAID is a way of distributing data over multiple hard disks, for achieving better performance and redundancy. Multiple RAID algorithms exist, and most of them are supported by ZFS and Btrfs. Finally, both file systems support data compression for increasing throughput [27, 51].

4.1.2 ZFS discussion

ZFS is the first file system which introduces many unique features that no other file system ever had. Some of them are (1) its pooled storage, (2) its self-healing properties, and (3) the usage of variable block sizes. These unique features are enough to make ZFS one of the most interesting file systems around. Let me say a few words about each feature.

Pooled storage is used to eliminate the disadvantages of LVM. A storage pool is a collection of physical devices. Many file systems can be created inside a pool. In fact, the ZFS developers recommend using one file system per user [83, chap. 2], for organisational purposes. Devices can be added/removed from the pool dynamically, and in this case the space also becomes available/unavailable for all file systems [51], [83, chap. 1], [84].

Self-healing in ZFS is done using a RAID technique called RAID-Z. RAID-Z is nothing more than using RAID-5 combined with COW and transactional semantics. What self-healing practically means is that whenever ZFS detects that there is (meta)data corruption on a block of a disk because of a bad checksum,

it can correct it using the "right" block (the one with the expected checksum) from a redundant disk [17, 18].

Most file systems use a fixed block size. ZFS supports variable block sizes, using the concept of metaslabs ranging from 512 bytes to 1 megabyte (MB) [21]. The main idea of using multiple block sizes is that since files usually have a small size initially, a small block size will work optimally for them. But as the size of a file gets increased, its block size can also be increased, to continue working optimally based on the new file size [51].

There are even more things that make ZFS different. First of all, instead of following the typical design of most POSIX file systems which use index-nodes (i-nodes) as their primary data structure [120, chap. 4], ZFS organises data into many data structures, each one being responsible for a very specific job: For instance (1) variable sized block pointers are used to describe blocks of data, (2) objects are used to describe an arbitrary piece of storage, for example a plain file, (3) object sets are used to group objects of a specific type, for example group file objects into a single file system object set, (4) space maps are used to keep track of free space, etc. [19, 84, 132].

Another big difference of ZFS compared to other file systems, including Btrfs which will be described in the next section, is that it is designed to be platform independent. Because of that, it integrates many operating system parts which traditional file systems tend to reuse, since they already exist on the operating system they are being developed for. Examples of such parts are the (1) Input Output (I/O) scheduler, which decides about the priorities of processes and which one to pick from the waiting queue. ZFS uses its own I/O scheduler and a scheduling policy called System Duty Cycle (SDC) [50, 70]. (2) Block cache, which is the part of the main memory used for improving performance by holding data which will hopefully be requested in the near future. ZFS uses its own cache and a modification of the Adaptive Replacement Cache (ARC) algorithm [132]. (3) Transactional semantics. In contrast to Btrfs as we will see below, ZFS uses transactions with full ACID semantics for all file system modifications to guarantee that data are always consistent on disk.

ZFS already supports data deduplication. There is also a beta version of encryption support. Encryption is important for security reasons, while deduplication is important for saving disk space. Regarding compression, ZFS keeps metadata compressed by default. Therefore, enabling compression in ZFS means that data compression is activated. The minimum requirements for using ZFS are at least 128 MB hard disk space, and at least 1 GB of main memory [83, chap. 2]. Some people think that the memory requirements of ZFS are too high [71].

To conclude this section, I would like to mention that ZFS is not a research

effort which might be abandoned like many other file systems in the past, but a successful product: ZFS is already the primary file system of the OpenSolaris operating system. It can also be used optionally as the primary file system in FreeBSD, instead of UFS [42].

4.1.3 Btrfs discussion

Btrfs is not as ambitious as ZFS. The idea behind it is to use simple and well-known data structures, to achieve good scalability and performance - equal or even better than competing GNU/Linux file systems, including XFS, ext3/4, and ReiserFS [31, 77]. I think that the main reason behind the beginning of the Btrfs development was the legal issues which do not allow porting ZFS to GNU/Linux. The Linux kernel is distributed under the General Public License (GPL) version 2, while ZFS is distributed under the Common Development and Distribution License (CDDL) version 1.0. GPL and CDDL are both free software licenses, but they are not compatible. A module covered by CDDL cannot be legally linked with a module covered by GPL [48].

I believe that the biggest advantage of Btrfs over ZFS is the bazaar development model it follows. Like almost all GNU/Linux related projects after the successful development of the Linux kernel, Btrfs is open for contributions by anyone. This model has worked very well in the past, and is likely to work better than the cathedral development model of ZFS, which was created by a single company: former Sun Microsystems, now part of Oracle [100, chap. 2].

Time for a short technical discussion. Instead of using variable sized blocks, Btrfs uses fixed block sizes and groups them into extents. An extent is a group of contiguous data blocks [27, 33]. Using extents has some benefits, like reduced fragmentation and increased performance, since multiple blocks are always written in a continuous manner. Of course, like any logical grouping technique, extents have their own issues. For example, when data compression is enabled, even if you only need to read a part of an extent, you have to read and decompress the entire of it. For this reason, the size of compressed extents in Btrfs is limited.

Everything around Btrfs is built based on a single data structure: the COW friendly b+tree [9]. When using COW with a tree data structure, a modification of a single node propagates up to the root. Traditional b+trees have all their leaves linked. This fact makes b+trees incompatible with COW, because a modification of a single leaf means that COW must be applied on the whole tree. Moreover, applying COW on the whole tree means that all nodes must be exclusively locked and their pointers must be updated to point at the new

places, which results in a very low performance. The COW friendly b+tree introduces some major optimisations on traditional b+trees, to make them COW compatible. To be more specific, the COW friendly b+tree refers to (1) b+trees without linked leaves, to avoid the need of applying COW on the whole tree when a single leaf is modified, (2) the usage of a top-down update procedure instead of a bottom up, which results in applying COW to a minimum number of top nodes, and (3) the usage of lazy reference counts per extent for supporting fast and space-efficient cloning [102].

Btrfs is trying to reuse as many existing components as possible. For this reason, its performance is highly affected by the I/O scheduler of the Linux kernel [50], as well as the system caches: the dentry cache, the page cache, etc. [22, chap. 12]. Btrfs at the moment supports data compression only through zlib, which is used in the Linux kernel as a general compression mechanism. Only data (i-node) compression is supported. Metadata compression is not a planned feature.

Regarding integrity, Btrfs currently supports only Cyclic Redundancy Check (crc32c) [28]. As for transactions, even if Btrfs is said to use transactions, I do not agree with the usage of this term. Btrfs transactions are far from an intuitive way of ensuring consistent system updates. They should be used with great care since they can lead to deadlocks and other kinds of problems. Btrfs transactions (1) are not atomic, (2) can lead to deadlocks or cause other problems (like out of space behaviour) which leave the system inconsistent (3) require from the programmer to have a deep understanding of the file system and operating system internals. Since Btrfs transactions violate ACID semantics (at least atomicity and consistency), they should not be called transactions [30, 41]. It is not clear if Btrfs protects all file system modifications using transactions. Most likely it does not, since they are dangerous. An interesting feature that Btrfs already supports and ZFS does not is dynamic file system shrinking. Dynamic file system shrinking simply means “make my file system smaller without asking me to take it offline” [34]. The recommended requirements for using Btrfs are 1 GB hard disk space and 128 MB main memory [29].

Before closing the Btrfs discussion, I should say that Btrfs is still under heavy development and is not ready to be used as the primary file system for storing your critical data.

4.2 ZFS and Btrfs performance analysis and scalability test

I have decided to do a performance analysis between ZFS and Btrfs, because (1) we are investigating which file system fits better with FenixOS, (2) I found only a single one available [50], and it is always better to get at least a second opinion before making any conclusions.

Before getting into technical details, I should say that in contrast to what most people think, benchmarking is not an easy thing. At least, proper benchmarking [123]. It lasts forever, and it requires to take care of many small but extremely important details [124]. Many people use benchmarks, but most of the results they present are not accurate [125]. I admit that I felt very impressed when I found out that a whole thesis can be devoted in analysing the performance of file systems [38]. I did my best to keep my tests as fair and accurate as possible.

4.2.1 System configuration

I conducted all my performance analysis experiments on a Dell OptiPlex 755 machine. The machine has a 3 GHz Intel Core 2 Duo E8400 CPU, with 64 KB of L1 cache, and 6 MB of L2 cache. The machine contains 2 GB of DDR2 SDRAM. The operating system disk is a 7200 RPM Seagate SATA (ST340014AS) with 40 GB capacity. The benchmark disks are two 7200 RPM Hitachi SATA (HDS722540VLSA80) with 40 GB capacity. The SATA controller is an Intel 82Q35 Express Serial KT.

The scalability test experiments have been conducted on a virtual machine configured with VirtualBox 3.1.8. We are aware that the results of a virtual machine are not as accurate as the results of experiments executed on real hardware [125]. However, even if a public machine with sufficient hardware resources is available (four 2.93 GHZ Intel Xeon X5570 CPUs and 12 GB of RAM), I do not have the privileges to add a benchmark disk for running my experiments. The best I can do is to configure a virtual machine with two virtual disks on the available public machine. The virtual machine has 16 CPUs with the Physical Address Extension (PAE/NX) option enabled. The motherboard has the Input Output APIC (IO APIC) option enabled, and the Extended Firmware Interface (EFI) option disabled. From the acceleration options, VT-x/AMD-V and Nested Paging are enabled. The virtual machine contains 4 GB of RAM to avoid excessive caching. The operating system is installed on a virtual IDE disk with 16 GB capacity. The IDE controller type is PIIX4. The benchmark virtual disk is a

SATA with 1 TB capacity. The SATA controller type is AHCI.

The operating system for running the Btrfs performance analysis experiments is Debian GNU/Linux testing (squeeze). The system is running a 2.6.34 kernel, downloaded from the Btrfs unstable Git repository [32] at the 29th of May 2010, and not further customised. For the ZFS performance analysis experiments, the system is OpenSolaris 2009.06 (x86 LiveCD) [90].

For the scalability test the operating systems are the same as the ones used in the performance analysis tests, with the only difference that their 64 bit versions are installed.

4.2.2 Benchmarking procedure

The benchmarking community suggests that at least three different benchmarks should be used when doing a performance analysis, to increase the accuracy of the results. The recommended approach for file systems is to use (1) at least one macro-benchmark, which is good for getting an overall view of a system's performance or (2) at least one trace/workload generator, which is good for exercising a system against a real-world workload, and (3) multiple micro-benchmarks, which are good for highlighting details about the behaviour of specific file system parts [123]. I have decided to use one application emulator for emulating macro-benchmarks, Filebench [107], one trace/workload generator, IOzone [56], and two micro-benchmarks, bonnie-64 [12] and Bonnie++ [14].

The machine is rebooted before the execution of a new benchmark, for example when switching from bonnie-64 to IOzone. It is not rebooted between each execution of the same benchmark, for example between the second and third execution of an IOzone test. This is done to make sure that the system caches are clean every time a new benchmark begins. Every test is executed 11 times. The first result is ignored, because it is used just for warming the caches. All the results are calculated using 95% confidence intervals, based on Student's t-distribution. No ageing is used on the operating systems. All non-essential services and processes are stopped by switching into single user mode before executing a test. This is done using the command `sudo init 1` in Debian, and `svcadm milestone svc:/milestone/single-user:default` in OpenSolaris. I am using a script-based approach to minimise typing mistakes [123].

ZFS and Btrfs are always mounted with the default options. This means that in ZFS metadata are compressed and data are uncompressed. In Btrfs, both data and metadata are uncompressed. The scalability test is done only on a single disk. The performance analysis experiments are conducted both on a single

disk and on a software RAID 1 configuration using two disks (mirroring). I will demonstrate two simple examples of creating a RAID 1 ZFS and Btrfs configuration for completeness. If `c7d0` is the disk which has OpenSolaris installed, `c7d1` is the primary disk, and `c8d1` the secondary, the command to create a RAID 1 ZFS pool called mir is `zpool create mir mirror c7d1 c8d1`. Similarly, if `/dev/sda` is the disk which has GNU/Linux installed, `/dev/sdb` is the primary disk, and `/dev/sdc` the secondary, the command to create a RAID 1 Btrfs file system is `sudo mkfs.btrfs -m raid1 -d raid1 /dev/sdb /dev/sdc`.

4.2.3 Performance analysis results

In this section I present the performance analysis results of the benchmarking frameworks that I have decided to use for my experiments.

4.2.3.1 bonnie-64 performance analysis results

I will begin with the performance analysis results of bonnie-64. bonnie-64 is the 64 bit version of the classic Bonnie micro-benchmark [15], written by the original Bonnie author. Bonnie has become popular because of its simplicity, and the easy to understand results which it produces [24]. For these reasons, no matter how old it is and despite its disadvantages [125] Bonnie is still extensively used in serious developments [37, 125]. I am using bonnie-64 version 2.0.6, downloaded from the Google code repository [13]. Tables 4.1 and 4.2 show the results of bonnie-64 for ZFS and Btrfs, on the single disk configuration.

I have chosen to set the file size ("Size" column) to 12 GB to make sure that the system caches are bypassed. The author of Bonnie suggests to set this number equal to at least four times the size of the main memory [15]. I am setting it to six times the size of the computer's RAM to ensure that the caches will not be used. From all the results displayed in Tables 4.1 and 4.2, the most important are the block sequential input and output. Per character operations are not of great importance since no serious application which makes heavy I/O usage will read/write per single character [76]. Rewrite is also not very important since we know that COW based file systems never write data in place. However, it can still provide an indication about how effective the cache of the file system is [40] (if the cache is not bypassed) and how well a COW file system performs when doing rewrite operations. A last thing to note is that the CPU usage and the random seeks are not precise for multiprocessor (and therefore multi-core) systems [15]. For this reason I will use the CPU usage just as an indication about how busy the CPUs are. The random seek results will

be used as a general indicator of the efficiency of the "random seek and modify" operation relative to the speed of a single CPU. An example of the command that I execute for getting the performance analysis results using bonnie-64 is `Bonnie -s 12288 -d /mnt/btrfs -m "Single Disk" >> bonnie-btrfs-0`.

Table 4.1: Sequential output on the single disk

Single Disk		**Sequential Output**					
File system	Size	Per Char		Block		Rewrite	
	GB	M/sec	%CPU	M/sec	%CPU	M/sec	%CPU
ZFS	12	28-29	10	28	3	17-18	2
Btrfs	12	52	54	52	3-4	21	3

Table 4.2: Sequential input and random seeks on the single disk

Single Disk		**Sequential Input**				**Random Seeks**	
File system	Size	Per Char		Block			
	GB	M/sec	%CPU	M/sec	%CPU	/sec	%CPU
ZFS	12	49-50	23-24	52	3	39-40	0.12
Btrfs	12	50	71-72	52	5	89-95	0.24

What we can generally observe in Tables 4.1 and 4.2 is that Btrfs outperforms ZFS on most tests. More specifically, Btrfs has an output rate of 52 MB/second when writing the 12 GB file per block, while ZFS reports 28 MB/second. This most probably happens because the b+tree extents are optimised for sequential operations and perform better than the ZFS metaslabs [19]. Another reason behind the higher sequential output data rates of Btrfs is the usage of simple and fast block allocation policies. As we will see, while such policies help the Btrfs file system to achieve high sequential output data rates on simple systems (for example servers with a single hard disk), they have a negative impact when using more complex configurations (redundant schemes like RAID 1).

When reading the 12 GB file per block, both file systems have the same input rate, 52 MB/second. This verifies that extents are good for both sequential reads/writes, since the file system can read/write in a contiguous manner. A reason why ZFS has a much better sequential input performance compared to its sequential output might be that it needs to use transactional semantics during file system writes [35, 50]. Btrfs transactions violate ACID and are most likely faster. Also, I do not think that they are used to protect all write operations [41]. ZFS transactions add more extra overhead since they protect all file system

modifications and respect ACID semantics. Transactions are not needed when reading data, since nothing needs to be modified. That makes ZFS sequential reads faster than sequential writes.

In the "Random Seeks" test bonnie-64 creates 4 child processes which perform 4000 seeks to random locations. On 10% of the random seeks, bonnie-64 actually modifies the read data and rewrites them. What we can see in this test is that Btrfs has an effective seek rate of 89-95 seeks/second, while ZFS has a lower rate: 39-40 seeks/second. This most likely happens because Btrfs is tightly connected with the Linux kernel, and performs better than ZFS regarding random seeks and modifications when the cache is not utilised. ZFS depends heavily on its ARC cache, and cannot achieve a high random seek rate when the cache is bypassed [101].

A negative aspect about Btrfs is that it generally keeps the CPUs more busy than ZFS. Using a single data structure (b+tree) for doing everything results in a simpler file system architecture and makes coding and refactoring easier than using multiple data structures - objects, object sets, metaslabs, space maps, AVL trees, etc. [19, 84, 132]. But I believe that it also affects certain operations, which end up being CPU bound.

Let's move to the results of bonnie-64 when RAID 1 (mirroring) is used. Theoretically, when mirroring is used the output rates should be slightly lower compared to the ones of using just a single disk. This is because the file system must write the same data twice (on both disks). The input rates might be increased, if certain optimisations are supported by the RAID implementation. For example, if the file system can read from both disks simultaneously, the input rates will be increased. Tables 4.3 and 4.4 display the mirroring results.

Table 4.3: Sequential output on mirroring

Mirror		**Sequential Output**					
File system	Size	Per Char		Block		Rewrite	
	GB	M/sec	%CPU	M/sec	%CPU	M/sec	%CPU
ZFS	12	27-28	10	27	3	19-20	2.17
Btrfs	12	52-53	53-59	52-53	4	22	4

The first thing that we can observe is that the output rates are more or less either the same or slightly decreased, both for ZFS and Btrfs. The same is not true for the input rates. Here we can see that ZFS, with a per block input rate of 82 MB/second, outperforms Btrfs, which has a per block input rate of 52-53 MB/second. This tells us that the software RAID 1 implementation of ZFS can

achieve better input rates that the Btrfs implementation. ZFS is more scalable regarding sequential input, which most likely means that the dynamic striping (any block can be written to any location on any disk) [51] of ZFS works better than the object level striping of Btrfs.

The things are different when observing the "Random Seeks" test. In this case, the Btrfs implementation seems to make use of both disks, and achieves an effective seek rate of 161-182 seeks/second. ZFS on the other hand, achieves a little higher effective seek rate than it does when a single disk is used: 66-68 seeks/second. The same argument still holds here: Btrfs, being tightly integrated with the Linux kernel and designed towards simplicity, achieves a higher random seek rate than ZFS when the cache is bypassed.

4.2.3.2 Bonnie++ performance analysis results

The next performance analysis results that we will investigate are produced by the Bonnie++ micro-benchmark [14]. I have decided to use Bonnie++ because I managed to find some results [88] that I can use as a reference. Apart from that, Bonnie++ is interesting due to its metadata related operations [76], which are not supported by bonnie-64. The version of Bonnie++ I am using is 1.96 [39]. An example of a Bonnie++ execution using the same configuration as in [88] is `bonnie++ -s 16384 -d /mnt/btrfs -m "Single Disk" -n 8:4194304:1024 -u 0 -f -q >> bonnie-btrfs-0`. To get details about what each command argument is, please refer to the Bonnie++ user manual.

A few notes before analysing the Bonnie++ results: (1) I have skipped the not really interesting per character results, to fasten the benchmarking process. (2) Bonnie++ by default reports results in KB/second. I have converted KB into MB [94] and rounded them up using a single digit precision, to make the results more readable. (3) Bonnie++ normally reports more numbers than the ones shown in my results, for example disk latencies. Whatever I have skipped, I

Table 4.4: Sequential input and random seeks on mirroring

Mirror		**Sequential Input**				**Random Seeks**	
File system	Size	Per Char		Block			
	GB	M/sec	%CPU	M/sec	%CPU	/sec	%CPU
ZFS	12	74-75	35	82	3.9	66-68	0.33
Btrfs	12	51	71-72	52-53	5	161-182	1

did it because either it was too time consuming to include it, or the benchmark could not report meaningful numbers, despite the different configurations I have tried [76]. Anyway, this is not very important. The important thing of this experiment is to see the relations between my bonnie-64 results, my Bonnie++ results, and the Bonnie++ results of [88].

Tables 4.5 and 4.6 show the Bonnie++ single disk results. First of all, the differences between the numbers presented in Tables 4.1, 4.2, 4.5, 4.6, and [88] are absolutely normal. This is happening because there are differences between the underlying hardware, the versions of software used, the operating systems, etc. [125]. What we should focus is not the actual numbers, but whether the numbers agree or not.

Table 4.5: Sequential output and input on the single disk

Single Disk		**Sequential Output**				**Sequential Input**	
File system	Size	Block		Rewrite		Block	
	GB	M/sec	%CPU	M/sec	%CPU	M/sec	%CPU
ZFS	16	26.5-26.6	3	16.9-17	3	50-50.5	3
Btrfs	16	47.8-48	8	20.8-21.2	5	51.7-52.5	4-5

Table 4.6: Random seeks and metadata creation on the single disk

Single Disk		**Random Seeks**		**Sequential Create**		**Random Create**	
File system	Size			Create		Create	
	GB	/sec	%CPU	/sec	%CPU	/sec	%CPU
ZFS	16	137-148	1	1100-1147	9-10	1246-1287	7-8
Btrfs	16	68-69	75-77	24-25	5	24	5

To begin, all results indicate that Btrfs achieves higher sequential output rates than ZFS. My tests show that the sequential input rates on single disks are identical for both file systems - slightly higher for Btrfs. I believe that the low performance of ZFS presented in [88] results because ZFS is configured to run to the user space of a GNU/Linux system, using a service like File system in User space (FUSE) [46]. In my tests ZFS runs in the kernel space of its native operating system. That is why it is performing much better. Having said that, it seems that extents can achieve higher sequential output rates than metaslabs on single disks. The sequential input rates on single disks are slightly higher for extents over metaslabs.

The next thing that we should observe is that in contrast to the bonnie-64 results, in Bonnie++ ZFS achieves a higher effective seek rate (137-148 seeks/second) than Btrfs (68-69 seeks/second). One thing that we should be aware of about the random seek test before making any conclusions, is that it is not the same for bonnie-64 [16] and Bonnie++ [40]. Two important differences are the number of the created child processes (four in bonnie-64 against three in Bonnie++) and the number of seeks which are performed (4000 in bonnie-64 against 8000 in Bonnie++). Without being able to make a safe conclusion because of such differences, the random seek result of Bonnie++ indicates that ZFS allocates related blocks close on-disk to reduce the disk arm motion [120, chap. 4].

The final observation about the single disk results is that ZFS outperforms Btrfs in the metadata creation tests: "Sequential Create" and "Random Create". I think that it is expected for ZFS to perform much better than Btrfs in all metadata tests, since in ZFS metadata are compressed by default, while in Btrfs they are not.

It is time for the RAID 1 results of Bonnie++. They are shown in Tables 4.7 and 4.8.

The first thing that we observe is that the output rates are consistent with the output rates shown in Table 4.3. Btrfs achieves higher output rates (both block and rewrite) than ZFS. The per block output rate of Btrfs, 48.4-48.6 MB/second, is much higher than the per block output of ZFS, 26-26.3 MB/second. The rewrite output rate is slightly higher for Btrfs: 21.5-21.8 MB/second, against 19-19.3 MB/second for ZFS. Extents achieve higher sequential output and rewrite rates on both single disks and mirroring. Those results verify that extents can scale regarding sequential output.

The input rates and the effective seek rates are also consistent with the rates of Tables 4.3 and 4.4. ZFS achieves a per block input rate of 76-77.7 MB/second, while Btrfs reports a per block input rate of 54.5-54.9 MB/second. The input rates verify that ZFS has a better data striping (called dynamic striping) implementation than Btrfs. Regarding random seeks (202-247 for ZFS against 109-118 for Btrfs), it is expected that the total number of read seeks will be increased when mirroring is used [40]. ZFS is still better on this test, which I suspect that happens because the block allocation policies of ZFS can scale more than those of Btrfs.

The final numbers that we should observe are related with metadata creation. There are no great changes when mirroring is used regarding metadata creation ("Sequential Create" and "Random Create"). ZFS still performs much better than Btrfs, and this is happening because metadata in ZFS are compressed by

default, whereas in Btrfs they are uncompressed.

Table 4.7: Sequential output and input on mirroring

Mirror		**Sequential Output**				**Sequential Input**	
File system	Size	Block		Rewrite		Block	
	GB	M/sec	%CPU	M/sec	%CPU	M/sec	%CPU
ZFS	16	26-26.3	3	19-19.3	3	76-77.7	5
Btrfs	16	48.4-48.6	8	21.5-21.8	5-6	54.5-54.9	5

Table 4.8: Random seeks and metadata creation on mirroring

Mirror		**Random Seeks**		**Sequential Create**		**Random Create**	
File system	Size			Create		Create	
	GB	/sec	%CPU	/sec	%CPU	/sec	%CPU
ZFS	16	202-247	2-3	1183-1270	9-11	1294-1353	8
Btrfs	16	109-118	113-142	25	5	25-26	5

You should now have an idea about the usefulness of micro-benchmarks. They raise important questions like: (1) "Why Btrfs achieves higher sequential output rates than ZFS?". "Is it because extents work better than variable block sizes?". "Or is it because ZFS is designed with innovation and portability in mind (complex data structures, separate cache, separate I/O scheduler, etc.), while Btrfs with simplicity (simple data structures, system caches, kernel's scheduler, etc.)?" "Is it expected or there is space for improvements on this ZFS part?". (2) "What is wrong with the RAID 1 input rates of Btrfs?". "Should the implementation be improved to support reading from both disks simultaneously? Or reads are not efficient because data are not striped efficiently when they are written on the disks?". (3) "Why is Btrfs keeping the CPUs so busy when performing random seeks?". "Is it because the b+tree algorithms are very CPU bound?". "Is this fine or the Btrfs developers must optimise the algorithms to make the file system less CPU bound?". (4) "Should Btrfs reconsider adding metadata compression to improve its metadata related operations, or this is not very important?".

But since it is not safe to make conclusions by using only micro-benchmarks, my next steps include using a workload generator and a macro-benchmark emulator.

4.2.3.3 Filebench performance analysis results

During my experiments I will use Filebench to emulate popular macro-benchmarks [107]. Filebench includes many nice features [111]. But I believe that the most important feature is the support for multithreading. The biggest problem with most benchmarks is that they are single threaded. A single threaded benchmark cannot produce heavy and realistic workloads for exercising a file system properly. Multithreaded benchmarks can exercise file systems much better. This fact makes the usage of Filebench more appropriate than using other perhaps more popular but single threaded benchmarks, like PostMark [58, 124].

In this experiment I will use Filebench to emulate the Standard Performance Evaluation Corporation (SPEC) SFS benchmark. SPEC SFS is officially used to measure the server throughput and response time of the Network File System (NFS) version 2 and version 3 servers [124]. The Filebench test which emulates SPEC SFS is called fileserver [110]. The major operations of the fileserver test are opening and closing files. The rest operations are reading file metadata (stats) and data, writing, randomly appending, creating, and deleting files. I am using Filebench version 1.4.8.fsl.0.5 [45]. The advantage of using the port of the File systems and Storage Lab (FSL) is that the same version works both in GNU/Linux and OpenSolaris. The disadvantage is that only the interactive part of Filebench has been ported [112], and because of a bug I have to quit and restart Filebench after each execution. I am using the recommended settings for small configurations, as given by the original Filebench developers [109]. A sample execution of Filebench using the fileserver test is given in Example 4.1.

Example 4.1 Running Filebench using the fileserver test

```
$ go_filebench
> load fileserver
> set $dir=/mir
> set $nfiles=50000
> run
> stats dump "zfs-0"
> quit
```

The Filebench fileserver summarised results for Btrfs and ZFS on the single disk configuration are displayed in Table 4.9. The fileserver test uses 50 threads by default during its execution. I have divided the Filebench results into three categories. “Throughput breakdown” shows the number of operations the file system can execute. “Bandwidth breakdown” shows how many MB/second the file system is able to use for any operation: read, write, random append, etc. “Efficiency breakdown” shows the code path length in instructions/operation

[108]. For all the reported results except "Efficiency breakdown", the higher the numbers are, the better the result is.

What we can see by observing Table 4.9 is that Btrfs has a higher throughput breakdown and bandwidth breakdown than ZFS. Btrfs is able to execute 32091-35755 operations in total and 535-596 operations/second, while ZFS can execute 15760-16387 in total and 260-270 second. Btrfs achieves a bandwidth rate of 12-14 MB/second, while ZFS achieves a bandwidth rate of 6 MB/second. ZFS has a better efficiency breakdown, which is 410-586 instructions/operation, whereas Btrfs has a longer code path length of 722-769 instructions/operation. This generally means that ZFS works more efficiently with multiple threads than Btrfs, even though it cannot outperform it (in this test).

Table 4.9: Fileserver results on the single disk

Single Disk	**Throughput breakdown**		**Bandwidth breakdown**	**Efficiency breakdown**
	Ops	Ops/sec	MB/sec	Code path length in uS/op
ZFS	15760-16387	260-270	6	410-586
Btrfs	32091-35755	535-596	12-14	722-769

The RAID 1 fileserver summarised results for Btrfs and ZFS are shown in Table 4.10. After observing the results, there are not a lot of things to comment. The only change is that the numbers are higher, but the outcomes are still the same: Btrfs has a higher throughput breakdown and bandwidth breakdown than ZFS, while ZFS has a higher efficiency breakdown than Btrfs.

Table 4.10: Fileserver results on mirroring

Mirror	**Throughput breakdown**		**Bandwidth breakdown**	**Efficiency breakdown**
	Ops	Ops/sec	MB/sec	Code path length in uS/op
ZFS	23649-24647	390-407	9-10	525-678
Btrfs	46707-49003	778-817	18-19	768-786

Apart from the summarised results that I have discussed, Filebench provides detailed results about the performance of the file system per operation. For example, Filebench reports the throughput, bandwidth, and efficiency breakdown of a single read file or write file operation. Such results are very important for file system developers, since they can help them recognise which specific operations

do or do not perform as expected [108]. However, it was too time consuming for me to include the detailed results, so I have decided to skip them.

Since I had the chance to cheat and take a look into the detailed results, I can tell you that Btrfs beats ZFS in the read, random append, and create tests. Especially file creation seems to be the bottleneck of ZFS. To give you some numbers, the read latency of Btrfs on the single disk is in terms of 155.9 millisecond (msec), while in ZFS is in terms of 586.9 msec. I suspect that this happens because Btrfs achieves better short-term prefetching than ZFS [101]. The random append latency on the single disk is in terms of 478.8 msec in Btrfs, while in ZFS is in terms of 613.7 msec. This makes sense since all file system modifications in ZFS are protected by ACID and are slower than Btrfs. File creation on the single disk is in terms of 0.1 msec in Btrfs, whereas in ZFS it is in terms of 552.4 msec! File creation is slow in ZFS because when a file is initially created, its metadata should also be compressed, since metadata are compressed by default in ZFS.

By using the macro-benchmark emulator and after observing the detailed results of Filebench, we can see that Btrfs outperforms ZFS on the fileserver test and is the right choice to use on NFS servers with small disk capacities, and applications that give special emphasis on reading, creating, and randomly appending files.

4.2.3.4 IOzone performance analysis results

The trace/workload generator I am going to use is IOzone [56]. IOzone is one of the recommended workload generators by the storage benchmarking community [125]. I think that IOzone is very convenient for creating what-if scenarios. For example [120, chap. 4] suggests that since disk space is nowadays cheap and huge, it might be better to use a large block size and accept the wasted disk space, for the sake of performance. Most file systems, including Btrfs, use a fixed block size which is usually 4 KB by default. ZFS is the only file system that I know which is trying to adapt the block size as the length of a file changes.

In this experiment I will try to run some basic tests with IOzone, using several different file sizes and a fixed record size of 64 KB. The 64 KB record size emulates applications that read and write files using 64 KB buffers. The goal of the experiment is to see the out of the box performance of Btrfs and ZFS against a predefined workload. An example of using IOzone to conduct this experiment is `iozone -i 0 -i 1 -i 2 -g 4G -q 64K -a -S 6144K -f /prim/i >> io`. For details about what each argument means, please consult the user manual of IOzone [89]. I am using IOzone version 3.347 (iozone3_347.tar) [56].

Tables 4.11 and 4.12 show the IOzone results on the single disk configuration. Only the "best" results are displayed. By best I mean the results where most of the tested file system operations achieve the maximum performance. IOzone reports results in KB/second. Using the same approach I used with the results of Bonnie++, I have converted KB into GB to make the output more readable.

The first thing that looks strange after observing the IOzone results is that both file systems report very high data rates: in terms of GB/second! I have discussed this issue with the main IOzone developer, Don Capps. His response was:

> "Once the file size is larger than the amount of RAM in the system, then you know for sure that you are seeing purely physical I/O speeds. Until the file size gets that large, the operating system and hardware are going to be trying to cache the file data, and accelerating I/O requests. From your results, it appears that your operating system has about 1 Gig of RAM that it is using to cache file data. If you look closely at the re-read results, you can also see the impact of the CPU L2."

Table 4.11: Write, rewrite, and read on the single disk

Single disk	**Size**	**Record length**	**Write**	**Rewrite**	**Read**
	KB	KB	GB/sec	GB/sec	GB/sec
ZFS	2048	64	1.6	3.6	6.2-6.3
Btrfs	2048	64	2	2.6	4.3-5

Table 4.12: Reread and random read/write on the single disk

Single disk	**Size**	**Record length**	**Reread**	**Random read**	**Random write**
	KB	KB	GB/sec	GB/sec	GB/sec
ZFS	2048	64	6.7	6	3.4-3.5
Btrfs	2048	64	4.9-5	4.9	2.7-2.8

In my case, the file size (4 GB) is two times larger than the amount of the system RAM (2 GB). It seems that the high rates occur because most data are cached.

Both file systems achieve their maximum performance when operating on 2 MB files (2048 KB). The biggest conclusion of this test is the significance of the cache

in the performance of ZFS. ZFS outperforms Btrfs in all tests but "Write". All the results of IOzone except (sequential) write contradict with the results of bonnie-64 and Bonnie++. This is not strange. In the bonnie-64 and Bonnie++ tests I bypassed the caches by setting a very big file size. In this test I make sure that the file system cache will be used by setting a small file size (`-g 4G`). I also make sure that the CPU level 2 cache will be used by defining it (`-S 6144K`). When the caches are used ZFS outperforms Btrfs on the majority of tests: COW works better in ZFS (rewrite). ZFS has a better cache utilisation than Btrfs (reread). And ZFS reports higher read, random read, and random write data rates. For once again, the Btrfs extents achieve higher sequential write data rates than the ZFS metaslabs. I believe that there might be a relation between the write test results and the slow file creation of ZFS, since before start writing in a new file you first need to create it.

Let's move on to the RAID 1 results of IOzone. They are shown in Tables 4.13 and 4.14. There are not a lot of things to comment regarding the mirroring results, since there are no significant changes. ZFS still outperforms Btrfs in all tests but write.

All results confirm that ZFS is the appropriate file system to use, both on single disks and on RAID 1 configurations, when the caches are utilised. In this test I am focusing on applications which use 64 KB buffers. But I believe that ZFS outperforms Btrfs when the caches are utilised no matter the size of the application buffers. This happens because the advanced ARC cache of ZFS performs much better than the existing system caches of the Linux kernel that Btrfs makes use of.

Table 4.13: Write, rewrite, and read on mirroring

Mirror	**Size**	**Record length**	**Write**	**Rewrite**	**Read**
	KB	KB	GB/sec	GB/sec	GB/sec
ZFS	2048	64	1.6	3.6	6.1-6.2
Btrfs	2048	64	2.1	2.6	4.8

Table 4.14: Reread and random read/write on mirroring

Mirror	**Size**	**Record length**	**Reread**	**Random read**	**Random write**
	KB	KB	GB/sec	GB/sec	GB/sec
ZFS	2048	64	6.4-6.7	6	3.4
Btrfs	2048	64	5	4.9	2.7

4.2.4 Scalability test results

In this experiment I am using again Filebench to test how well can ZFS and Btrfs scale. This time, I am using a different Filebench test. It is called varmail [110], and is nothing more than a multithreaded version of PostMark [58, 124]. Varmail uses by default 16 threads. The major operations of the varmail test are opening, closing, and reading files. The rest operations are file system synchronisation (`fsync`), random file appends, and file deletes. I have decided not to include any examples of running varmail, since you can consult Example 4.1. Only the name of the test changes. The results of varmail are displayed in Table 4.15.

It is not hard to see that ZFS beats Btrfs in all tests without problems. ZFS has better throughput breakdown, better bandwidth breakdown, and achieves a much shorter code path length. Btrfs has big troubles with the varmail test, which reveals its scaling problems. I was once again curious to see the specific operations where Btrfs cannot perform well. I found out that the biggest pain of Btrfs is by far `fsync`. Let's see the results of two `fsync` tests. In the first test ZFS reports 68.7 msec, while Btrfs reports 7949.2 msec! In the second test, ZFS reports 1403.8 msec, whereas Btrfs reports 5945.3 msec. In both cases, `fsync` in Btrfs is significantly slower than it is in ZFS. It seems that the issues of Btrfs with `fsync` are well-known to the free software community [103]. Btrfs is not (at least yet) optimised to work synchronously.

Table 4.15: Varmail results on the virtual machine

Virtual M.	**Throughput breakdown**		**Bandwidth breakdown**	**Efficiency breakdown**
	Ops	Ops/sec	MB/sec	Code path length in uS/op
ZFS	74939-148453	1231-2436	4-9	3060-4852
Btrfs	9088-31675	149-527	1-2	-2779961-8696048

Emulating popular benchmarks like PostMark helps us to decide if a file system is the right one for a specific purpose. In this case, Btrfs cannot scale enough to be used as the primary file system on mail/database servers with big disk capacities. ZFS is the best choice for mail/database servers with big capacities, and applications that rely heavily on `fsync` operations.

Chapter 5

A file system call interface for FenixOS

This chapter begins with a discussion regarding legacy file system interfaces. It then continues with presenting my contributions to the design of a file system call interface for FenixOS. The final part of the chapter includes a graphical representation and textual description of the major file system calls, as well as a hypothetical example of using the FenixOS file system interface.

5.1 On legacy interfaces

In chapter "Copy On Write based file systems" I have discussed about how COW based file systems provide (meta)data consistency using transactional semantics. I have explained what transactions are, and what ACID means with regard to file systems. I have also briefly mentioned the big problem with current file systems and operating systems: the lack of support for transactional semantics. Time to extend this discussion a little more.

Most system call interfaces of UNIX-like operating systems are POSIX compliant. Other system call interfaces follow their own approach, which is actually not very different from POSIX [120, chap. 1]. Either POSIX compliant or not,

the system call interfaces of all commercial operating systems have the same problem: they do not support transactions [96]. Transaction support at the operating system level is critical. Transactions provide a simple API to programmers for (1) eliminating security vulnerabilities, (2) rolling back unsuccessful software installations and failed upgrades, (3) ensuring that file system updates are consistent, (4) focusing on the logic of a program instead of struggling to find out how to write deadlock avoidance routines, (5) write programs which can perform better than lock-based approaches, etc. [11, 74, 96, 114, 126, 130].

We use system transactions to protect the system state. The reasons are that system transactions (1) do not require any special hardware or software (for example special compiler support), (2) support system calls, (3) everything inside a system transaction (including system calls) can be isolated and rolled back without violating transactional semantics, (4) can be optionally integrated with Hardware Transactional Memory (HTM) [11, 74] or Software Transactional Memory (STM) [126] to protect the state of an application [96].

Nowadays, most platforms are based on multi-core and multiple processors. For taking advantage of such architectures, multithreading and parallel programming are essential. Transactions can be extremely helpful for concurrent programming. Modern operating systems do not have an excuse for not supporting transactions, since databases have been supported them without problems for a long time [114, 130].

Of course, not only operating systems should support transactions, but they should also expose them to programmers. ZFS for instance, even though it offers ACID semantics for all file system modifications, transactions are transparently processed at the Data Management Unit (DMU) layer. They are only used internally and are not visible to programmers. At the file system level, only a legacy POSIX interface is available through the ZFS POSIX Layer (ZPL) [132].

Hopefully at this point I have clearly expressed my thoughts about the usefulness of transactions. Transactions are not useful only for file systems. They can also be useful for other operating system parts. Such parts are processes, memory, networking, signals, etc. [96]. But since my work focuses on file systems, my discussions from now on will be mostly related with the file system call interface.

5.2 FenixOS file system call interface

The final part of my thesis includes defining a modern file system call interface for FenixOS. My focus is on the higher interface of the file system to user

processes. It is not on the actual implementation of the system calls. FenixOS at the moment lacks of both a VFS and a file system. In a sense, you can think of my interface as the VFS of FenixOS.

The fact that FenixOS is a research operating system is very convenient for introducing new aspects. For example people who want to support transactions on systems like GNU/Linux must do it with a minimal impact on the existing file system and application code [114]. The result is either a slow user space implementation, or a complex user API. Others, for instance ZFS, support transactions only at the file system level. And even though ZFS supports transactions, it does not expose them. This limits the usefulness of transactions. In FenixOS we do not have such concerns. We do not need to be POSIX compliant because it is not a big deal for us if POSIX code cannot be used on FenixOS. We would rather prefer to provide to the programmers a better system call interface. A system call interface where transactions are exposed, and using them is intuitive and straightforward. And we believe that it is much better for us to design a transaction oriented system call interface from scratch, instead of trying to add transaction support to a non transaction oriented interface like POSIX.

At this point I would like to clarify that I have nothing against POSIX. POSIX was defined to solve a very specific problem. And indeed, I believe that it solves it very well. But with the transition to multiprocessor and multi-core architectures, the usage of concurrent programming models raises the need for designing new system call interfaces, which are more appropriate to transaction oriented code. If necessary, we can always create a POSIX compatibility layer to support POSIX code in FenixOS. Many operating systems have already done this successfully. Only a few examples are APE, ZPL, and Interix [82].

After studying several system call interface designs, I have concluded that the most innovative is the one used in ZFS. The “Related Work” chapter includes a discussion about each interface, but let me say a few words about them. Btrfs violates ACID semantics [41] which is not what we want. Valor is only file system oriented, has a complex API, and does not use COW [114]. We focus on COW and we want to provide an intuitive and simple API to users. Moreover, we would like to use transactions not only in file systems, but also in the rest operating system parts. Oracle’s Berkeley DataBase (BDB) is not considered a file system call interface, since it can be used only on flat files like databases [91]. Even though there are user space solutions based on BDB which provide transactions, they have proved to be very slow. TxOS is intuitive to use and has nice features, but it is tightly connected with the implementation of the Linux kernel. TxOS relies heavily on the data structures and mechanisms of GNU/Linux which means that it has a complex GNU/Linux related implementation [96].

ZFS seems to be a good choice. Obviously I am not referring to the legacy ZPL layer. This is nothing more than yet another POSIX interface. The interesting part of ZFS in this case is the DMU layer. When I discussed about ZFS, I have mentioned how it uses the concepts of objects and object sets to organise the on-disk data. Objects, object sets, and transactional semantics are extensively used in the DMU layer [84, chap. 3], [132]. We believe that following a design similar to the design of DMU is the right approach for creating a transaction oriented file system call interface for FenixOS. Moreover, using object based approaches can contribute to the creation of semantic and searchable VFS infrastructures [104, 113].

DMU cannot be used as is. The first issue which is implementation related is the programming language. While ZFS is written in C FenixOS is written in C++. I am not planning to begin another language flamewar here. A serious technical comparison of several programming languages, including C and C++ can be found in [98]. If you would like my opinion, I believe that C is faster, simpler, and requires less memory than C++. On the other hand, using C++ in operating systems helps you to take advantage of all the standard object oriented features - encapsulation, inheritance and polymorphism - as well as some exclusive features of the language: namespaces, operator overloading, templates, etc. Another problem is that DMU cannot be exposed as is. Since DMU is not exposed to programmers in ZFS, issues like file permissions and protection are taken care by the ZPL layer. In our case, we have to solve such issues at the VFS level.

5.2.1 Design decisions

Let me discuss about the design decisions regarding the file system call interface of FenixOS. Our main goal is to design a file system call interface which supports transactions. Most transaction oriented interfaces are inspired by Oracle's BDB [91]. A programmer uses simple `begin` and `commit` statements to initiate and complete a transaction. During the transaction, the programmer can use the `abort` statement to terminate the transaction and ignore all its changes. Some interfaces also provide a `rollback` functionality, which does the same thing as undo: it reverts all the changes of the most recent commit of the transaction which is rolled back. Most interfaces also provide the option of relaxing ACID (usually isolation and durability) guarantees for the sake of optimisation [91, chap. 1], [114]. We think that violating ACID is a poor design decision and we do not wish to adopt it.

Our interface will support the highest database isolation degree, which is number 3 [91, chap. 4]. Moreover, the interface will use a strong atomicity model. Defining the model of atomicity which will be used is important, since a direct

conversion of lock based code to transaction based code is generally unsafe [10]. We will use a strong atomicity model because it ensures that system updates are consistent even if both transactional and non-transactional activities try to access the same system resources. This is done by serialising transactional and non-transactional code and deciding about which one should be executed first, by assigning priorities. There is no need to use HTM/STM, since we want to protect only the system state and not the data structures of a specific application [96]. However, integrating STM with the interface of FenixOS should be straightforward, in case the state of an application needs to be protected.

Nested transactions will be supported if necessary. We will support nested transactions provided that they will not make the implementation very complex. We will use COW (lazy version management) instead of journaling (eager version management) to provide transaction support. COW has many advantages over using an undo log. Better performance, ability to abort transactions instantly, cheap snapshots and clones, etc. [96]. When using COW, all the file system changes of a transaction can be private and asynchronous until committing it. COW helps an operating system to provide transaction support without suffering from a big performance hit.

Apart from offering transactions, we have decided to increase the flexibility of the rest file system parts. An example of such a part is file protection. Traditional mode style file attributes [69] used by default in most UNIX-like operating systems are simple, but they are not rich enough to cover the needs of complex system configurations. As the complexity of a system increases, the inflexibility and the poor semantics of traditional UNIX attributes cause big headaches to system administrators. To solve this problem, some systems provide Access Control List (ACL) support. To put it simply, ACLs are entries which describe which users or processes have the permission to access an operating system object, and what operations they are allowed to perform on it. ACLs have better semantics and are richer than traditional UNIX file permissions, allowing systems to provide a better security model than the security model of traditional mode based systems.

We have decided to support the standard ACLs of the Network File System Version 4 Minor Version 1 (NFSv4.1) Protocol [106]. In contrast to "POSIX" ACLs which are not really POSIX since they were never accepted by the POSIX standard, NFSv4.x ACLs are a standard part of the NFS protocol. Apart from being standard, NFSv4.x ACLs are richer than "POSIX" ACLs. For those reasons, operating system developers have started to implement support for NFSv4 ACLs. ZFS, OpenSolaris, and FreeBSD already support NFSv4 ACLs [73], [83, chap. 8], [87].

Operating systems which add support for NFSv4.x but also support "POSIX"

ACLs and traditional mode file attributes are facing a big challenge. It is very hard to keep backward compatibility with all permission models. One problem is that conversions from one model to another are complex. A bigger problem is that when translating from a finer-grained to a less fine-grained model, for example from NFSv4 to "POSIX" ACLs. In this case, the NFSv4 semantics which are not supported by "POSIX" ACLs must be sacrificed [4]. Also, computing the mode from a given ACL and making sure that both are synchronised is time consuming and increases the complexity of the file system [44]. To avoid all those issues, we have decided to support only native NFSv4.1 ACLs. We are not planning to support "POSIX" ACLs or traditional mode style file attributes. This means that users will need some time to get familiar with NFS ACLs. But this is for their benefit.

Another feature that we are planning to support is memory mapped resources [85]. Memory mapped resources are objects that are placed (mapped) on the address space of a process. By objects I mean files, devices, shared memory, etc. Similar to virtual memory pages [115, chap. 8], [120, chap. 3], when memory mapping a resource, only a specific section of the resource can be mapped. Once mapped, the resource can be shared between different processes. Alternatively, the changes can be set as private to the process which did the mapping. Memory mapped resources have many useful applications: They can be used in a shared memory model to reduce IPC costs. Also, memory mapping is generally optimised because it is integrated with the paging system [59], [79, chap. 5]. We will use memory mapped resources in FenixOS to replace the legacy `read/write` system calls [85]. FenixOS, like other operating systems, will use a unified cache manager [79, chap. 5]. This means that there will be no separate block cache, page cache, dentry cache, etc. A single virtual memory cache will be used to cache everything. Using a unified cache with memory mapped resources adds one more benefit. When the resource is mapped it is placed directly in the cache to be edited. There is no need to copy it in a separate memory location which acts as the cache.

5.2.2 File system calls

It is time to present the actual file system calls that I ended up with. Let's start with some general concepts.

A file in UNIX-like systems is an unstructured sequence of bytes. Everything is represented as a file. The various UNIX file types are shown in Figure 5.1. Historically, there are two kinds of files in UNIX: files and directories. Files are divided into regular and special files. Regular files can either be plain files, which are human readable, or binary, which are not human readable. Special files can

be character files, which are used for modelling serial I/O devices like terminal and printers, or block files which are used for modelling disks. Directories are files that can contain other files inside them and are used to organise the structure of a file system.

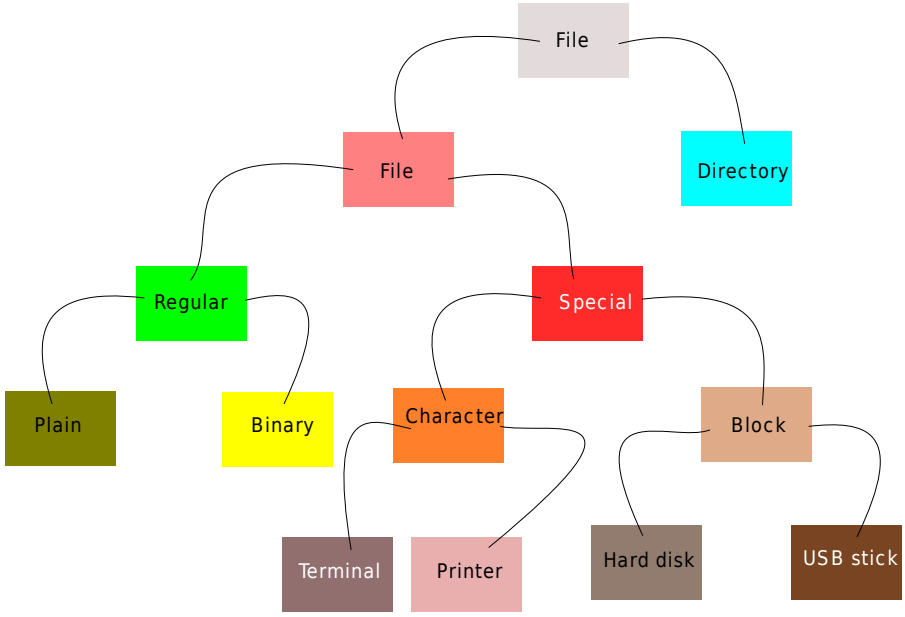

Figure 5.1: File types of UNIX-like operating systems.

There are many different ways of implementing files and directories. Different file systems follow different approaches. For example, in Btrfs a file is described by a name and the i-node related with it. An i-node is the data structure used for keeping track of the blocks that belong to a particular file [120, chap. 4]. ZFS uses a different data structure, called the dnode. A dnode is a collection of blocks that make up a ZFS object [84, chap. 3]. Whether the file system uses i-nodes, dnodes, or another data structure is not our concern, since we are working on a higher level. What we should observe is that all file systems need a way of keeping track of the blocks that belong to a particular file. It does not matter if this is a plain file, a device file, or a directory (which is nothing more than a list of files). Thus we can abstract all those different kinds of files and their data structures in a single concept: the `VObject`. A `VObject` can be used to model a regular file, a special file, a directory, etc. It is up to the file system developer to take care of the implementation details regarding the data structures related with a `VObject`.

Apart from providing an abstraction of the different files, we need a way of

grouping those files together. This is what a file system is: a group of files sharing the same properties. Such properties can be the (1) owner of the file system, which is the operating system user who has full privileges over it, (2) file system type, which indicates whether it is ZFS, Btrfs, FAT32, etc., (3) root directory, which is the parent directory of all the files that belong to the file system, etc. In FenixOS, the abstraction which is used to describe a file system is the `VObjectSet`. Examples of `VObjectSets` are a ZFS file system installed on a hard disk, a FAT32 file system installed on a USB stick, etc.

Up to now we have defined the abstractions of the different file and file system types. But this is not very far from what all VFS tend to do [63]. We should not forget that we want an interface which supports transactions, and exposes them to programmers in a simple way. Our next step is to define what a transaction is in FenixOS. A `VTransaction` in FenixOS adds support for transactional semantics to the file system call interface. An executed file system call is always under the control of a `VTransaction`. Transactions control whole `VObjectSets`, and operate on their `VObjects`. This makes sense, since a file (`VObject`) always belongs to a file system (`VObjectSet`).

Figure 5.2 shows a graphical representation of the concepts being discussed so far. It also includes the major file system calls of FenixOS. The class diagram shown in Figure 5.2 is not a conceptual class diagram [7], but a compact version of the actual system call definition. It is compact because the point is to focus on the important stuff and not on every small detail.

By observing Figure 5.2 we can see that the most important class of the file system call interface is `VTransaction`. All the file system calls are inside it because whatever happens during a system call must be protected by ACID. A significant different between our interface and DMU is that a single `VTransaction` can operate on more than one `VObjectSets`. This means that a single transaction can modify more than one file systems. DMU supports only one object set/-transaction. But we believe that supporting multiple object sets/transaction is important. A good usage of this feature is creating an index of the contents of a ZFS file system which is installed on the hard disk, and a FAT file system which is installed on a USB stick atomically. This can be done only if multiple `VObjectSets`/`VTransaction` are supported.

Another thing that we can observe in Figure 5.2 is that only the `VTransaction` can add `VObjects` in a `VObjectSet`. That is why there is no direct connection between a `VObject` and a `VObjectSet`. That also makes sense, since files should be created/removed from a file system only using transactional semantics. If files can be added/removed from a file system outside a transaction, then transactions are useless.

Table 5.1 includes a short description of the file system calls, in the same order as shown in Figure 5.2. Let me say a few words about each system call. In the following descriptions I assume that the user who executes the system calls has the permissions to perform the requested operations. What happens in real is that every system call tries to satisfy the requested service. Only if the user has the necessary permissions, the system call will apply the requested changes. Otherwise, an error number, message, etc. is returned to the user.

`addObjectSet` adds an existing `VObjectSet` to a `VTransaction`. For example, it does not create a new file system. It passes an existing file system under the control of a transaction. All the file systems which might be modified should be added to a transaction. If multiple file systems must be modified by a single transaction, you must add all of them to the same transaction. Currently only two object sets/transaction are supported. But it is not hard to extend the interface to support more than two object sets/transaction. `removeObjectSet` removes a `VObjectSet` from a `VTransaction`. For example, it does not remove a file system from the disk. It just removes it from the transaction which controls it.

`openObject` opens an existing `VObject` for processing. For example, it does not create a new file. It opens a file which exists inside a `VObjectSet`. On success, `openObject` returns an object identifier, which is a concept similar with POSIX file descriptors. A file descriptor is a small number which can be used for referencing (reading, writing, removing, etc.) a file. An object identifier is a small number which can be used for referencing a `VObject`. `addObject` adds a new empty `VObject` to a `VObjectSet`. For example, it creates a new directory inside a ZFS file system. `checkObjectPermissions` is used to find out whether an operation on a `VObject` should be allowed/denied. An example is to check whether a file can be added inside a file system directory. `objectInfo` returns a structure which contains useful information about a `VObject`. Examples of such information can be the access permissions of a directory, the size of a file, the owner of a file, etc. `lookupObject` checks whether a `VObject` exists inside a `VObjectSet`. An example is to check if the file `/etc/motd` exists inside a Btrfs file system. `renameObject` gives a new name to a `VObject` of a `VObjectSet`. For example it can be used for renaming the file `/home/bob/notes` to `/home/bob/notes-old`. `removeObject` deletes a `VObject` from a `VObjectSet`. For example it can be used to delete the file `/home/bob/notes-old`. `moveObject` moves a `VObject` to a new `VObjectSet` location. An example is to move the file `/home/bob/notes` to the directory `/home/bob/notes-2010` (the result is `/home/bob/notes-2010/notes`). `closeObject` releases a `VObject` and all the resources related with it. For example, it closes the file `/home/bob/notes` and makes its object identifier available for reuse.

`memoryMap` maps a `VObject` to the address space of a process. For example it

can be used when a program want to write inside a file. `memoryUnmap` does the opposite work. It removes the `VObject` from the process.

`prepareForCreation` is used to inform the transaction that a `VObject` is going to be created inside the `VObjectSet` which it controls. An example is when creating a new directory inside a file system. `prepareForRemoval` does exactly the opposite: it is used when a `VObject` is going to be removed from a `VObjectSet`, for example when you plan to remove a file from a file system. `prepareForChanges` informs the transaction that a `VObject` is going to be modified. This is used when you want for example to edit a file.

`begin` states that the transactional part of the program begins. Only the code between `begin` and `commit` is protected by ACID. `commit` Only the code enclosed between `begin` and `commit` is protected by ACID. `commit` states the end of a transaction which means that after `commit` all the file system modifications of the transaction are permanent. Finally `abort` can be used by the programmer at any time to exit a transaction immediately and ignore all the modifications that happened inside it. An example of aborting a transaction is when you want for instance to create a new file and edit it within the same transaction. If you get a file system error because the file already exists (but did not exist before your transaction) it means that most probably somebody else has created it before you. In this case it is better to abort the transaction instead of trying to modify the file.

At this point you might argue that a UML class diagram and a description of the system calls is not enough for getting the whole picture. For this reason I have decided to include a small and simplified example which shows how a programmer can use the file system interface of FenixOS. The example will not work if you try it, since my job was to define and document the system calls, not to implement them.

Figure 5.3 shows a typical hierarchical file system structure of UNIX-like systems. The parent directory of all, called the root directory is represented as `/`. Two different file systems reside in the root directory. One is a ZFS `VObjectSet`, installed on the hard disk and located in `/zfs`. The other is a FAT32 `VObjectSet`, installed on a USB stick and located in `/media/stick`. In both file systems, there are different kinds of files. For example in the ZFS `VObjectSet` there is a `VObject` called "bob" which is a plain file, and a `VObject` called "work" which is a directory. What we want to do is to use our interface for renaming the "bob" `VObject` to "alice", and the "alice" `VObject` to "bob" atomically. If one of the rename operations fails, the other should not take place. This scenario might sound useless to you, but it is very similar with the scenario of adding a new system user. Think of it for a moment. The add user scenario says "I want to add this line in both `/etc/passwd` and `/etc/shadow`".

Either both files must be modified or none of them. The only difference is that in my scenario the two files reside in two different file systems, while in the add user scenario both files are located in the same file system.

The example of Listing 5.1 begins with the assumption that the two `VObjectSets` shown in Figure 5.3 exist and are called `zfs` and `fat`. Initially the two object sets which will be modified are added to a single `VTransaction`. The code continues by defining the on-disk path of the two objects (in this case plain files) which are also assumed to exist and will be modified. The next step is to a define the permissions that will be applied when trying to open the objects for processing. To get more details about the NFSv4.x permissions please consult [106]. The next part of the code deals with informing the transaction about the specific objects which are going to be modified. Since a transaction controls whole object sets, it is important to inform it about the specific object sets that you are planning to modify. This is important because we are following the semantics of the DMU [1] which requires from the programmers to state explicitly which objects they intend to create or modify. The next system call, `begin`, states that the transactional part of the code actually begins. After that, the two objects are opened for processing. `openObject` returns the object identifiers of the objects. After opening the objects, the actual rename takes place. Normally the programmer would continue to work with the object sets and their objects, but for the needs of this simple example we are done. The objects are closed so that the object identifiers can be reused by other objects, and the transaction is committed, which means that the changes will be applied on the hard disk and the USB disk, respectively.

Table 5.1: Short description of the FenixOS file system calls

System Call	Description
`addObjectSet`	Add an object set to the transaction
`removeObjectSet`	Remove an object set from the transaction
`openObject`	Open an existing object
`addObject`	Add a new empty object to an object set
`checkObjectPermissions`	Determine whether an operation on an object should be allowed/denied
`objectInfo`	Get information about an object
`lookupObject`	Search for an object in an object set
`renameObject`	Rename an object
`removeObject`	Remove an object from an object set
`moveObject`	Move an object in a new path
`closeObject`	Close an object and release all the resources related with it
`memoryMap`	Map an object in memory. Replaces `read/write`
`memoryUnmap`	Remove a mapping from the memory
`prepareForCreation`	Inform the transaction that an object is going to be created
`prepareForRemoval`	Inform the transaction that an object is going to be removed
`prepareForChanges`	Inform the transaction that an object is going to be modified
`begin`	State that the transactional part of the code begins
`abort`	Abort a transaction ignoring all the modifications made during it
`commit`	Commit a transaction to make sure that all the modifications made during it will be persistent

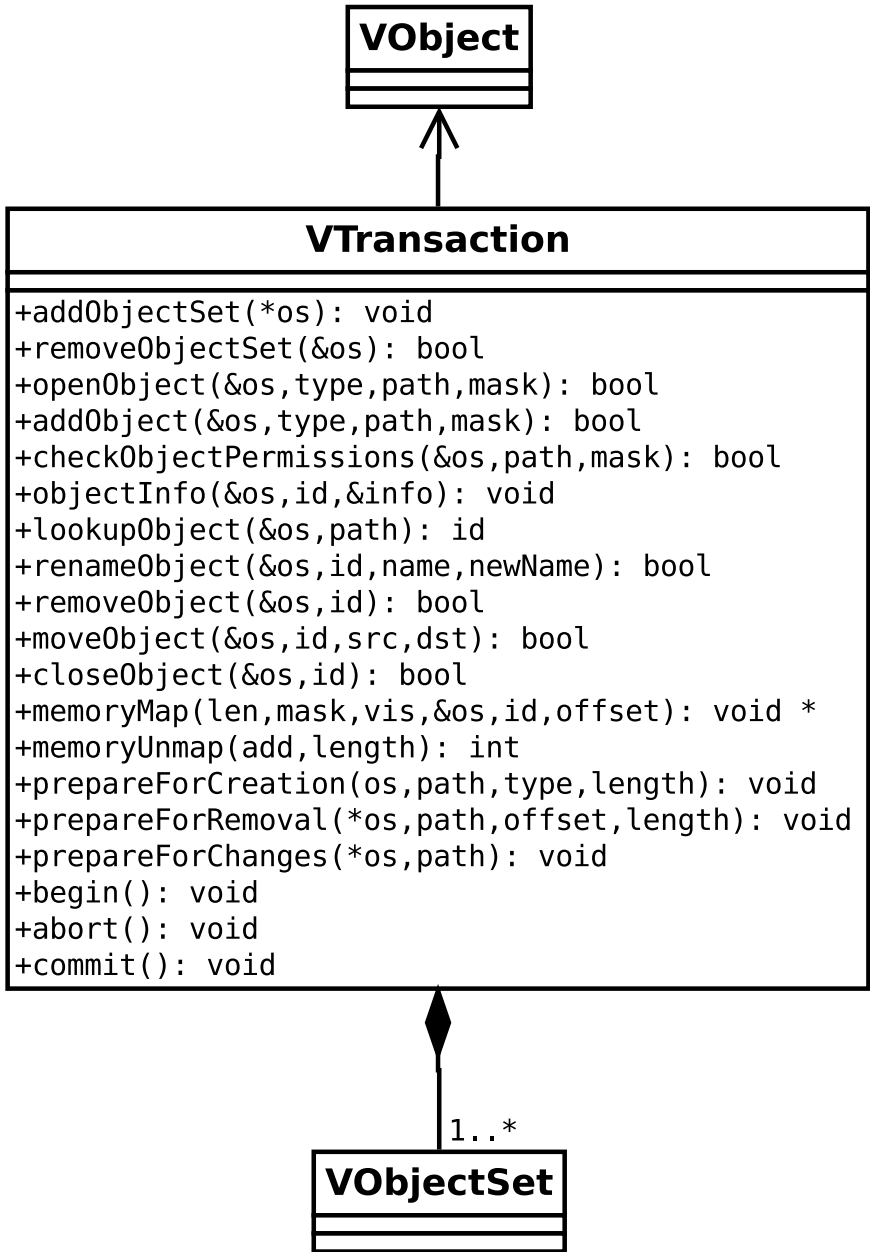

Figure 5.2: Compact class diagram of the FenixOS file system call interface. A transaction uses at least one object set which acts as the file system. Objects (plain files, directories, etc.) are added/removed from the used object set(s) under the control of a transaction

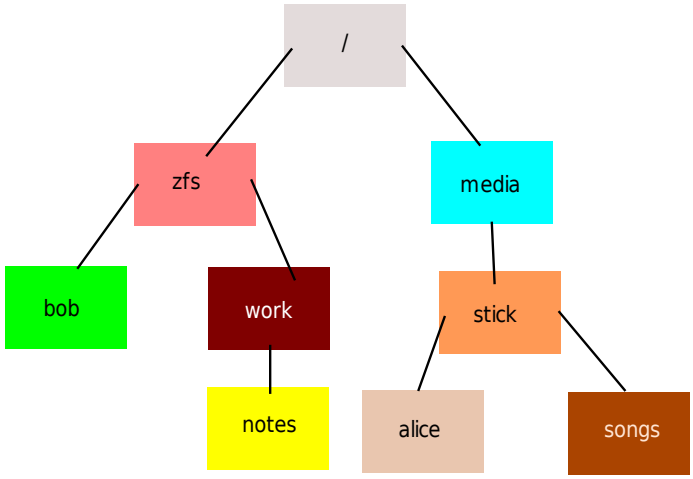

Figure 5.3: The directory structure of the example. Under the root directory /, there are two object sets: A ZFS object set is located in /zfs, and a FAT object set is located in /media/stick. Two files of the object sets, located in /zfs/bob and /media/stick/alice must be renamed atomically

Listing 5.1: Example of renaming two files atomically using a single transaction

```
// create a transaction and add two object sets
VTransaction tx (zfs)
tx.addObjectSet (fat)

// define the on-disk path of the objects that will be modified
const char zfsPath [] = ''/zfs/bob''
const char fatPath [] = ''/media/stick/alice''

// define the object permissions to try applying on open
mask = ACE4_READ_DATA | ACE4_WRITE_DATA | ACE4_READ_NAMED_ATTRS |
ACE4_WRITE_NAMED_ATTRS | ACE4_EXECUTE | ACE4_DELETE

// inform the transaction about the modification of the objects
tx.prepareForChanges (zfs, zfsPath)
tx.prepareForChanges (fat, fatPath)

// the transactional part starts now
tx.begin ()

// open the objects for processing
zfsId = tx.openObject (zfs, File, zfsPath, mask)
fatId = tx.openObject (fat, File, fatPath, mask)

// rename the two objects
tx.renameObject (zfs, zfsId, ''bob'', ''alice'')
tx.renameObject (fat, fatId, ''alice'', ''bob'')
```

```
// continue working...

// close the objects when done
tx.closeObject (zfsId)
tx.closeObject (fatId)

// commit the transaction
tx.commit ()
```

Chapter 6

Development process, time plan, and risk analysis

In this chapter I first talk about the development process that I have followed during my thesis. I then show a time plan with the tasks I have completed and give some details about each task. The chapter ends with an analysis of the risks and a description of the challenges that I have faced during the thesis.

6.1 Development process

The development process I was recommended to use during my thesis is called the Agile Unified Process (AUP) [6]. AUP tries to combine the agility of eXtreme Programming (XP) [128] with the activities found in the Rational Unified Process (RUP) [99]. If I tell you that I have used everything included in AUP during my thesis I will be a big liar. Some things are simply not applicable. For instance, there is nothing to say about project funding since my Master thesis is not funded. Similarly, team working is not possible since a Master thesis is an individual work. Others are not possible. For instance, unit testing cannot be done since although a testing framework for FenixOS has been developed, it has not been integrated into the system yet. Testing is not very important in my case anyway, since my work involves defining rather than implementing the file

system calls. During my thesis I have focused on (1) dividing the project into small iterations, (2) communicating with my supervisor on a weekly basis, (3) doing a risk analysis, (4) refactoring, and (5) respecting the coding standards.

6.2 Time plan

Table 6.1 shows the time plan of my thesis. At each iteration, the goal was to plan accurately only the next two tasks and provide only a rough estimation for the rest. This means that most tasks were overestimated or underestimated [5]. Let me say a few words about each task. In task "Operating system reliability, structure, and performance study" I read papers and articles related with the reliability, structure, and performance of operating systems [23, 49, 52, 53, 54, 95, 118, 119, 121]. The next task, "Legacy file system study", included reading textbook chapters and a paper related with the structure and organisation of the most common file systems [78], [115, chap. 12], [120, chap. 4]. In task "Btrfs design study" I tried to gather information about the design of the Btrfs file system [9, 31, 64, 77, 80, 102]. After this task I made a presentation about the Btrfs design to the rest FenixOS developers, to give them an idea about it. In task "ZFS on-disk format and design study" I studied the design of ZFS [2, 17, 18, 19, 20, 21, 35, 51, 70, 83, 84, 117, 132]. After learning some things about both file systems, I prepared a document describing the major similarities and differences between ZFS and Btrfs. Most if not all of them are discussed in chapter "Overview, performance analysis, and scalability test of ZFS and Btrfs".

At this point we agreed with my supervisor that our next step would be a performance analysis and scalability test between ZFS and Btrfs. However there was a problem: We had to wait for the IT support to take care of our ticket request regarding the preparation of the required hardware for doing the tests. This took a long time, and it was going to last even longer. Thankfully, a system administrator has kindly provided us the hardware at the end, faster than expected. Anyway, I tried to use time efficiently instead of just waiting for the hardware. My first idea was to cooperate with two students who where doing their Bachelor thesis. Their thesis involved a reimplementation of the ZFS stack for FenixOS. But since this is a lot of work, they were focusing on the read parts. All stuff related with writes were left as future work. So the idea was to look into the missing parts. Bad idea actually, since this is my first project related with operating system programming. Understanding the heavily undocumented C code of ZFS proved not to be a good start. At least I revised the code of the students, which involved code writing, bug fixing, and documenting. During this task I found [26] very useful in understanding the on-disk format of ZFS.

After the end of this task I was left without a coding related topic to work on. While searching for a relevant topic, I decided to read some more material about COW and file systems in general [37, 43, 47, 55, 57, 61, 72, 93, 105, 127, 131]. I came up with some ideas to my supervisor, but he came up with a better one: The design of "A file system call interface for FenixOS". You can find the details about my outcomes in the relevant chapter.

Table 6.1: The time plan of my thesis

Task	Deliverable	Duration (weeks)
Operating system reliability, structure, and performance study		1
Legacy file system study		1
Btrfs design study	Presentation	1
ZFS on-disk format and design study		1
High-level ZFS/Btrfs comparison	Text document	1
BSc thesis code contributions	Source code	1
Latest research on COW and file systems study		1
Proper benchmarking study and framework selection		1
ZFS and Btrfs performance analysis and scalability test	Results and evaluation	2
Transactional semantics and modern file permissions study		1
First draft of a transactional VFS	UML design	1
FenixOS file system call interface design	Source code	4
Thesis report and feedback corrections	PDF document	5

While I was ready to start reading about transaction oriented file system call interfaces and modern file permissions, the hardware for doing the performance tests was ready. Before doing the actual tests, I studied about how to approach benchmarking and which frameworks are recommended, and how do they work. This is what task "Proper benchmarking study and framework selection" was about [12, 14, 56, 107, 123, 124, 125]. The next task, "ZFS and Btrfs performance analysis and scalability test" involved executing the tests, collecting the results, and evaluating them. After completing the tests, it was time to start the second part of the thesis. Task "Transactional semantics and modern file permissions study" involved reading papers about transactional memory, transactional semantics, and NFS permissions [8, 10, 11, 74, 91, 96, 106, 114, 126, 130]. After reading some additional papers and book chapters about virtual file systems and memory mapping [22, chap. 12], [63, 75], [79, chap. 5], [104, 113], I came up with a UML class diagram of a transaction oriented file system call interface - "First draft of a transactional VFS". It turned out that the diagram was more complex than it needed to be. The problem was that I included too many ideas. While some of them are important for users, they are not really important for FenixOS at the moment. Examples include arbitrary object attributes [47, chap. 5] and smart pointers [3]. After having a discussion with my supervisor, we decided that I should focus only on the most important things: transactions and file permissions.

Task "FenixOS file system call interface design" involved studying more carefully the semantics of DMU [84, chap. 3], [132], simplifying the UML diagram, defining the system calls presented in chapter "A file system call interface for FenixOS", and making sure that the interface is intuitive and straightforward to use. The final task, "Thesis report and feedback corrections" included writing the report you are reading, and revising it after getting the necessary feedback. At the same time, I tried to polish the system calls.

6.3 Risk analysis

The first thing that I should note is that I am using Agile Risk Management for the needs of my thesis, because it fits well with AUP [81]. In Figure 15.2 of [36, chap. 15] you can see the potential risks of a new computer science/information technology project. Not all of these risks are related with my thesis. For example "Unclear payment schedule" is irrelevant, since my thesis is not funded. A description of the identified risks follows.

From the commercial risks, an ill-defined scope can cause problems and create misunderstandings. We made sure that the scope of the thesis was clearly

defined and revised it during the different tasks.

The visible planning and resource risks were that I personally lack key skills since I have not participated in any operating system programming projects in the past. To minimise this risk we made sure that I had the proper guidance during the prototype implementation part of the thesis. We also ensured that the topic was not unrealistic and too ambitious, because there was a limited amount of time for working on it.

All the risks described above are too abstract and in the rest of this section I will try to be more concrete and identify the specific risks which are directly related with my thesis.

Due to financial and house contract issues, I had to go back at my home country in the middle of July, and continue working on my thesis remotely. My supervisor said that we could do this if we were careful with the communication issues. The problem was that he has not tried it in the past. Therefore, this case involved some risks: Communication issues, remote access to computers, technical assistance. I think that everything worked well, if we exclude the communication part.

It took some time until I could find a programming related topic to work on. I finally found one in the middle of June. The first problem is that the topic should involve some research. It cannot be implementation related only. This is a problem because even if a lot of research is done on file systems, most missing features of COW based file systems where I am focusing involve only implementation.

The fact that I am not an experienced C++ system programmer itself involved many risks. I am interested in operating system design and development. This is the main reason why I approached my supervisor and asked him if we could find a topic as a thesis, assuming that I have no previous operating system programming experience. However, it seems that an MSc thesis assumes that a student is already an expert to the topic he/she selects, and that the result of the thesis should be a contribution to the research community. This is definitely not my case. I am familiar with C and C++, but I have not been involved in kernel development or file system development in the past.

I had spent some time doing things that were not directly related with my thesis. In the beginning I thought that I would have to port ZFS or Btrfs in FenixOS. But this is not considered advanced enough to be presented as the work of an MSc thesis. Then comes the story about adding write support in the ZFS reimplementation that I have already explained you.

I have also wasted some time running irrelevant benchmarks. The problem is that ZFS has metadata compression enabled by default, and deactivating it is not suggested. When I first started running the benchmarks for Btrfs, I enabled compression, assuming that it is metadata compression. I later discovered that Btrfs does not support metadata compression at all. It was the data that I was compressing. And compressed data outputs cannot be compared between ZFS and Btrfs, because they do not support the same compression algorithm. Thus, I had to get the Btrfs results again with compression disabled, and live with the fact that metadata will be compressed only in ZFS.

I had the wrong impression that file systems are a good starting point to approach operating systems, because they usually belong to a higher level, compared to the rest operating system parts: process management, memory management, scheduling, device drivers, etc. The problem is that modern file system code is very complex, because it is trying to solve hard problems, such as reliability, recoverability, easy administration, and scalability, but at the same time needs to perform extremely well and fast.

I had to attend a 3-week course which started in the beginning of June and lasted until the end of June. The course kept me busy from Monday to Friday, 9:00 - 17:00. I tried to do some work for my thesis in the remaining hours, but my productivity was obviously reduced until the end of the course.

Chapter 7

Related work

This chapter is a discussion about the work of other people focusing on (1) analysing the performance and testing the scalability of ZFS and Btrfs, (2) designing transaction oriented system call interfaces.

7.1 Performance analysis and scalability tests

There is a lot of work available which compares Btrfs against competing file systems of GNU/Linux [67, 68] or presents individual file system results [97]. But at the time I have decided to do my tests I could only find two comparisons between Btrfs and ZFS [50, 88]. I cannot say a lot about [88] since there is no analysis and evaluation of the results. As I have noted in a previous chapter, I believe that ZFS is running in user space in [88]. That is why it has very low performance. In [50] ZFS outperforms Btrfs in all tests but sequential write. However, my results have shown that Btrfs has evolved since that time. The micro-benchmarks results have shown that Btrfs outperforms ZFS in sequential writes and rewrites, and performs equally to ZFS in sequential reads. ZFS is better than Btrfs in metadata creation. All these are happening on single disk configurations. On mirroring, ZFS outperforms Btrfs in all tests but sequential output.

Recently more ZFS and Btrfs comparisons have been published [65, 66]. Even if the phoronix results are not the most reliable, since people often criticise the company for publishing black box results without trying to understand them, we can make some observations by analysing the results. In [66], the low performance of ZFS when using FUSE is confirmed. What we can also observe is that in some cases COW based file systems can equally compete or even perform better than popular journaling file systems, like ext4. This means that even if COW based file systems are not the best option to use on a database server (pgbench test), they are ready to serve our everyday needs and offer us all their advanced features at a small performance cost. In my opinion, it does not make sense to use a COW based file system on databases since all good database management systems offer ACID semantics and snapshot support using journaling at good performance. By observing the results of [65], we can also observe that (increased thread count results) COW based file systems can scale better than their competitors.

7.2 Transaction oriented system call interfaces

One of the most interesting system call interfaces is TxOS [96]. TxOS adds support for system transactions on a GNU/Linux system. This means that the kernel data structures and memory buffers are modified to support transactions. Like FenixOS, TxOS uses a strong atomicity model for serialising transactional and non-transactional code [10]. What I most like about TxOS is its simplicity. Programmers just need to enclose the code that they want to be transactional between the `sys_xbegin` and `sys_xend` system calls. That is all. Wrap the code that adds a new user between these two system calls and the code will be protected by ACID. There is no way that a system crash will leave `/etc/passwd` or `/etc/shadow` in an inconsistent state. Either both files will be updated or none of them. I tried to keep the same simplicity when defining the interface of FenixOS. The only difference is that a programmer has to use the various `prepareFor` system calls before executing `begin`. But this is the way DMU works. Other attractive features of TxOS are the usage of COW for supporting ACID and its easy integration with STM and HTM to provide protection of an application's state.

Another effort for providing ACID semantics to the file system is Amino [130]. Amino offers transaction support to user level applications, without changing the POSIX interface of GNU/Linux. Amino is built on top of BDB. It uses `ptrace` to service file system calls and stores all data in a b-tree schema of BDB. Similar to TxOS, Amino simplifies the usage of transactions by providing the basic `begin`, `abort`, and `commit` system calls. As the authors of Amino admit,

the biggest pain of supporting transactions in user space is the high overheads that they generate, especially when data-intensive workloads are used. Such overheads are not acceptable and result in low system performance [114].

A more recent effort of the Amino developers is Valor [114]. Valor is a transactional file interface built as a GNU/Linux kernel module. Valor is designed to run on top of another file system, for example ext3. A fundamental difference between Valor and TxOS is that Valor does not use COW. It facilitates logging instead, by using a separate log partition to keep a record of all the transactional activities. Beware that logging is not the same as journaling. While journaling can protect only metadata and supports transactions which are finite in size and duration, logging provides support for very long transactions. COW has also the same problem with journaling regarding transaction length: While COW can protect both data and metadata, by default it supports transactions provided that they can fit into the main memory of the computer. Logging does not have such a limitation, since the log is kept on the hard disk. Researchers argue that the transaction length limitation of COW can be solved by swapping the uncommitted transaction state to the hard disk or any other secondary storage, but I have not found any implementations which solve this problem yet [96]. Nevertheless, I believe that the benefits of using COW instead of logging are more than the actual disadvantages. For instance, logging has the same disadvantage as journaling: All changes need to be written twice. Once to the log and once to the actual file system. Write ordering is used to ensure that the changes are first written to the log and then to the file system.

The thing that I do not like most about Valor is the complexity of using it. Valor has seven system calls that are tightly connected with its architecture. A programmer needs to understand the system calls and ensure that they are all used in the right order. This is by far more complex than using simple `begin`, `abort`, and `commit` statements. A final thing to note is that Valor can be used only on file systems. It cannot be used as an interface for offering transactions at the operating system level. This is unfortunate, since transactions can be useful not only for file systems, but also for other system parts: processes, memory, networking, signals, etc.

Chapter 8

Conclusions

This is the final chapter of my thesis. It summarises the contents of the thesis, outlines my contributions, and proposes future work.

8.1 Conclusions

Modern hard disks are not perfect. According to disk manufacturers, it is expected that there will be one uncorrectable error every 10 to 20 TB of capacity. Disk capacities in terms of thousand TB are normal for today's data centres. Moreover, since disk capacity is relatively cheap, many personal computer users have already started to use disks with capacities in terms of TB.

The majority of file systems and operating systems that are currently in use cannot offer consistent system updates. Most modern operating systems are using a journaling file system, which is unable to provide data integrity and consistency. Journaling file systems can only protect metadata. Not protecting data is not a design decision of the file system developers. Data are equally or even more important than metadata. It is a limitation of journaling file systems. Since we cannot rely on hardware (disks fail, power failures shutdown systems, etc.) or human (software bugs crash operating systems, failed software upgrades lead to broken systems, etc.), a better solution than journaling is needed.

The most recent solution is called COW. COW provides (meta)data consistency using transactional semantics. Also, it can offer (meta)data integrity using checksums at an acceptable overhead. Not only COW offers (meta)data consistency and integrity, but it also supports the creation of cheap snapshots and clones. Users can take fast online backups without the need to use any special software technique like LVM, or expensive and complex commercial backup software. Everything is taken care by the file system.

ZFS and Btrfs are two modern free software COW based file systems. ZFS has a very interesting architecture and design. It is the first file system which supports many nice features that no other file system ever had. Some examples are its pooled storage, its self-healing properties, and the usage of variable block sizes. In contrast to Btrfs, ZFS uses full ACID semantics to offer transactional support, and is stable enough to be used as the primary file system of OpenSolaris and optionally FreeBSD. Btrfs is trying to build everything around a single concept: the COW friendly b+tree. Unlike ZFS which is designed to be platform independent, Btrfs is reusing as many existing GNU/Linux components as possible. The main goal of Btrfs is to provide to the users all the nice features of COW, but at the same time achieve better performance and scalability than competing GNU/Linux file systems.

The performance analysis between ZFS and Btrfs has shown the strengths and weaknesses of each file system. Because of its simplicity and tight integration with GNU/Linux, Btrfs performs better than ZFS on single disk systems when the caches are bypassed, and as soon as it becomes stable, it seems to be the right choice for all but metadata creation intensive applications on single disks. Btrfs should also be the preferred choice for NFS file servers and applications which rely on creating, reading, and randomly appending files. Note that all these apply only when the caches are bypassed. When the caches are utilised, ZFS seems to outperforms Btrfs in most cases, but I cannot make any conclusions since in my micro-benchmark tests I bypassed the caches. ZFS is the right choice for both single disks and mirroring for applications which use exclusively 64 KB buffering. I believe that when the caches are utilised, ZFS can outperform Btrfs no matter the size of the buffers. ZFS is also more scalable than Btrfs, which means that it is more appropriate to use it on mail servers, database servers, and applications that require synchronous semantics (`fsync`). Since FenixOS focuses on fault tolerance and scalability, ZFS is the best choice for FenixOS at the moment.

With the transition to multi-core and multiple processor systems, offering intuitive concurrent programming models becomes a must. Transactions can be extremely useful in concurrent programming. Only a few examples of using transactions in operating systems are eliminating security vulnerabilities, ensuring file system consistency, and providing an alternative to lock-based pro-

gramming approaches. Not only modern operating systems and file systems should support transactions, but they must also expose them to programmers as a simple API. We use system transactions since no necessary hardware or software is required for supporting them. Furthermore, system transactions can be isolated, rolled back, and execute multiple system calls without violating transactional semantics.

Most UNIX-like operating systems are POSIX compatible. But we believe that the time has come to define a modern transaction oriented, object based system call interface. My work focuses on the file system call interface of FenixOS. I have discussed the most important design decisions of our interface. Our major decisions include using (1) COW to provide transactional semantics, (2) NFSv4.1 ACLs as the only file permission model, (3) memory mapped resources for replacing the legacy `read/write` system calls. I have also presented the file system calls that I have defined and gave an example of how they can be used in practise.

8.2 Future work

Since Btrfs is under heavy development and new features are still added in ZFS, it is important to continue analysing their performance and testing their scalability. It will be interesting to see the updated results of [50]. The author informed me that he is planning to update his results at the end of 2010. Also, I would like to see if all the results of bonnie-64 and Bonnie++ are consistent when using the same file size, since there is a difference in the effective seek rates when the size is changed from 12 GB to 16 GB. I had the chance to test the two file systems only on single disks and RAID 1 but since they support more RAID configurations, it is important to see how they can perform on those configurations. Since Filebench can emulate more applications, like web servers and web proxies, it will be nice to see the results of such tests. In the workload generator test, I used IOzone with a fixed record size of 64 KB and concluded that the variable block size mechanism of ZFS worked better than the default 4 KB block size of Btrfs. It would be also interesting to see what happens if the block size of Btrfs is set to 64 KB and a variable IOzone record size is used.

I have documented the design goals and defined the file system calls of the FenixOS interface, but unfortunately there was no time to start implementing it. It would be a nice experience for me to see how COW, transactions, NFS 4.x ACLs, and memory mapped resources are implemented. Since the interface is minimal, it is likely that I have omitted some necessary system calls. I believe that it will not be hard for the developers who will implement the first file system

for FenixOS to add any required system calls. Finally, my design is relaxed. For example, apart from an `id` and a `type`, a `VObject` does not contain any other information. It might be necessary to add more details about `VObjects`, `VObjectSets`, etc. at the VFS level. Some examples are including the checksum of a `VObject` and the owner of a `VObjectSet`.

Bibliography

[1] Ahrens, M. DMU as general purpose transaction engine? http://www.mail-archive.com/zfs-discuss@opensolaris.org/msg11098.html. Retrieved: 2010-09-08.

[2] Ahrens, M. Is it magic? http://blogs.sun.com/ahrens/entry/is_it_magic. Retrieved: 2010-08-20.

[3] Alexandrescu, A. Smart Pointers in C++. http://www.informit.com/articles/article.aspx?p=31529. Retrieved: 2010-09-10.

[4] Allison, J. Mapping Windows Principals and ACLs to POSIX using Samba. http://www.citi.umich.edu/projects/nfsv4/jallison-acl-mapping/img0.html. Retrieved: 2010-09-06.

[5] Ambler, S. W. Agile Project Planning Tips. http://www.ambysoft.com/essays/agileProjectPlanning.html. Retrieved: 2010-09-09.

[6] Ambler, S. W. The Agile Unified Process (AUP). http://www.ambysoft.com/unifiedprocess/agileUP.html. Retrieved: 2010-09-09.

[7] Ambler, S. W. UML 2 Class Diagrams. http://www.agilemodeling.com/artifacts/classDiagram.htm. Retrieved: 2010-09-07.

[8] Amsterdam, J. Atomic File Transactions, Part 1. http://onjava.com/pub/a/onjava/2001/11/07/atomic.html. Retrieved: 2010-08-21.

[9] Aurora, V. A short history of btrfs. http://lwn.net/Articles/342892/. Retrieved: 2010-08-22.

[10] Blundell, C., Lewis, E. C., and Martin, M. M. Deconstructing Transactional Semantics: The Subtleties of Atomicity. In *Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD)* (2005), pp. 48–55.

[11] Blundell, C., Lewis, E. C., and Martin, M. M. K. Unrestricted Transactional Memory: Supporting I/O and System Calls within Transactions. Technical Report TR-CIS-06-09, Department of Computer and Information Science, University of Pennsylvania, 2006.

[12] bonnie-64. `http://code.google.com/p/bonnie-64/`. Retrieved: 2010-08-26.

[13] Command-line access. `http://code.google.com/p/bonnie-64/source/checkout`. Retrieved: 2010-08-26.

[14] Bonnie++ now at 1.03e (last version before 2.0)! `http://www.coker.com.au/bonnie++/`. Retrieved: 2010-08-26.

[15] Bonnie. `http://www.textuality.com/bonnie/`. Retrieved: 2010-08-27.

[16] textuality - Bonnie - introduction. `http://www.textuality.com/bonnie/advice.html`. Retrieved: 2010-08-31.

[17] Bonwick, J. RAID-Z. `http://blogs.sun.com/bonwick/entry/raid_z`. Retrieved: 2010-08-24.

[18] Bonwick, J. Smokin' Mirrors. `http://blogs.sun.com/bonwick/entry/smokin_mirrors`. Retrieved: 2010-08-24.

[19] Bonwick, J. Space Maps. `http://blogs.sun.com/bonwick/entry/space_maps`. Retrieved: 2010-09-09.

[20] Bonwick, J. ZFS Block Allocation. `http://blogs.sun.com/bonwick/entry/zfs_block_allocation`. Retrieved: 2010-09-09.

[21] Bonwick, J. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1* (1994), pp. 6–6.

[22] Bovet, D. P., and Cesati, M. *Understanding the Linux Kernel*. O'Reilly Media, 2000.

[23] Boyd-Wickizer, S., Chen, H., Chen, R., Mao, Y., Kaashoek, F., Morris, R., Pesterev, A., Stein, L., Wu, M., Dai, Y., Zhang, Y., and Zhang, Z. Corey: An Operating System for Many Cores. In *8th USENIX Symposium on Operating Systems Design and Implementation* (2008), pp. 43–58.

[24] Bray, T. Bonnie 64. `http://www.tbray.org/ongoing/When/200x/2004/11/16/Bonnie64`. Retrieved: 2010-08-26.

[25] Breed, G. Bit Error Rate: Fundamental Concepts and Measurement Issues. *High Frequency Electronics* (2003), 46–48.

[26] Bruning, M. ZFS On-Disk Data Walk (or: Where's my Data?). `http://www.bruningsystems.com/osdevcon_draft3.pdf`. Retrieved: 2010-09-10.

[27] Btrfs wiki - Main Page. `https://btrfs.wiki.kernel.org/index.php/Main_Page`. Retrieved: 2010-08-23.

[28] 2.7 What checksum function does Btrfs use? `https://btrfs.wiki.kernel.org/index.php/FAQ#What_checksum_function_does_Btrfs_use.3F`. Retrieved: 2010-08-25.

[29] Basic Filesystem Commands. `https://btrfs.wiki.kernel.org/index.php/Getting_started#Basic_Filesystem_Commands`. Retrieved: 2010-08-25.

[30] Copy on Write Logging. `https://btrfs.wiki.kernel.org/index.php/Btrfs_design#Copy_on_Write_Logging`. Retrieved: 2010-08-25.

[31] Btrfs design. `https://btrfs.wiki.kernel.org/index.php/Btrfs_design`. Retrieved: 2010-08-25.

[32] btrfs kernel module Git Repository. `https://btrfs.wiki.kernel.org/index.php/Btrfs_source_repositories#btrfs_kernel_module_Git_Repository`. Retrieved: 2010-08-26.

[33] Using Btrfs with Multiple Devices. `https://btrfs.wiki.kernel.org/index.php/Using_Btrfs_with_Multiple_Devices`. Retrieved: 2010-08-25.

[34] Resizing partitions (shrink/grow). `https://btrfs.wiki.kernel.org/index.php/FAQ#Resizing_partitions_.28shrink.2Fgrow.29`. Retrieved: 2010-08-25.

[35] Insights into ZFS - today: The nature of writing things. `http://www.c0t0d0s0.org/archives/5343-Insights-into-ZFS-today-The-nature-of-writing-things.html`. Retrieved: 2010-09-09.

[36] Cadle, J., and Yeates, D. *Project Management for Information Systems*. Pearson Prentice Hall, 2008.

[37] Carvajal, J. M. C., Garnier, J.-C., Neufeld, N., and Schwemmer, R. A High-Performance Storage System for the LHCb Experiment. *IEEE Transactions on Nuclear Science 57* (2010), 658–662.

[38] Christoffer, M., and Oppegaard, A. Evaluation of Performance and Space Utilisation When Using Snapshots in the ZFS and Hammer File Systems. Master's thesis, University of Oslo, 2009.

[39] Coker, R. Bonnie++ 1.96. `http://www.coker.com.au/bonnie++/experimental/`. Retrieved: 2010-08-30.

[40] Coker, R. Bonnie++ Documentation. `http://www.coker.com.au/bonnie++/readme.html`. Retrieved: 2010-08-27.

[41] Supporting transactions in btrfs. `http://lwn.net/Articles/361457/`. Retrieved: 2010-08-25.

[42] Dawidek, P. J. Porting the ZFS File System to the FreeBSD Operating System. *AsiaBSDCon 2007* (2007), 97–103.

[43] Dillon, M. The HAMMER Filesystem. `http://www.dragonflybsd.org/hammer/hammer.pdf`. Retrieved: 2010-08-22.

[44] Falkner, S., and Week, L. NFSv4 ACLs: Where are we going? `http://www.connectathon.org/talks06/falkner-week.pdf`. Retrieved: 2010-09-06.

[45] FSL's Filebench Linux/FreeBSD Port. `http://www.fsl.cs.sunysb.edu/~vass/filebench/`. Retrieved: 2010-08-31.

[46] Filesystem in Userspace. `http://fuse.sourceforge.net/`. Retrieved: 2010-08-28.

[47] Giampaolo, D. *Practical File System Design with the Be File System.* Morgan Kaufmann, 1999.

[48] Various Licenses and Comments about Them. `http://www.gnu.org/licenses/license-list.html#GPLIncompatibleLicenses`. Retrieved: 2010-08-25.

[49] Härtig, H., Hohmuth, M., Liedtke, J., Wolter, J., and SchHönberg, S. The performance of μ-kernel-based systems. In *ACM Symposium on Operating Systems Principles* (1997), pp. 66–77.

[50] Heger, D. A. Workload Dependent Performance Evaluation of the Btrfs and ZFS Filesystems. `http://www.dhtusa.com/media/IOPerf_CMG09DHT.pdf`. Retrieved: 2010-08-24.

[51] Henson, V., Ahrens, M., and Bonwick, J. Automatic Performance Tuning in the Zettabyte File System. USENIX Conference on File and Storage Technologies (FAST). Work in progress report.

[52] Herder, J. N., Bos, H., Gras, B., Homburg, P., and Tanenbaum, A. S. Reorganizing UNIX for reliability. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (2006), 81–94.

[53] Herder, J. N., Bos, H., Gras, B., Homburg, P., and Tanenbaum, A. S. Fault isolation for device drivers. In *2009 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2009* (2009), pp. 33–42.

[54] Herder, J. N., van Moolenbroek, D. C., Appuswamy, R., Gras, B., and Tanenbaum, A. S. Dealing with Driver Failures in the Storage Stack. In *2009 4th Latin-American Symposium on Dependable Computing, LADC 2009* (2009), pp. 119–126.

[55] Hitz, D., Lau, J., and Malcolm, M. File system design for an NFS file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference* (1994), pp. 19–19.

[56] IOzone Filesystem Benchmark. `http://www.iozone.org/`. Retrieved: 2010-08-26.

[57] Jain, S., Shafique, F., Djeric, V., and Goel, A. Application-level Isolation and Recovery with Solitude. *ACM SIGOPS Operating Systems Review 42* (2008), 95–107.

[58] Katcher, J. PostMark: A New File System Benchmark. Technical Report TR3022, Network Appliance, 1997.

[59] Kath, R. Managing Memory-Mapped Files. `http://msdn.microsoft.com/en-us/library/ms810613.aspx`. Retrieved: 2010-09-07.

[60] Kernighan, B. W., and Ritchie, D. M. *The C Programming Language, Second Edition*. Prentice Hall, 1988.

[61] Kiselyov, O., and chieh Shan, C. Delimited continuations in operating systems. In *Proceedings of the 6th international and interdisciplinary conference on Modeling and using context* (2007), pp. 291–302.

[62] Kiwi, K. H. Logical volume management. `http://www.ibm.com/developerworks/linux/library/l-lvm2/`. Retrieved: 2010-08-20.

[63] Kleiman, S. Vnodes: an architecture for multiple file system types in Sun UNIX. In *USENIX Association Summer Conference Proceedings, Atlanta 1986* (1986).

[64] Klosowski, P. B-Tree File System. http://dclug.tux.org/200908/BTRFS-DCLUG.pdf. Retrieved: 2010-08-24.

[65] Larabel, M. Benchmarking ZFS On FreeBSD vs. EXT4 & Btrfs On Linux. http://www.phoronix.com/scan.php?page=article&item=zfs_ext4_btrfs&num=1. Retrieved: 2010-09-10.

[66] Larabel, M. Benchmarks Of ZFS-FUSE On Linux Against EXT4, Btrfs. http://www.phoronix.com/scan.php?page=article&item=zfs_fuse_performance&num=1. Retrieved: 2010-09-10.

[67] Larabel, M. EXT4 & Btrfs Regressions In Linux 2.6.36. http://www.phoronix.com/scan.php?page=article&item=linux_2636_btrfs&num=1. Retrieved: 2010-09-10.

[68] Larabel, M. File-System Benchmarks With The Linux 2.6.34 Kernel. http://www.phoronix.com/scan.php?page=article&item=linux_2634_fs&num=1. Retrieved: 2010-09-10.

[69] Lavigne, D. An Introduction to Unix Permissions. http://onlamp.com/pub/a/bsd/2000/09/06/FreeBSD_Basics.html?page=1. Retrieved: 2010-09-06.

[70] Leal, M. ZFS Internals (part 9). http://www.eall.com.br/blog/?p=1498. Retrieved: 2010-08-24.

[71] Leidinger, A. ZFS Tuning Guide. http://wiki.freebsd.org/ZFSTuningGuide. Retrieved: 2010-09-16.

[72] Levrinc, R. LLFS A Copy-On-Write File System For Linux. Master's thesis, Vienna University Of Technology, 2008.

[73] ACLs. http://wiki.linux-nfs.org/wiki/index.php/ACLs. Retrieved: 2010-09-06.

[74] Liu, Y., Zhang, X., Li, H., Li, M., and Qian, D. Hardware Transactional Memory Supporting I/O Operations within Transactions. In *Proceedings of the 2008 10th IEEE International Conference on High Performance Computing and Communications* (2008).

[75] Martin, B. Notes on libferris. http://witme.sourceforge.net/libferris.web/FerrisNotes/index.php. Retrieved: 2010-09-10.

[76] Martin, B. Using Bonnie++ for filesystem performance benchmarking. http://www.linux.com/archive/feature/139742. Retrieved: 2010-08-27.

[77] Interview: Chris Mason about Btrfs. http://liquidat.wordpress.com/2007/08/07/interview-chris-mason-about-btrfs/. Retrieved: 2010-08-25.

[78] McKusick, M. K., Joy, W. N., Leffler, S. J., and Fabry, R. S. A fast file system for UNIX. *ACM Transactions on Computer Systems (TOCS) 2* (1984), 181–197.

[79] McKusick, M. K., and Neville-Neil, G. V. *The Design and Implementation of the FreeBSD Operating System*. Addison Wesley, 2004.

[80] McPherson, A. A Conversation with Chris Mason on BTRfs: the next generation file system for Linux. http://www.linux-foundation.org/weblogs/amanda/2009/06/22/a-conversation-with-chris-mason-on-btrfs-the-next\-generation-file-system-for-linux. Retrieved: 2010-08-25.

[81] Mhala, A. Agile Risk Management. http://it.toolbox.com/wiki/index.php/Agile_Risk_Management. Retrieved: 2010-09-10.

[82] HOW TO: Install Interix. http://support.microsoft.com/kb/324081. Retrieved: 2010-10-01.

[83] Microsystems, S. *Solaris ZFS Administration Guide*. Sun Microsystems, Inc., 2009.

[84] Microsystems, S. *ZFS On-Disk Specification*. Sun Microsystems, Inc., 2009.

[85] mmap - map pages of memory. http://opengroup.org/onlinepubs/000095399/functions/mmap.html. Retrieved: 2010-09-07.

[86] Backup of MySQL Databases on Logical Volumes. http://wiki.zmanda.com/index.php/Backup_of_MySQL_Databases_on_Logical_Volumes. Retrieved: 2010-08-20.

[87] Napierala, E. T. NFSv4_ACLs. http://wiki.freebsd.org/NFSv4_ACLs. Retrieved: 2010-09-06.

[88] Bonnie++ V1.03c Benchmark results. http://nerdbynature.de/benchmarks/v40z/2010-02-03/bonnie.html. Retrieved: 2010-08-28.

[89] Norcott, W. D., and Capps, D. Iozone Filesystem Benchmark. http://www.iozone.org/docs/IOzone_msword_98.pdf. Retrieved: 2010-09-01.

[90] OpenSolaris Download Center. http://hub.opensolaris.org/bin/view/Main/downloads. Retrieved: 2010-08-26.

[91] Oracle. *Oracle Berkeley DB. Getting Started with Transaction Processing for C++. 11g Release 2*. Oracle, 2010.

[92] Hybrid Kernel. `http://wiki.osdev.org/Hybrid_Kernel`. Retrieved: 2010-08-14.

[93] Peterson, Z. N. J., and Burns, R. Ext3cow: A Time-Shifting File System for Regulatory Compliance. *ACM Transactions on Storage (TOS) 1* (2005), 190–212.

[94] Philipp, M. The Byte Converter. `http://webdeveloper.earthweb.com/repository/javascripts/2001/04/41291/byteconverter.htm`. Retrieved: 2010-09-01.

[95] Pike, R., Presotto, D., Dorward, S., Flandrena, B., Thompson, K., Trickey, H., and Winterbottom, P. Plan 9 from Bell Labs. `http://plan9.bell-labs.com/sys/doc/9.html`. Retrieved: 2010-08-14.

[96] Porter, D. E., Hofmann, O. S., Rossbach, C. J., Benn, A., and Witchel, E. Operating System Transactions. In *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles* (2009), pp. 161–176.

[97] Pratt, S. 2.6.35 performance results. `http://btrfs.boxacle.net/repository/raid/perftest/perfpatch/perfpatch.html`. Retrieved: 2010-09-10.

[98] Prechelt, L. An Empirical Comparison of Seven Programming Languages. *ACM Computer 33* (2000), 23–29.

[99] Rational Unified Process. Best Practices for Software Development Teams. `http://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251_bestpractices_TP026B.pdf`. Retrieved: 2010-09-09.

[100] Raymond, E. S. *The cathedral & the bazaar*. O'Reilly, 2001.

[101] Rockwood, B. ZFS performance issue - READ is slow as hell... `http://www.mail-archive.com/perf-discuss@opensolaris.org/msg02034.html`. Retrieved: 2010-10-01.

[102] Rodeh, O. B-trees, shadowing, and clones. *ACM Transactions on Storage (TOS) 3* (2008), 2.

[103] Rogers, B. btrfs stuff. `https://launchpad.net/~brian-rogers/+archive/btrfs`. Retrieved: 2010-10-01.

[104] Schandl, B., and Haslhofer, B. The Sile Model – A Semantic File System Infrastructure for the Desktop. In *Proceedings of the 6th European Semantic Web Conference on The Semantic Web: Research and Applications* (2009).

[105] Shaull, R., Shrira, L., and Hao, X. Skippy: A new snapshot indexing method for time travel in the storage manager. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2008), pp. 637–648.

[106] Shepler, S., Eisler, M., and Noveck, D. Network File System (NFS) Version 4 Minor Version 1 Protocol. `http://tools.ietf.org/search/rfc5661`. Retrieved: 2010-09-06.

[107] FileBench. `http://www.solarisinternals.com/wiki/index.php/FileBench`. Retrieved: 2010-08-26.

[108] Example Comparison of the multistreamread.f workload. `http://hub.opensolaris.org/bin/view/Community+Group+performance/example_comparison`. Retrieved: 2010-08-31.

[109] Recommended parameters. `http://www.solarisinternals.com/wiki/index.php/FileBench#Recommended_parameters`. Retrieved: 2010-08-31.

[110] Application Emulation. `http://www.solarisinternals.com/wiki/index.php/FileBench#Application_Emulation`. Retrieved: 2010-08-31.

[111] FileBench Features. `http://www.solarisinternals.com/wiki/index.php/FileBench#Filebench_Features`. Retrieved: 2010-08-31.

[112] Running interactively with go_filebench. `http://www.solarisinternals.com/wiki/index.php/FileBench#Running_interactively_with_go_filebench`. Retrieved: 2010-08-31.

[113] Song, Y., Choi, Y., Lee, H., Kim, D., and Park, D. Searchable Virtual File System: Toward an Intelligent Ubiquitous Storage. *Lecture Notes in Computer Science 3947/2006* (2006), 395–404.

[114] Spillane, R., Gaikwad, S., Chinni, M., Zadok, E., and Wright, C. P. Enabling transactional file access via lightweight kernel extensions. In *Proceedings of the 7th conference on File and storage technologies* (2009), pp. 29–42.

[115] Stallings, W. *Operating Systems: Internals and Design Principles, Fifth Edition*. Prentice Hall, 2004.

[116] Stallman, R. M. *Free Software Free Society*. GNU Press, 2002.

[117] Stanik, J. A conversation with Jeff Bonwick and Bill Moore. *ACM Queue 5* (2007), 13–19.

[118] Swift, M. M., Bershad, B. N., and Levy, H. M. Improving the reliability of commodity operating systems. In *ACM Symposium on Operating Systems Principles* (2003), pp. 207–222.

[119] Tanenbaum, A. S. Tanenbaum-Torvalds Debate: Part II. `http://www.cs.vu.nl/~ast/reliable-os/`. Retrieved: 2010-08-13.

[120] Tanenbaum, A. S. *Modern Operating Systems 3e*. Pearson Prentice Hall, 2008.

[121] Tanenbaum, A. S., Herder, J. N., and Bos, H. Can We Make Operating Systems Reliable and Secure? *IEEE Explore 39* (2006), 44–51.

[122] Tanenbaum, A. S., and Woodhull, A. S. *Operating Systems Design and Implementation, Third Edition*. Prentice Hall, 2006.

[123] Traeger, A., Joukov, N., Wright, C. P., and Zadok, E. The FSLer's Guide to File System and Storage Benchmarking . `http://www.fsl.cs.sunysb.edu/docs/fsbench/checklist.html`. Retrieved: 2010-08-25.

[124] Traeger, A., Zadok, E., Joukov, N., and Wright, C. P. A nine year study of file system and storage benchmarking. *ACM Transactions on Storage (TOS) 4* (2008), 5–56.

[125] Traeger, A., Zadok, E., Miller, E. L., and Long, D. D. Findings from the First Annual Storage and File Systems Benchmarking Workshop. In *Storage and File Systems Benchmarking Workshop* (2008), pp. 113–117.

[126] Volos, H., Tack, A. J., Goyal, N., Swift, M. M., and Welc, A. xCalls: safe I/O in memory transactions. In *Proceedings of the 4th ACM European conference on Computer systems* (2009).

[127] Wang, Z., Feng, D., Zhou, K., and Wang, F. PCOW: Pipelining-based COW snapshot method to decrease first write penalty. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (2008), 266–274.

[128] Wells, D. The Rules of Extreme Programming. `http://www.extremeprogramming.org/rules.html`. Retrieved: 2010-09-09.

[129] Checksums. `http://www.wireshark.org/docs/wsug_html_chunked/ChAdvChecksums.html`. Retrieved: 2010-08-22.

[130] Wright, C. P., Spillane, R., Sivathanu, G., and Zadok, E. Extending ACID semantics to the file system. *ACM Transactions on Storage (TOS) 3* (2007), 1–42.

[131] Xiao, W., and Yang, Q. Can We Really Recover Data if Storage Subsystem Fails? In *2008 28th IEEE International Conference on Distributed Computing Systems (ICDCS)* (2008), pp. 597–604.

[132] ZFS Source Tour. `http://hub.opensolaris.org/bin/view/Community+Group+zfs/source`. Retrieved: 2010-08-24.