Migration of WAFL to BSD

by

Sreelatha S. Reddy

B.E., College of Engineering, Pune, India, 1998

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
Master of Science
in
THE FACULTY OF GRADUATE STUDIES
(Department of Computer Science)

We accept this thesis as conforming
to the required standard

The University of British Columbia

December 2000

© Sreelatha S. Reddy, 2000
In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia

Vancouver, Canada

Date 20th December, 2000
Abstract

The UNIX kernel has been around for quite some time. Considering this time factor, we realise that compared to the other components of the operating system, the filesystem has yet to see the implementation of novel and innovative features. However, filesystem research has continued for other kernels.

The Write Anywhere File Layout filesystem developed at Network Appliance has been proved to be a reliable and new filesystem with its ability to take snapshots for backups and to schedule consistency points for fast recovery. WAFL has been built as a filesystem component in the ONTAP kernel which runs on the NFS filer.

WAFL has been optimized for the ONTAP environment. The ONTAP kernel differs considerably from the FreeBSD kernel. ONTAP is an appliance operating system optimized for fast NFS service while the FreeBSD kernel is a general-purpose operating system. There exist architectural and semantic distinctions in the design of the two kernels. This thesis was essentially an attempt to bridge these differences by integrating WAFL into FreeBSD. We focus more on the design issues and the problems encountered during the migration of WAFL from its old environment to its new environment, rather than performance.

By introducing the ONTAP WAFL to FreeBSD, we are interested in knowing how WAFL would fare in the BSD kernel given that the BSD and ONTAP kernels are quite different. We also want to see how WAFL performs as compared to FFS, the local filesystem in the FreeBSD kernel.

In this thesis, we describe the design and implementation issues encountered while porting the WAFL filesystem to the BSD kernel. The implementation has been carried out in the current version of the FreeBSD 4.0 kernel.
Contents

Abstract ii

Contents iii

List of Tables vi

List of Figures vii

Acknowledgements viii

Dedication ix

1 Introduction 1

1.1 Overview .................................................. 3
1.2 Motivation ................................................ 3
1.3 Methodology ............................................. 4
1.4 Synopsis .................................................. 5
1.5 Related Work ............................................. 6
  1.5.1 Filesystem design similar to WAFL ...................... 6
  1.5.2 Specialized filesystems .................................. 7
  1.5.3 Swarm ................................................ 8
  1.5.4 Porting the SGI XFS filesystem to Linux ................. 9
  1.5.5 MS-DOS on BSD ...................................... 9
2 Background

2.1 FreeBSD: an operating system for the PC
   2.1.1 Introduction
   2.1.2 VFS layer
   2.1.3 BSD buffer cache
   2.1.4 Sleep/Wakeup Model
   2.1.5 Resource management in the UNIX kernel
   2.1.6 UNIX filesystems

2.2 ONTAP: an operating system for an appliance
   2.2.1 Introduction
   2.2.2 Buffer Cache
   2.2.3 Suspend/Restart Model
   2.2.4 WAFL

3 Discussion: WAFL Vs. FFS

3.1 BSD FFS
   3.1.1 Write Allocation in FFS
   3.1.2 Synchronous and Asynchronous writes
   3.1.3 Inode structure

3.2 WAFL
   3.2.1 Inode structure
   3.2.2 Interaction with the buffer cache
   3.2.3 Write allocation
   3.2.4 Meta-data

4 Design Issues and Implementation

4.1 Prototype
   4.1.1 Mapping between Vnode routines and WAFL routines
4.1.2 Details on routines implemented .................................................. 31
4.1.3 WAFL-BSD buffer cache ................................................................. 32
4.1.4 Mapping between WAFL buffers and BSD buffers ......................... 34
4.1.5 Basic Reads and Writes ................................................................. 35
4.1.6 Semantic Difference ...................................................................... 36
4.1.7 Initial Design .................................................................................. 37
4.1.8 Final Design .................................................................................. 40
4.1.9 Callbacks ...................................................................................... 42
4.1.10 Allocating and Scavenging buffers .............................................. 42
4.1.11 Consistency Points ....................................................................... 43
4.1.12 Snapshot Creation and Deletion .................................................. 43
4.1.13 Implementing suspend/restart ..................................................... 44
4.1.14 Support for Multiple Volumes ..................................................... 45

5 Performance ......................................................................................... 46
5.1 Vinum ............................................................................................... 46
5.2 Experimental Setup ........................................................................... 47
  5.2.1 Andrew Benchmark ..................................................................... 47
  5.2.2 Micro-benchmark ......................................................................... 48
  5.2.3 PostMark .................................................................................... 49
5.3 Summary .......................................................................................... 52

6 Conclusion and Future Work ................................................................ 53
6.1 Conclusion ....................................................................................... 53
6.2 Future Work ..................................................................................... 54

Bibliography ........................................................................................... 56
List of Tables

5.1 Andrew Benchmark Results .............................................. 47
5.2 Measurement of a recursive delete. .................................... 48
5.3 Basic operations on 1000 "size exactly 4KB" files ................ 48
5.4 Operations on large files ............................................... 49
5.5 Postmark Results: 500 files, 500 transactions ...................... 52
5.6 Postmark Results: 1000 files, 1000 transactions ................... 52
## List of Figures

1.1 WAFL: before and after ................................................. 5

2.1 FreeBSD kernel .......................................................... 11
2.2 Vnode layer in the kernel .............................................. 12
2.3 Format of the buffer header .......................................... 14
2.4 Buffer queues ............................................................ 15
2.5 Sleep/Wakeup model .................................................... 17
2.6 Data ONTAP operating system architecture ....................... 19

3.1 Inode structure in FFS .................................................. 24
3.2 Buffer tree in WAFL ..................................................... 26

4.1 Interaction with the VFS layer ....................................... 30
4.2 WAFL and FFS in FreeBSD ........................................... 33
4.3 Final layout .............................................................. 34
4.4 Modified WAFL buffer ................................................ 35
4.5 Various states of WAFL buffers ...................................... 41

5.1 Read performance ....................................................... 50
5.2 Creation of 1000 files ................................................ 51
Acknowledgements

My deepest thanks to Norm, Mike, my family, my DSG group and friends.

Sreelatha S. Reddy

The University of British Columbia
December 2000
In loving memory of my Ammamma.
Chapter 1

Introduction

The main purpose of a computer is to create, manipulate, store and retrieve data. A filesystem provides the machinery to support these tasks. Suitable design and implementation of this key component is very important. In general, systems are more reliable if they have to deal with less functionality. Thus, before designing filesystems, it is very essential to think how they will be used and how they are expected to interact with the neighbouring systems. We are entering an era of computing that is being dominated by specialized and easy-to-use devices that avoid the complexities of the PC. The single, monolithic PC operating system has been followed by a microkernel OS, distributed OS and now, an appliance OS. There is already a widespread use of appliances in our day to day use. We have microprocessors in most devices ranging from cell-phones to microwaves.

The first network appliances were print servers, mail servers and web servers. As networks are growing faster by the day, the need arises for the end-boxes to be fast too. CISCO [Netw] recognized this need and came up with the idea of building specialized devices which can perform the routing function. Traditional routers have evolved into highly specialized computing platforms. Thus, instead of having a general-purpose computer perform the routing function, why not optimize a device to perform a specific function? Another example of an appliance is Raw Iron (Oracle) [PR N98]. Raw Iron is a DBMS in a box. The box comes with the ORACLE DBMS,
running on top of a minimal version of Solaris, and is network-ready.

Operating systems are built to support a general-purpose environment on which you can run and build anything you want. The UNIX operating system has been designed and built to handle many features. It supports multiple users, virtual memory, graphical user-interfaces, etc. But if you know what exactly you want to run, you can strip away a lot of the generality from the underlying environment and customize it for a specific operation. Following this pattern, an appliance is built by stripping away all the general-purpose features and optimizing the performance for a specific purpose (routing, data storage etc.). Thus, in keeping with the appliance culture, the current trend is to replace general-purpose computers with appliances.

In addition to routers, nowadays we have dedicated NFS servers or filers. These filers are doing wonders to the data storage market. The server is now the bottleneck between the network and the users. Hence, the use of appliances becomes important simply because they are simple, reliable and effective. Filers concentrate on file management exclusively unlike their general-purpose counterparts.

The NetApp filer [Karl] is one such filer which has been designed solely to handle NFS requests. Since a NFS server typically receives more write requests than read requests, write performance is optimized in the file server. NetApp filers have demonstrated that they can deliver extremely fast access times, faster than a general-purpose server. In addition, they have also been very easy to administer. Thus, the simple design resulted in less code, and code that is easier to test and support.

The file system on the NFS filer is WAFL (Write Anywhere File Layout) [hitz94], a filesystem that grows with the addition of more disks. The filer consists of a light-weight, real-time kernel running on standard Intel-based or Compaq Alpha hardware. The filer works on top of RAID, thus increasing disk availability and protecting against disk loss. WAFL has been built to work and exploit the RAID layer beneath it, thus optimizing the write performance.

Thus, today we have these two very distinct types of computers, the appliance and the general purpose computer. As we discussed earlier, the environments in an appliance and a general
purpose computer differ greatly. Everything in an appliance has been designed to serve a specific purpose. On the other hand, in a PC running UNIX, the system has been designed to cater to many services thus making it a general-purpose system. The big question is what will happen if we take a module from an appliance and try and fit it in a general purpose environment. If a particular module performs well in an appliance environment, the interesting issue is to see how it would fare in a general-purpose environment. If there is a performance degradation, was this caused by the environment or was it some other factor? These are some of the interesting issues that we plan to answer during the course of this thesis. More specifically, it would be interesting to see how WAFL would perform when compared to the UNIX Fast File System.

1.1 Overview

To summarize, on one hand we have WAFL which is closely knit with its operating system ONTAP, and optimized to perform well on a network box. On the other hand, we have WAFL fitted in a general-purpose environment, trying to cope with the demands of a general-purpose environment.

WAFL has been designed to work specifically in a NFS appliance. It was built primarily for fast NFS file service, to support large file systems, to optimize write performance using RAID and to restart quickly even in case of a system crash or power failure. By using WAFL in BSD, we hope to solve some of the problems faced by BSD, like long rebooting times and frequent crashes.

1.2 Motivation

The motivation for this thesis has been to see where exactly the performance gain lies. Would the specialized design of WAFL be to our benefit in FreeBSD?

A major problem with the UNIX filesystems is the time taken by fsck to perform a consistency check in the event of an unclean shutdown. Though storage medium failure is rare, system crashes caused by software bugs or transient hardware problems are more frequent. Hence, it is essential that the recovery of the system be much faster.
Our goal is to take two different architectures and make them compatible, i.e., check the compatibility of an appliance oriented file system in a general-purpose architecture.

WAFL is one of the few filesystems with snapshot capabilities. During a snapshot, top-level inodes are replicated to provide an entry point into the hierarchical structure. This results in a large number of entry points within the filesystem, each representing the state of the filesystem at the time when the snapshot was taken. This assures backup of the entire filesystem, thus easing retrieval of old or deleted files, just like undelete but even better. With the cost of backups being high and restorations proving to be slow (especially restoring from tape devices), the snapshot ability proves to be useful. Such an ability is not implemented in any of the file systems available for UNIX.

All these features are provided by WAFL. At what cost can we have the same features in BSD? How would it affect the performance? How important is architectural compatibility? These are a few of the questions we plan to answer during the course of this thesis.

1.3 Methodology

FreeBSD is an advanced UNIX operating system developed for PC compatible computers. It runs on computer systems based on the Intel x86 architecture and also the DEC Alpha architecture. FreeBSD is a 32-bit operating system and includes the following features

- Pre-emptive multitasking
- Multi-user facilities
- TCP/IP networking
- Memory protection
- Demand-paged virtual memory
FreeBSD has always provided a fertile ground for the creation of various filesystems. Well-known for its performance and reliability, it is being used in internet services, education, research, networking and software development. Therefore, BSD was an ideal choice for a general-purpose environment.

![Diagram showing FreeBSD,ONTAP, WAFL, RAID, and Disk]

Figure 1.1: WAFL: before and after

The entire filesystem code for WAFL has been ported to FreeBSD in such a way, that WAFL interacts with the operating system just as FFS interacts with the operating system. WAFL interacts with the VFS layer and the buffer cache layer, similar to the FFS interaction with these two layers.

1.4 Synopsis

In the following chapters, we present the issues related to porting WAFL to the FreeBSD system and explain the design decisions we made. Chapter 2 is an in-depth explanation about the workings of the BSD kernel, the UNIX filesystems and the ONTAP environment on the filer. Chapter 3 discusses the design of the FFS filesystem and the WAFL filesystem. Chapter 4 explains the design issues encountered during implementation. It also presents the implementation of WAFL in the BSD kernel. We evaluate the performance of WAFL integrated in a general-purpose environment in Chapter 5 and finally present our conclusions in Chapter 6 along with a description of further work needed.
1.5 Related Work

This section looks at prior work related to this thesis.

1.5.1 Filesystem design similar to WAFL

There are many filesystems that share some common features with WAFL.

Logging/Journaling filesystems

Filesystems update their metadata using synchronous writes. Each metadata update may involve many writes. Thus, if the system crashes during a write sequence, the filesystem may be in an inconsistent state. Normally, fsck has to examine and repair all the meta-data structures. This might take a long time on large file systems. Thus, journaling is used to avoid corruption. Hence, in some cases a log is used to store metadata writes. If the system crashes, information is replayed from the log to complete the operation. Thus, an entire scan of the filesystem is avoided.

Episode

Episode [chut92] shares a few goals with WAFL. It has been designed to utilize disk bandwidth efficiently and to scale with increases in disk capacity. Just like WAFL, it logs meta-data to obtain good performance and to restart quickly after an unclean shutdown. It runs as a stand-alone filesystem as well as a distributed filesystem. It also supports the concept of a volume, which separates disk block storage from logical file systems. Thus, a pool of disks can provide storage to many filesystems at the same time. Episode uses a layered architecture and a generalization of files called containers to implement filesets. Thus, a fileset is said to be a logical filesystem which represents a connected subtree. One administrative mechanism called fileset cloning is similar to the snapshots in WAFL. A fileset clone is a read-only fileset containing a snapshot of a normal fileset, and it shares data with the original fileset using copy-on-write techniques. Clones are created very quickly without blocking access to the current fileset. The logging techniques used to keep
filesystem meta-data consistent are similar to the logging techniques used in databases.

Episode does not guarantee consistency of user-data in the event of a system crash as the consistency of the filesystem depends only on the consistency of the meta-data. As a result, Episode performs meta-data update operations significantly faster than the Berkeley Fast File System.

Other similar filesystems

Listed below are a few filesystem that share some features or goals in common with WAFL.

• Cedar [giff88]: In Cedar, files are immutable. Any change made to a file results in a new file with a different version number. Thus, a user can always retrieve previously deleted files.

• Echo [swar93] is another filesystem that builds on the Cedar filesystem. It uses logging techniques, but the difference here is that Echo logs all modifications: changes to meta-data as well as user data.

A few other filesystems that make use of logging techniques are the LFS [selt93], JFS [Stev00] and VxFS [Mart99].

• XFS [swee96] is a 64-bit filesystem built from scratch. It contains a volume manager just like WAFL. It has the ability to handle large files. XFS uses a space manager to allocate disk space for the file system. XFS performs write-allocation whereby inodes are positioned close to the files or directories that they reference.

1.5.2 Specialized filesystems

The idea of specialized filesystems is not new. It probably existed from the time appliances were invented.

Alpine

Alpine [brow85] is a filesystem that has been designed specifically to operate as a service on a computer network. Its primary purpose is to store files that represent databases. Alpine performs
all of its communication via a general-purpose remote-procedure call facility. In addition, it also uses logging techniques to ensure atomicity. It runs on the Alpine server. The server was built exclusively to support database research. While Alpine can run on a workstation in addition to a file server, it can never be the only filesystem on a computer; it needs the support of a standard Cedar filesystem.

1.5.3 Swarm

Swarm [hart99] is a network storage system that uses log-based striping to achieve high performance and scalability. Clients collect application writes into an append-only log and the log is striped across multiple storage servers to obtain performance that is a function of the number of storage servers. The Swarm Storage Server is the component of Swarm that serves file data.

Sting on Swarm

Swarm is an infrastructure that provides a log-structured interface to a collection of storage devices. Thus, Swarm provides a scalable, reliable and cost-effective data storage.

Filesystems are tightly coupled with the low-level storage devices making it difficult to extend or configure the filesystem. Thus, in order to extend the functionality of standard UNIX filesystems, it is necessary to modify the filesystem directly to support the desired features.

Sting [hart99] is a log-structured filesystem for Linux that is based on Swarm. Sting interacts with applications through the VFS layer and it relies on the striped log abstraction of Swarm to access storage rather than accessing a block device. Sting makes use of a special file called the inodemap to keep track of all inodes. The rest of the metadata are stored in fixed blocks. Sting uses records to recover its state after a crash. Thus, the idea of Sting on Swarm is similar to WAFL communicating with the RAID manager in ONTAP. The RAID manager stripes writes across the subdisks in a WAFL volume.
1.5.4 Porting the SGI XFS filesystem to Linux

XFS [swee96] is the advanced journaled filesystem that runs on IRIX. SGI ported the XFS to Linux to address the constraints of traditional Linux filesystems. XFS was originally designed to address the issues of small filesystem sizes, small file sizes, statically allocated meta-data and slow recovery times using *fsck*.

This involved the introduction of 2 layers:

1. *linvfs*: to map linux VFS to the IRIX VFS. On Linux, the filesystem interface occurs at the level of a file and the inode. The *linvfs* layer maps all the operations at these two levels into vnode operations that XFS expects.

2. *pagebuf*: to retain most of the advantages of the cache layer in IRIX. This layered buffer cache module sits on top of the Linux page cache.

The motivation for this thesis is somewhat similar to the ideas behind porting the XFS filesystem. But, since the systems involved are different, the issues dealt with are somewhat different.

1.5.5 MS-DOS on BSD

DosFs is a new filesystem for UNIX that uses MSDOS data-structures for permanent storage. It can be used just like a traditional UNIX filesystem. DosFs provides better disk utilization as compared to FFS. It also shows performance improvement over FFS. Changes were also made to the MS-DOS structures to accommodate a multi-user system. The motivation for this project was to extend a single-user filesystem into a multi-user, safe networked file system.
Chapter 2

Background

To understand the decisions made during the course of this project, it is necessary to understand the environment in which WAFL executes on the Network Appliance filer, i.e., Data ONTAP, and the new environment, i.e., FreeBSD. This chapter is an overview of both these environments.

2.1 FreeBSD: an operating system for the PC

2.1.1 Introduction

FreeBSD is one of the several monolithic operating systems derived from BSD 4.4. The first version of FreeBSD was the result of a patchkit developed for Bill Jolitz’s 386BSD. Thus, version 1.0 was released in 1993. The version used for this project is the 4.0 release. The largest change since the first version was the revamping of the virtual memory system with the merged VM/buffer cache. The VM system and the robust network services provided by the kernel were the major reasons for the increase in performance.

This fully featured, stable, and easy to install OS runs on standard PC hardware and work on porting it to Alpha and UltraSPARC processors is underway.
2.1.2 VFS layer

Most of the operating systems have their own native filesystem format. But there is always a need to access another filesystem in order to ensure interoperability and flexible data transfer. Hence, the approach taken is to have a filesystem independent layer that mediates access to all the filesystems. Thus, originated the virtual filesystem (VFS).

The virtual filesystem layer is an object-oriented layer that provides a uniform interface to all the filesystems residing beneath it in the BSD kernel. The current releases of FreeBSD unlike the original releases can support multiple filesystems in the same kernel. This is possible because of the existence of the VFS layer. To implement a filesystem in the kernel, one inherits from the abstract class that abstracts a file system.

The VFS establishes a set of functions that each filesystem must implement. This isolates the implementation of the filesystem from the rest of the OS. It is the responsibility of each individual filesystem to map these generic functions to the details of performing the operation in a particular
A vnode is a generic abstraction of a file or directory and corresponds to an inode in the real filesystem. The primary role of the VFS layer is to establish the relationship between a vnode and a filesystem. The routines that perform this task in the VFS layer are the *namei* and the *lookup* routines. However, the routine *namei* is not filesystem specific. It uses the vnodes and invokes the methods or the pure virtual functions that are attached to the vnode. The vnode structure keeps a pointer that refers to filesystem specific information about the vnode. This pointer connects the abstract notion of a file or directory with the filesystem specific information. Hence, a vnode defines a unique file and one can access the filesystem inode given the vnode.

Listed below are some of the VFS layer support routines:

- *namei and lookup*: These routines follow each pathname until the terminal point (file, directory or character device) is reached. The result is a pointer to the locked vnode corresponding to the terminal point.

- *getnewvnode*: This routine grabs a vnode from the free list thus returning a free vnode that can be used for some other purpose.

Each vnode maintains a reference count that keeps track of how many parts of the system
are using the vnode. Thus, if a vnode is no longer being used, it can be recycled and used for a different file. The following routines manipulate the reference count.

- **vrele**: This routine decrements the reference count and if the reference count of the vnode reaches zero, it returns the vnode to the free list. Any routine that has finished dealing with the vnode should perform a `vrele` on it.

- **vput**: This routine is similar to `vrele` but in addition to decrementing the reference count, it unlocks the vnode.

- **vref**: This routine increments the reference count for a vnode. Any routine that is using the vnode should perform a `vref` on it.

### 2.1.3 BSD buffer cache

Performing I/O to or from the disk is an expensive operation since access to the disk is always slow. Hence, having the filesystem go to the disk, for every I/O requested by the user, will degrade the system response and throughput. Thus, a cache is incorporated into the design to alleviate the cost of accessing a slow device. A cache provides faster access to data residing on disk. It keeps copies of data on disk in an area that is fast to access. The cache cannot hold all of the data on the disk. Thus, the larger the buffer space, the more effective the cache is. Hence, it is considered a good design to have the filesystem sit on top of the buffer cache. Any filesystem cannot interact directly with the disk, it has to go through the buffer cache.

Each buffer header contains information such as the logical address, physical address, buffer size, flags, etc. Search for a buffer is accomplished using the combination of the vnode pointer and logical block number. This combination yields a unique hash value that is then used to hash onto the buffer hash queue. Each buffer always exists on a hash queue.

The algorithm for buffer allocation must be safe. Processes must not sleep forever waiting for a buffer to get released. The user has no control over the allocation of buffers.
Operation

The cache uses system memory to hold copies of disk data. If a program requests a disk block that is residing in the cache, the block is read or written directly from the cache without accessing the disk. On a read, if a requested block is not present in the cache, it is read from the disk into the cache. On a write, if a requested block is not present in the cache, the cache makes room for it, writes to it and marks it dirty. Dirty blocks in the cache are written to the disk later.

Management

Cache management is a matter of deciding what stays in the cache and what needs to be flushed out. Good management is very crucial for good performance of the cache. If useful data is dropped too quickly or if old data is not dropped when appropriate, the cache will not be effective.

The effectiveness of a disk cache is a measure of how often data that is requested is found in the cache. The locality of the blocks referenced also determine the effectiveness of the cache.

The important algorithms used for buffer allocation are:
- **getblk**: The `getblk` kernel service first checks whether the specified buffer is in the buffer cache. If the buffer resides there, but is in use (i.e., present on the LOCKED queue), the sleep service is called to wait until the buffer is no longer in use. Upon waking, the `getblk` service tries again to access the buffer. If the buffer is in the cache and not in use (i.e., present on any of the RECYCLE queues), it is removed from the free list and marked as busy. The corresponding buffer header is then returned. If the buffer is not in the buffer cache, another buffer is taken from the free list and returned. The `getblk` service returns a buffer that contains the data in the block, only if it is already in memory. Otherwise it allocates a new buffer and reads the data from the disk. On return, the buffer is marked busy.

```
<table>
<thead>
<tr>
<th>Hash queue</th>
<th>vp, blkno</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>free list</td>
<td>LOCKED</td>
</tr>
<tr>
<td></td>
<td>LRU</td>
</tr>
<tr>
<td></td>
<td>AGE</td>
</tr>
<tr>
<td></td>
<td>EMPTY</td>
</tr>
</tbody>
</table>
```

Figure 2.4: Buffer queues

- **bread**: The `bread` kernel service assigns a buffer to the given block. If the specified block is already in the buffer cache, then the block buffer header is returned. Otherwise, a free buffer is assigned to the specified block and the data is read into the buffer. The `bread` service waits for I/O to complete, to return the buffer header. The `bread` service is guaranteed to return a buffer actually containing the current data for the block. On return, the buffer is marked busy.
• *brelse:* This kernel service frees the specified buffer. The *brelse* kernel service awakens any processes waiting for this buffer or for another free buffer. The buffer is then put on the list of available buffers. The buffer is marked 'not busy' so that it can either be reclaimed or reallocated.

• *bwrite:* This routine writes the buffer data to disk. This is a synchronous routine, hence it waits for the I/O to complete before returning. It puts the buffer on the appropriate device queue, by calling the device strategy routine.

• *bawrite:* This is an asynchronous version of the bwrite routine, that does not wait for the I/O to complete.

### 2.1.4 Sleep/Wakeup Model

The essence of the sleep/wakeup model in UNIX operating systems can be explained by the following algorithm:

```plaintext
while(condition A == false)
    sleep(event: condition A == true)

condition A=true
wakeup(event: condition A is true)
```

Processes go to sleep because they are waiting for the occurrence of some events like completion of an I/O, waiting for a process to exit, waiting for system resources to become available, etc. The processes enter the sleep state. Once the event occurs, all the processes sleeping on that event wake up and make a transition to the ready-to-run state. When an event occurs, someone (a process or an interrupt handler) will call *wakeup(event)*, which wakes up all processes sleeping on
that event. If no process is sleeping on the event, \textit{wakeup()} has no effect. Sleeping processes do not consume any resources nor is the kernel required to continuously monitor the sleeping processes.

Unix kernel assigns a fixed priority to a process when it goes to sleep. The priority will be the scheduling priority of the process when it wakes up.

### 2.1.5 Resource management in the UNIX kernel

A "resource" is anything that may cause a process to wait. Assume that each resource is represented by a status variable, which may be FREE or BUSY. Whenever a process needs a resource, it checks the resource status to see whether the resource is available. If the resource status is FREE, it sets the status to BUSY and proceeds to use the resource. If the status is BUSY, it calls \textit{sleep(event, priority)} to wait for the resource to become available.

When the resource becomes FREE, a \textit{wakeup(event)} call wakes up all such sleeping processes. Each awakened process must try to get the resource again.
2.1.6 UNIX filesystems

The OS provides a hierarchical directory structure in which a directory can contain other files or directories, thus creating a large tree structure. Each node of the tree can be a directory, file, a link to another node or even a device. UNIX allows access to files based on the file's permission or mode. The file permission field has three components: USER, GROUP and OTHER, and each of them has three parts READ, WRITE and EXECUTE.

Physical filesystems

Storage space on a computer usually resides on several devices. This encompasses several different types of media, including hard drives, CD-ROM drives, and floppy drives. Each of these devices has a distinct physical filesystem associated with it. There are numerous types of physical filesystems found under UNIX, including: ufs, ffs, msdosfs and cd9660.

Logical filesystems

The UNIX kernel provides a standard uniform interface to all the filesystems. All the filesystems are accessed via the same set of system calls. UNIX designates one filesystem as the root filesystem. This filesystem is mounted upon boot at the top of the hierarchy. Other filesystems are attached to the existing hierarchy using the `mount` command. Mount takes a filesystem and maps it to an existing directory in the file tree, called the mount point. Once mounted, it is accessed like any other file or directory within the filesystem.

The internal structure of a filesystem usually consists of the following:

- **Superblock:** contains the basic information regarding the filesystem such as size, number of blocks, list of free blocks and list of allocated blocks, etc.

- **Inode:** contains information about an individual file in the filesystem. The inode also points to disk blocks that contain file data and meta-data.
2.2 ONTAP: an operating system for an appliance

2.2.1 Introduction

The Network Appliance Storage Architecture is driven by a robust, tightly-coupled, multi-tasking, real-time microkernel called Data ONTAP. This compact kernel minimizes complexity and improves reliability. In fact, Data ONTAP software is less than two percent of the total size of general purpose operating systems. By eliminating functions not associated with file service, such as graphical systems or the ability to run local applications, overall system performance increases.

ONTAP provides support for many protocols such as NFS, CIFS, HTTP web service, etc. A unique multiprotocol feature provides simultaneous file service to UNIX, Windows and Web clients without compromising compatibility or performance. CIFS (Common Internet File System)/SMB (Server Message Blocks), another common networking protocol, is tightly integrated into the Data ONTAP microkernel and takes full advantage of the performance, reliability and scalability of NetApp filers.

The Data ONTAP kernel utilizes the robust WAFL (Write Anywhere File Layout) file system. WAFL and RAID were designed together to avoid the performance problems that most
file systems experience with RAID and to ensure the highest level of reliability.

The Data ONTAP software is based on a simple, message-passing kernel that has fewer failure modes than general purpose operating systems, thus demonstrating higher availability.

2.2.2 Buffer Cache

The buffer cache is very tightly integrated with the WAFL filesystem. It consists of 2 parts:

1. The buffer queues that are responsible for recycling buffers.

2. The buffer tree structure that is unique to every WAFL file.

We go into the details of how WAFL manages its buffers in Chapter 3.

2.2.3 Suspend/Restart Model

Any sort of communication in Data ONTAP takes the form of a message. Thus, we have the notion of messages “suspending” and “restarting”. Data ONTAP heavily relies on this model.

What is the suspend/restart model? If a message requires a resource that is not available, it suspends and gets added to a wait-list. After suspending a message, a longjump is performed to return to the main processing loop to receive another message to work on.

When the resource becomes available, the messages are restarted by asynchronously sending them back to the administrator itself. Restarted messages are treated no differently than messages arriving for the first time. The message usually takes the same path again when it is restarted. If the code upto the suspend did not change state, then the suspend/restart behaviour is similar to sleep/wakeup. Since there is no preemption, no message can modify a resource that has been located. This eliminates locking, which is a big advantage in the suspend/restart model unlike the sleep/wakeup model, where locking creates lots of problems.

If blocking is common, then the suspend/restart model is inefficient because the code at the beginning of the operations is re-executed every time a message is suspended. If blocking is a rare event, then this model proves to be efficient as the complexity and overhead of locking are avoided.
2.2.4 WAFL

WAFL is a journaled filesystem that runs on the filer. The filesystem state is always maintained on disks. All operations that modify the filesystem, such as writes and creates, are performed on a transactional basis and are journaled on a non-volatile RAM. Journaling is performed at RAM speeds thus significantly enhancing system performance.

In the event of system failure, WAFL replays the filesystem operations that did not complete before the failure, from the NV-RAM.

WAFL with its RAID data-manager has several features that are typically not found in general-purpose server systems such as:

- File system journaling at RAM speeds
- RAID4 data protection and dynamic disk expansion capability
- File system snapshots that allow users to recover data
- Multiprotocol data locking protection.

Chapter 3 examines WAFL in more detail.
Chapter 3

Discussion: WAFL Vs. FFS

This chapter provides an in-depth explanation about the internal workings of each filesystem. Each filesystem is geared to face a different environment. Hence, there are quite a few differences, but at the same time there exist a few similarities.

WAFL works at the same level as FFS in the FreeBSD operating system. Hence, in this chapter, we analyze the design of each filesystem. From the design, we will present our viewpoints as to where exactly we visualize the improvement. As we examine the features of each filesystem, we also analyse it from the performance point of view.

3.1 BSD FFS

FFS (Fast File System) is the native filesystem of the BSD operating system. The design of the FFS still forms the basic design that is used to build many other filesystems. It offered new levels of performance and reliability that was uncommon in filesystems before.

The FFS consists of a superblock, a block bit-map, an inode bit-map and an array of pre-allocated inodes. The super-block is immutable and is replicated for reliability.

FFS uses file system blocks of 8192 bytes. This helps improve the performance of the system. FFS also manages fragments within a block. The fragments can be as small as 512 bytes but they are normally 1024 bytes.
3.1.1 Write Allocation in FFS

Write allocation is very important for improving performance in FFS. Performance degrades when the seek time of the disk drives is too long. Seek time is the time taken to move the disk heads from one part of the disk to another. Seek time is minimized if file data is organized carefully on disk.

Hence, the concept of cylinder groups was introduced. A cylinder group is a vertical slice of a disk, in other words, a collection of all the blocks residing on the same track but on different heads of the disk. Reading successive blocks in a cylinder group involves switching heads, which is an electronic operation thus being much faster than the mechanical operation of moving heads. This locality offered by cylinder groups helps improve performance. Hence, a large chunk of data can be split so that it resides on different heads of the disk, i.e., within a cylinder group. Nowadays, most of this write allocation is done by the drive controllers of disk drives since they have a more intimate knowledge of the geometry of a disk than the filesystem.

As a part of its write allocation policy, FFS attempts to place file data close to its inode. This helps in minimizing the seek time between reading the file meta-data and the file contents. In addition, FFS also tries to order the filesystem meta-data writes carefully, as this would help the file system consistency check program to recover faster in the event of an unclean shutdown.

3.1.2 Synchronous and Asynchronous writes

FFS forces all the metadata writes to be written synchronously to the disk and does not buffer them in memory unlike the other writes. Synchronous writes guarantee that changes made to the meta-data are reflected correctly on the disk.

3.1.3 Inode structure

An inode contains all the information necessary for a process to access a file. As we mentioned before, each inode defines a unique file. The static form of the inode resides on the disk. In order
for the kernel to manipulate them, it has to read the disk inode into an in-core inode and then perform the operations on the inode.

Inode management is similar to buffer management. The inode lock when set prevents other processes from accessing the inode. When the lock is released, any process sleeping on the inode is awakened. There also exists a reference count which keeps track of the number of active references to the file. An inode also records additional information such as the size of the file, who owns it, its creation time and modification time.

### 3.2 WAFL

WAFL supports NFS using a message-passing interface to the WAFL administrator process. WAFL heavily relies on the assumption that there exist multiple processes running in the kernel. All communication between processes takes place via message-passing.

The WAFL code is structured in three layers:

- Top layer: This layer deals with message-handler functions that get called from the main
processing loop of the administrator process.

- Middle layer: This layer consists of internal functions that use the suspend/restart model while dealing with resource allocation.

- Bottom layer: This layer consists of functions that communicate with the RAID administrator using asynchronous message-passing.

Unlike FFS, WAFL always uses a constant 4K disk block size. This is geared to the standard NFS transfer size.

### 3.2.1 Inode structure

Access to a meta-data file (the "inode file") in WAFL provides access to all the inodes in the system. WAFL inodes are quite similar to the FFS inodes. They contain all the file attributes necessary for a successful NFS `getattr` call. In addition, they contain block pointers to identify blocks containing the file data. A WAFL inode can also hold the data for very small files, i.e., those smaller than 64 bytes.

An in-core inode can hold 16 buffer pointers. These buffers can point to lower-level buffers or simply contain file data. For files smaller than 64 KB, the inode uses the 16 buffer pointers to point to direct data blocks. For files smaller than 64 MB, the pointers in the inode point to indirect blocks which point to actual file data. Inodes for larger files point to doubly indirect blocks. File data only resides in blocks at the lowest level.

WAFL handles indirect blocks differently from FFS. All blocks in a WAFL inode are always at the same level of indirection. This has worked out well since files at a particular indirection level are larger than the FFS files using 4K block sizes.

### 3.2.2 Interaction with the buffer cache

Buffers in WAFL contain information quite similar to that in BSD buffers. The information consists of the inode that the buffer belongs to, the logical block number in the file, the physical block number
Buffers for a file can exist at 3 different levels i.e. level 0 corresponding to the actual user data, level 1 corresponding to the singly indirect block pointers and level 2 corresponding to the doubly indirect block pointers.

(Note: The most recent version of WAFL supports buffers up to level 5.)

Higher level buffers keep pointers to two separate data areas. They are:

- **data**: this points to actual data on the disk
- **data2**: this points to child buffers for indirect blocks. In addition, they are used for storing directory hash information for level 0 buffers.

Thus, buffers are used to build a buffer tree for a file. A buffer tree consists of all the cached blocks for a WAFL file. Level 0 buffers contain user data and are pointed to by the level 1 buffers. Level 1 buffers contain singly indirect blocks and buffer pointers for the data specified in the singly indirect blocks. Level 1 buffers are pointed to by the level 2 buffers, and so on.
Files smaller than 64 bytes are level 0 files, as the data fits in the inode itself. Files larger than 64 bytes but smaller than 64 kilobytes are level 1 files, and files larger than 64 kilobytes are level 2 files.

3.2.3 Write allocation

Even though WAFL resides on top of RAID, the write-allocation strategy is performed at the WAFL level and not at the RAID level.

Most of the filesystems do not perform well when integrated with RAID. Normally, filesystems schedule small writes that are scattered all over the RAID disks. Also, only writes for one file are optimized. Thus, writes for several different files also tend to be scattered across all the RAID disks. This results in the parity disk seeking excessively. WAFL, on the other hand, takes advantage of RAID. It writes blocks to RAID stripes that are near each other, thus reducing traffic to the parity disk. The write allocation code in WAFL deals with the issues of allocating space for blocks on disks and scheduling their writes. WAFL's block allocation code needs to understand how RAID volumes are laid out since WAFL is responsible for assigning block numbers.

The rules used in write allocation are:

- Write as many blocks as possible in each stripe.

- Keep successive writes on the same disk near each other. This means that all the writes would result in the same cylinder, thus reducing seek time for subsequent reads.

- Keep writes at the same offset in each disk. Thus, only a few parity blocks will need to be computed, thus reducing head seek time for the parity disk. If writes are at different offsets, the parity disk will seek excessively while trying to compute various parity blocks.

3.2.4 Meta-data

FFS keeps its meta-data at fixed locations on the disk. On the other hand, WAFL keeps its meta-data in files. This feature ensures that meta-data can be written anywhere on disk like any other
normal user data. WAFL uses the following three files to hold meta data:

- **Inode file**: describes all other files in the file system

- **Block-map file**: corresponds to the free block bitmap in FFS. This file contains a 32 bit entry for each block in the system. A block can be reused only if all the bits are clear. Bit 0 reflects the use of the block by the active filesystem while the rest of the bits reflect the use of the block by the snapshots.

- **Inode-map file**: corresponds to the free inode bitmap in FFS. This file contains an 8-bit entry for each block in the inode file. The entry is a count of the number of allocated inodes per block.
Porting WAFL from ONTAP to BSD was challenging, mainly due to differences in the architectural support provided to the filesystems in each system. The task of getting WAFL to perform reliably meant that the interaction between WAFL and VFS, and WAFL and the BSD buffer cache had to be designed and implemented.

For a filesystem, one needs to have an efficient buffer management strategy. Through the course of this project, we realised that there is no one right way to design and implement the interaction between the WAFL buffer cache and the BSD buffer cache due to the difference in semantics.

This section enumerates the various issues and trade-offs of some of the techniques that we used, and the difficulties and questions that we faced.

4.1 Prototype

The current prototype of WAFL runs in the FreeBSD 4.0 kernel and performs the same functions as any other local filesystem.
4.1.1 Mapping between Vnode routines and WAFL routines

Any system call that needs to access the local filesystem is made to go through the Virtual File System layer. The vnode routines get the information from the user and pack it into a VOP structure. Information can include file name, offset in file, number of bytes to read, etc. The information varies depending on the system call made by the user.

Internal to WAFL, all communication is handled via messages. WAFL routines were originally configured to handle network messages. WAFL makes use of the $sk\_Msg$ structure or the simple kernel Message. The WAFL administrator scans every incoming $sk\_Msg$ and decides which routine it calls. All the required data and the return values are stored in this structure so that at the end of the routine, the same message is returned with the appropriate fields filled in. We have retained this form of internal communication to minimize any unnecessary changes. We allocate a pool of messages at boot-up time. The pool can be accessed by the following interface routines $wafl\_get\_msg()$ and $wafl\_return\_msg()$.

![Diagram](image)

**Figure 4.1: Interaction with the VFS layer**

Thus, at the beginning of any filesystem call, we get a free message, fill it up with incoming data, call the necessary routines and fill up the same message with outgoing data. We transfer the results from the message into the VFS fields and return the message to the pool.
4.1.2 Details on routines implemented

The routines implemented can be split into two categories:

1. Filesystem routines

2. Vnode/Inode routines

The filesystem routines implemented are:

- mount: This routine gets a free volume and either initializes the data-structures with new data or reads the data from the disk. Thus, the routine is common to mounting a new filesystem or a filesystem that already exists. In the latter case, it reads the “fsinfo” block from the first block on the disk.

- unmount: This routine performs the task of doing a final sync-ing of all the dirty blocks before destroying all the data-structures necessary to take the volume off-line.

- root: This routine is required by BSD to identify the root of the filesystem.

- init: This routine provides boot-up code specific to WAFL. Some of the operations performed at boot-up time are:

  1. Setting up a pool of messages
  2. Setting up the buffer table
  3. Setting up the inode table
  4. Setting up volumes and disks.

The vnode routines implemented are:

- Get Attribute

- Set Attribute

- Lookup
• Create
• Open
• Close
• Read Directory
• Read
• Write
• Remove
• Rename
• Make directory
• Remove directory
• Setting up a symbolic link
• Reading a link

Note: Open and Close are the only routines which are not supported by WAFL. All requests to WAFL come in via NFS messages. Hence, WAFL implements only those calls that are supported by NFS.

4.1.3 WAFL-BSD buffer cache

In order to place WAFL on top of the BSD buffer cache, it was necessary to link WAFL buffers to the BSD buffers. This section talks about the various alternatives we had while integrating the WAFL filesystem in BSD. After considering all the options, we describe our final design in detail.

WAFL is unified with its own buffer cache management and housekeeping. FreeBSD as an OS also supports a buffer cache for all the filesystems. Since WAFL resides at the same level as
FFS, we wanted WAFL to interact with the various components of the FreeBSD OS in the same manner that FFS does.

FFS interacts with the raw disk through the BSD buffer cache only. Bypassing the buffer cache and reading from or writing to the disk directly is not the way standard BSD filesystems perform. Hence, we decided to keep the BSD buffer cache. All the housekeeping of the buffers is already done by the BSD code. Thus, we decided to take advantage of this and reuse the functionality of BSD buffer management rather than having the WAFL buffer cache code interact directly with the raw disk.

Thus, we had the following options:

1. WAFL buffer cache interacts with raw disk.

   This option was rejected because on the appliance, WAFL interacts with the RAID layer. So WAFL does not have an intimate knowledge of the disk. WAFL does write allocation and allocates volume block numbers for each buffer to be written. The volume block number is then translated by the RAID layer appropriately. Changing WAFL to have knowledge of the way the disk works would have meant a lot of unnecessary changes. Besides, we would not be taking advantage of the BSD buffer cache code.

2. Skip the WAFL buffer cache totally and have all the WAFL reads/writes access the BSD buffer cache directly.
The WAFL buffer cache is very tightly integrated with the filesystem. Changing it would have resulted in changing the entire structure of WAFL. Hence, this option was not feasible.

3. WAFL buffer cache interacts with BSD buffer cache which in turn interacts with the raw disk.

Option 3 was the optimal choice as we made use of both the WAFL and the BSD buffer cache code. We shall now go into a detailed explanation of the modified design.

![Diagram](image.png)

Figure 4.3: Final layout

4.1.4 Mapping between WAFL buffers and BSD buffers

A higher level WAFL buffer keeps pointers to two data areas data and data2. "data" actually points to data from disk while data2 has various functions. For indirect blocks, data2 contains the buffer pointers (data2.bufp) for the volume block numbers in data (data.vol_bno). For level 0 directory buffers, data2 contains the directory hash information. Level 0 buffers just contain the data pointer which points to user data.

We modified the structure of a WAFL buffer slightly, to introduce the BSD buffer. Every WAFL buffer has a corresponding BSD buffer which contains the data that is to be reflected on
the disk. Thus, as the diagram depicts, each WAFL buffer points to a BSD buffer, and its data pointer points to the same data as the BSD buffer’s data pointer.

To help keep track of its parent WAFL buffer, each BSD buffer has a pointer back to the WAFL buffer it is attached to.

4.1.5 Basic Reads and Writes

Whenever WAFL wishes to read or write user data, it uses messages to communicate with the RAID subsystem. We translated a “read” message to a BSD “bread” and a “write” message to a BSD “bawrite”. As mentioned before, bawrite is an asynchronous write unlike the synchronous bwrite. We prefer the asynchronous version as once the write is queued for completion, it does not affect us and we can go ahead and perform other tasks. At the time of allocation of a buffer, we use the BSD routine “getblk”, which uses the vnode pointer and logical block number, to hash onto a free and unused buffer.

Thus we have the following major routines:

\[
\text{waflallocbuf} \rightarrow \{\ldots \text{getblk}() \ldots\}
\]
waflreadbuf -> {......bread()......}
waflwritestripe -> { . .bawrite()....}

4.1.6 Semantic Difference

The BSD buffer cache works in such a way that after every getblk or bread, the buffer is locked and removed from any of the free queues before being returned. Such buffers cannot be locked again. This translates to the fact that once you have a buffer in your possession, you effectively cannot getblk or bread it again.

One would probably argue that bread should work on a locked buffer as you just wish to fill it with valid data. But, bread contains a call to getblk which in turn locks the buffers.

In contrast, after a bwrite or a bawrite, the buffer is unlocked and released via brelse onto one of the free queues. Now, the buffer can be reused by any other process for any other purpose.

With all this information, we notice a subtle difference in the semantics of the BSD buffer cache and the WAFL buffer cache. In WAFL, once a buffer is allocated it is not released. They are released only in exceptional cases such as file deletion, etc. When WAFL buffers are released, the data associated with them is invalidated. Hence, even if a buffer needs to be read or written, it is not released.

This mismatch gave rise to lots of problems and design decisions. The main problem occurs when WAFL schedules multiple consistency points. Each consistency point contains a batch of writes. Hence, at the end of a consistency point, all the BSD buffers that were being used get released (by brelse). Technically, these buffers do not belong to any process even though the pointers remain valid. WAFL yet considers these buffers to be good while BSD has already disposed them after the completion of the write.

Thus, it was necessary to adopt a lock-all or unlock-all policy irrespective of whether a read/write has occurred.
4.1.7 Initial Design

We came up with two policies:

- Policy 1: Unlock all buffers and when they get reused make sure WAFL gets notified about it.

According to this policy, the corresponding BSD buffers for each WAFL buffer should remain unlocked. We have two scenarios in this case:

1. Read and Release: we have

   wafl_read_buf -> { .. bread(..) and brelse(..) }

2. Write and Release: The BSD bawrite releases the buffer automatically after the write is complete.

We release the buffers after reading or writing to them. Since they are released, they will reside on one of the free queues. Inspite of being on the free queues, the buffer pointers remain valid until they are reused by another process. At this point, when we are sure that the buffer is being reused for some other purpose, we schedule a callback that removes the corresponding WAFL buffer from the tree.

Once the WAFL buffer is removed from its tree, it will be reloaded again by the WAFL code when necessary. If the WAFL buffer is yet in the tree and its corresponding BSD buffer has been reused then the data pointers are incorrect.

Deleting the WAFL buffer from its buffer tree, once we get the callback from the BSD buffer cache code, can be troublesome because certain parts of the code assume that the WAFL buffer has been loaded, and is valid. If a callback occurs and we delete a valid WAFL buffer, the call will fail if the buffer was assumed to be loaded by WAFL. It is thus necessary to know when it is safe to release the buffer and when it would be unsafe to do so. After a careful scan of the code, we realised that this was difficult because not only is it difficult to keep track of
when a buffer is in use, but it is hard to make sure its child in the buffer tree is also not in use. We could not isolate sections in the code where deleting a buffer might cause the system to panic. Hence this design was discarded.

At this point, we realise that it would be easier and more reliable if we kept the buffers locked and released them only when necessary. Following this approach, WAFL controls the release of its buffers, not the BSD buffer cache.

- Policy 2: Keep all BSD buffers that are needed by WAFL locked

This led to our design 2 which followed the policy of keeping all valid buffers locked.

Let us consider the two scenarios again

1. Read and keep locked:
   For reading buffers we use bread. Since the BSD bread routine returns a locked buffer, we keep it that way and do not release it as we did in the previous scenario.

2. Write and keep locked:
   For writing data to buffers we use bawrite. As we discussed before, this routine releases the buffer onto the free queues. We changed the routine so that for all the WAFL buffers we do not release the corresponding BSD buffers even if the write is completed.

   The routine biodone() is called when an asynchronous write completes. To implement the “Keep it locked” policy, the biodone routine has been changed as follows:

   biodone()
   {
       if(asynchronous write)
       {
           if( BSD buffer belongs to a WAFL buffer and data is valid)
               return without releasing;
           else
               ...
   }
Thus, with this policy we have locked BSD buffers hanging around for a long time unless we decide to release them. So the question arises “When do we release the buffers?” Can we rely on WAFL to decide when would be the right time to lock/unlock the buffers? It turns out that WAFL does not support the concept of locking/unlocking. The only time when WAFL releases its buffers is when it wishes to make them invalid; for example, when a file is deleted.

As a solution, we could keep track of all the locked buffers for every system call. We can be sure that the number of buffers used per system call would always be a small number since even if we read a big file, the read request is broken down into smaller requests by the VFS layer before handing it to WAFL for further processing. Thus, we need a data structure to keep track of all locked buffers for every system call so that we can unlock them at the end of the system call. If two system calls are independent of each other and they access a totally different set of blocks, it is fine. But, if two system calls touch the same block, then there might be the problem of releasing the block while the other process is using it. There is also the question of what to do with dirty buffers, buffers that get locked repetitively, and buffers that belong to meta-files.

We realised that this design too had a few flaws. If we unlock buffers after every system call, we are going to be unlocking and releasing WAFL buffers and reloading them at the beginning of every system call. This will harm the performance. We should keep a few buffers around in the cache to avoid reloading every buffer. This especially applies to buffers at a higher level in the tree. Even buffers belonging to metafiles such as the inode file need to be present across system calls. If we release the buffers of the inode file and reload them at the beginning of every system call, our performance will take a big blow especially if the inode file is huge.

On the other hand, if we keep buffers locked and do not release them, eventually BSD will run out of buffer space.
4.1.8 Final Design

In our final design, we reuse some of the buffer housekeeping from WAFL. Our policy is to keep buffers that are in WAFL's recycle queue in an unlocked state. The rest of the buffers in the buffer tree remain locked.

WAFL maintains two queues: the empty queue and the recycle queue. In the empty queue, we queue up buffers with no associated data, i.e., the corresponding BSD buffer has been invalidated. In the recycle queue, we queue up buffers with associated data. Thus, buffers in the recycle state have valid data pointers but they can be safely invalidated. Thus, only buffers in the recycle queue are eligible for invalidation, as a result of which they get queued onto the empty queue. So, when a "recycled" buffer is invalidated it is removed from the buffer tree it belongs to. In future, if the buffer is needed, it is reloaded again.

When a WAFL buffer is queued onto the recycle queue, the corresponding BSD buffer is released so that it resides on one of the free queues in the BSD buffer cache (i.e., LRU, AGE, EMPTY). This buffer can be reused by any other process. If it is reused, the BSD code makes a callback to ensure that its parent WAFL buffer in the recycle queue is invalidated. Hence, we make use of callbacks to ensure that we are notified whenever a BSD buffer is taken out of our possession.

The rest of the WAFL buffers in the buffer tree that are locked cannot be recycled, hence they do not exist on any of the free queues. The corresponding BSD buffers are locked as well.

To summarise, the different states that WAFL-BSD buffers can exist in are as follows:

1. WAFL buffer exists in the empty queue. It does not have data, hence, its BSD buffer does not exist.

2. WAFL buffer is in the buffer tree and the recycle queue as its data is not needed. The corresponding BSD buffer is unlocked and released onto the BSD free queue.

3. WAFL buffer does not exist on the recycle queue and is locked. The BSD buffer also does not exist on any of the BSD free queues and is also locked.
Pointers linking buffers in a buffer tree

Pointers linking buffers onto Recycle/Empty Queue

Buffers that exist in a tree and on the recycle queue

Buffers that are locked and cannot be recycled

Empty Buffers

Figure 4.5: Various states of WAFL buffers
4.1.9 Callbacks

There are three scenarios in which we schedule callbacks from the BSD code into the WAFL code:

1. Reusing a BSD buffer that belongs to a WAFL buffer: In this case, we make sure that the WAFL buffer in question is still valid. Once the check is complete, the buffer is removed from the buffer tree and the recycle queue, and moved into the empty queue.

2. Asynchronous Write completion: When an asynchronous write that originated in WAFL is completed, we make a callback to clear the write flags and check the buffer for its queueing status. This is synonymous to the RAID layer sending a message to WAFL indicating that the stripe has been written.

3. Keeping BSD buffer locked after a write: After an asynchronous write is completed, we check to see if the write originated from WAFL. If so, we do not release the BSD buffer onto the free queue and keep it locked.

4.1.10 Allocating and Scavenging buffers

At boot-up time, we allocate a fixed number of buffers. Initially, all buffers exist in the empty queue. When we allocate a WAFL buffer, we remove a WAFL buffer from the empty queue and attach a BSD buffer to it. This is a valid WAFL buffer. As buffers get used, recycled and invalidated they return to the empty queue without any attached BSD buffers.

If there is lots of activity happening in the filesystem, we might run out of buffers in the empty queue. Hence, at the time of allocation, we attempt scavenging to refill the empty queue. The task of scavenging consists of examining the recycle queue, removing the least recently used buffer from the queue, invalidating it and adding it to the empty queue.

We establish a low water mark and high water mark in order to necessitate the process of scavenging. If the empty buffer count falls below the low water mark, we start the process of scavenging.
4.1.11 Consistency Points

Consistency points are a key feature of WAFL. To establish a consistency point, WAFL writes out all dirty blocks and inodes to the disk. Writes occur only during consistency points.

We retained this feature of WAFL in BSD, using a kernel-level process created at boot-up time. This process is dedicated to the task of triggering a consistency point after a certain interval has occurred. Currently, this process, or the WAFL CP daemon as we call it, schedules a consistency point every 10 seconds.

In the prototype, consistency points are scheduled:

1. When the CP daemon triggers it.
2. When a volume is taken offline at the time of unmounting.
3. For snapshot operations.
4. When there are too many dirty buffers in the system.

Multiple conditions can exist at the same time. Hence, we set a flag to ensure that, once a consistency point is taking place, no one else triggers another one until the consistency flag is reset.

4.1.12 Snapshot Creation and Deletion

A snapshot is a special form of a consistency point. Snapshots are read-only copies of the filesystem. When a snapshot is created, the root inode is copied to the snapshot inode. Every block in the filesystem is updated so that it is a part of the active file system and the snapshot being created. Thus, at the time of creation, a snapshot points to the same blocks as the active file system. But, as new blocks get allocated, dirtied and written out, the snapshot begins to take up space. Old blocks are not overwritten if they belong to a snapshot. Data for the active file system is written out to new blocks. Snapshots can be spread over multiple consistency points for a big file system.

To enable snapshots in our prototype, we implemented two system calls.

1. snapcreate: Creation of a snapshot in a particular volume
2. snapdelete: Deletion of a snapshot in a particular volume

A maximum of 21 snapshots can be created at any time. Thus, once WAFL is mounted, the user has the ability to create/delete snapshots via the system calls. A snapshot provides us with the latest stable copy of our filesystem. Thus, we can go back in time to retrieve our files with snapshots.

4.1.13 Implementing suspend/restart

As WAFL is based on the suspend/restart model, we had to come up with an effective replacement for this model in BSD. Suspending and starting the operation again from the beginning cannot be accomplished in a BSD kernel as the kernel relies on the sleep/wakeup model.

We use event-based process blocking instead of suspend/restart. Thus, instead of suspending, we make the process sleep on an event. Once the event occurs, the process wakes up and continues execution from the same point. Some of the instances where we have made use of this model are:

1. If a dirty buffer is being written out and it is still locked, we sleep on the buffer. As soon as the write completes, we schedule the corresponding wakeup.

2. If we wish to access an inode that is dirty or is still a part of an in-progress CP, we schedule a consistency point and sleep until the inode is flushed to the inode file.

3. If the number of dirty buffers in the active file system exceeds the high water mark, we sleep. The wakeup is scheduled only after a few buffers are written out and the number reaches a low water mark.

4. During snapshot creation/deletion, when we wish to schedule a CP, if there is already another CP in progress, we sleep until the in-progress CP reaches completion.
4.1.14 Support for Multiple Volumes

At the time of boot-up, we initialize a fixed number of volumes. Each volume is given a unique identifier (for eg. vid1, vid2, etc.). The volume identifier is a part of the filehandle that uniquely identifies each file. Thus, we can allocate inodes separately for each volume. Since we have distinct inodes, the buffers are also distinct. This facilitates mounting of WAFL on multiple volumes and copying of files across volumes. This feature can be extended so that a volume can contain groups of disks. The current implementation associates a volume with a single block device.
This section presents measurements of the performance of WAFL in FreeBSD. We evaluate the performance of our system using three benchmarks: a micro-benchmark, the Andrew benchmark and the Postmark benchmark. WAFL is compared to FFS, the local filesystem on FreeBSD. We also evaluate the performance of WAFL running on Vinum.

The comparisons demonstrate that WAFL performs very well as compared to FFS especially in meta-data operations. We provide a detailed explanation below.

5.1 Vinum

The Vinum Volume Manager is a block device driver which implements virtual disk drives. It isolates disk hardware from the block device interface and maps data in ways which result in an increase in flexibility, performance and reliability. Vinum implements the RAID-0, RAID-1 and RAID-5 models, both individually and in combination. In RAID-5, a group of disks are protected against the failure of any one disk by an additional disk with block checksums of the other disks.

For the purpose of testing, we configured vinum to emulate a RAID-5 organization on 3 subdisks. Each subdisk provides two gigabytes of storage. The equivalent of one subdisk would be used to store the parity.

We tested the performance of WAFL on this configuration so as to simulate WAFL running
on RAID4 on the network box. Since vinum does not support RAID-4, we opted for the RAID-5 configuration with a rotating parity block.

5.2 Experimental Setup

The experiments were run on a single workstation. This workstation has a 166MHz Pentium-II processor, 64 Mb of memory and runs FreeBSD 4.0.

5.2.1 Andrew Benchmark

The Andrew benchmark measures normal filesystem behavior that we see daily. This test ensures that the filesystem can function practically.

The Andrew benchmark emulates a software development workload. It has four phases: (1) creates subdirectories recursively; (2) copies a source tree; (3) examines the status of all the files in the tree without examining their data and (4) examines every byte of data in all the files.

The script operates on a collection of files constituting an application program. The operations represent typical actions of an average computer user. The input to the benchmark is a source tree of about 70 files. The files total about 200 KB in size.

We report the mean of 10 runs of the benchmark.

<table>
<thead>
<tr>
<th>Test</th>
<th>WAFL BSD</th>
<th>FFS BSD</th>
<th>WAFL Vinum BSD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test1: Creating directories (secs)</td>
<td>0.37</td>
<td>1.21</td>
<td>0.33</td>
</tr>
<tr>
<td>Test2: Copying files (secs)</td>
<td>2.17</td>
<td>3.02</td>
<td>1.29</td>
</tr>
<tr>
<td>Test3: Recursive Directory statistics (secs)</td>
<td>1.55</td>
<td>1.30</td>
<td>1.60</td>
</tr>
<tr>
<td>Test4: Scanning each file (secs)</td>
<td>2.95</td>
<td>3.35</td>
<td>3.03</td>
</tr>
</tbody>
</table>

Table 5.1: Andrew Benchmark Results

Test1 results confirm the importance of asynchronous meta-data updates in WAFL. FFS, on the other hand, ensures filesystem consistency by flushing meta-data synchronously. We do not notice any significant advantage of running WAFL over a RAID-5 configuration of Vinum except
in Test2 where we are copying a bunch of files sequentially. This indicates that vinum is striping
the file data across disks. Thus, all the disk drives share the write load thereby increasing the
performance.

We conclude that WAFL performs reasonably well when compared to FFS. In addition,
these tests highlight the importance of using a log-structured journaling filesystem, *softupdates* or
some other method of safely delaying, grouping, and avoiding writing to disk synchronously for
meta-data operations, in order to improve performance.

5.2.2 Micro-benchmark

The micro-benchmark is an attempt to study WAFL’s performance using a different perspective.
The micro-benchmark measures the time taken to perform basic operations on files with varying
sizes. This test was essential considering that WAFL operates on blocks with a fixed size of 4096
bytes and FFS operates on blocks of 8192 bytes but also uses fragments (512 - 1024 bytes). For
the purpose of our tests, we opted for files with sizes 2KB, 4KB, 6KB, 8KB, 10KB, 12KB, 14KB
and 16KB.

<table>
<thead>
<tr>
<th></th>
<th>WAFLBSD</th>
<th>FFSBSD</th>
<th>WAFLVinumBSD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recursive deletion of</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a large directory</td>
<td>0.08</td>
<td>2.33</td>
<td>0.08</td>
</tr>
<tr>
<td>(10 directory entries,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>70 file entries)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.2: Measurement of a recursive delete.

Table 5.2 measures a recursive delete on a substantially big directory. We notice the impact
of asynchronous writes on meta-data operations such as create, delete, etc.

<table>
<thead>
<tr>
<th></th>
<th>WAFLBSD</th>
<th>FFSBSD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creation (secs)</td>
<td>5.85</td>
<td>34.33</td>
</tr>
<tr>
<td>Overwriting with 4K</td>
<td>18.49</td>
<td>25.51</td>
</tr>
<tr>
<td>data (secs)</td>
<td>4.98</td>
<td>31.31</td>
</tr>
</tbody>
</table>

Table 5.3: Basic operations on 1000 “size exactly 4KB” files
Table 5.3 measures create, overwrite and delete on 4KB files. FFS seems to be slower. We presume that FFS is using fragments to store 4096 bytes instead of using one 8192 KB block, thus making it slower.

<table>
<thead>
<tr>
<th></th>
<th>WAFL_{BSD}</th>
<th>FFS_{BSD}</th>
<th>WAFL_{Vinum}_{BSD}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copying a large file (6MB) (secs)</td>
<td>6.19</td>
<td>5.82</td>
<td>7.97</td>
</tr>
<tr>
<td>Deleting a large file (6MB) (secs)</td>
<td>0.05</td>
<td>0.08</td>
<td>0.04</td>
</tr>
</tbody>
</table>

Table 5.4: Operations on large files

Table 5.4 measures create and delete on a 6MB file. WAFL and FFS perform approximately the same. WAFL may fall short of valid buffers when copying a large file. Scarcity of buffers will trigger a consistency point thus delaying the completion of the operation in WAFL. However, one would expect WAFL to perform better over Vinum due to the sequential writes. But, one cannot predict how Vinum behaves or how it performs the write allocation, since it is an invisible layer.

Figure 5.1 measures the read performance for files of various sizes. We established the median of a 1000 reads for files of one size. We repeated the experiment on other file sizes. We attribute the erratic performance of FFS reads to the usage of fragments. We notice that the performance for WAFL files that fit within the same number of blocks is approximately the same. For example, a 2 KB and a 4KB file would employ the same number of blocks (i.e., one), since WAFL has a fixed block size.

In figure 5.2, we measure the creation of 1000 files for various file sizes.

5.2.3 PostMark

PostMark is a reasonable simulation of an Internet mail or USENET news system. It creates lots of relatively small files (between 0.5KB and 10KB in size), performs operations (or transactions) on them, and then deletes them in rapid succession. The transactions performed include read, append, create and delete.

For our tests, we configured PostMark with the following values:
1. Transactions: 500

2. Files range between 500 bytes and 9.77 kilobytes in size

3. Random number generator seed is 42

4. The base number of files is 500

5. 0 subdirectories will be used

6. Block sizes are: read=512 bytes, write=512 bytes

7. Biases are: read/append=5, create/delete=5
8. Using Unix buffered file I/O

We reconfigured Postmark to handle 1000 files and 1000 transactions as shown in Table 5.6.

We observe a significant difference in performance between WAFL and FFS. This could be due to the fact that WAFL groups all the writes and schedules them together for the next consistency point. The time WAFL takes to complete the entire operation is so small that it probably does not involve a consistency point. Thus, for WAFL we are probably observing the performance of a bunch of operations which do not access the disk at all. On the other hand, FFS is accessing the disk more frequently due to its use of synchronous meta-data updates.
<table>
<thead>
<tr>
<th></th>
<th>$WAFL_{BSD}$</th>
<th>$FFS_{BSD}$</th>
<th>$WAFL_{VinumBSD}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Time (secs)</td>
<td>1</td>
<td>45</td>
<td>1</td>
</tr>
<tr>
<td>No. of transactions per second</td>
<td>500</td>
<td>31</td>
<td>500</td>
</tr>
<tr>
<td>Data Read (Kbytes/sec)</td>
<td>1310</td>
<td>29.03</td>
<td>1310</td>
</tr>
<tr>
<td>Data Written (Kbytes/sec)</td>
<td>4200</td>
<td>93.38</td>
<td>4200</td>
</tr>
</tbody>
</table>

Table 5.5: Postmark Results: 500 files, 500 transactions

<table>
<thead>
<tr>
<th></th>
<th>$WAFL_{BSD}$</th>
<th>$FFS_{BSD}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Time (secs)</td>
<td>11</td>
<td>112</td>
</tr>
<tr>
<td>No. of transactions per second</td>
<td>500</td>
<td>25</td>
</tr>
<tr>
<td>Data Read (Kbytes/sec)</td>
<td>227.97</td>
<td>22.39</td>
</tr>
<tr>
<td>Data Written (Kbytes/sec)</td>
<td>768.93</td>
<td>75.52</td>
</tr>
</tbody>
</table>

Table 5.6: Postmark Results: 1000 files, 1000 transactions

### 5.3 Summary

Our performance measurements show that WAFL performs better than FFS due to WAFL's asynchronous writes. Consistency points scheduled by WAFL enable us to perform asynchronous writes in the FreeBSD kernel, even for metadata operations.

If the entire operation starts and ends within the 10 second interval between two successive consistency points, the user does not bear the overhead of accessing the disk. WAFL also seems to perform favourably to the workload of a news system. Thus, WAFL seems to be a feasible alternative for the FreeBSD kernel.
Chapter 6

Conclusion and Future Work

6.1 Conclusion

As a result of this thesis, we have an implementation of WAFL in the current release of FreeBSD version 4.0. WAFL interacts with the other components of the kernel in the same way that FFS does. Most importantly, the project involved implementing two layers: the top layer that interacts with the VFS layer in BSD and the bottom layer that interacts with the BSD buffer cache.

A file system designer needs to make many choices when implementing a file system. Not all features may be appropriate or even necessary for the system. In addition, system constraints dictate some choices while available time and resources may dictate others.

Porting filesystems is a lot more work than porting user applications as one has to deal with the internals of the kernel directly. The migration of WAFL to BSD has been challenging due to the semantic differences in the two designs. WAFL has been customized for the ONTAP kernel, an appliance OS, to provide fast NFS access times. The architectural and semantic differences between the ONTAP and BSD kernels have strongly influenced the final design.

The buffer cache in any system is inherently very complex. Our system involved layering the WAFL buffer cache over the BSD buffer cache. It was an attempt to find a clean interaction between two conceptually different buffer caches.
We have supported the implementation of WAFL in BSD with some performance numbers. The performance measurements demonstrate that WAFL performs reasonably well when compared to FFS. WAFL’s main advantage lies in meta-data operations where it shows a significant performance gain due to asynchronous updates. A notable application area where improved performance is noticed is news processing. News involves many tiny files, which tends to be challenging to handle.

This thesis has presented our arguments that the implementation of a reliable filesystem like WAFL in the UNIX kernel would be desirable and our performance section proves that such an implementation is feasible.

6.2 Future Work

On the filer, WAFL takes advantage of the RAID-4 layer beneath it and optimizes its write allocation. The write-anywhere design allows WAFL to operate efficiently with RAID. Thus, WAFL takes advantage of the RAID-4 layout and minimizes the write performance penalty incurred by RAID. We have tested WAFL over a RAID-5 configuration of Vinum. Since WAFL interacts with RAID-4 in ONTAP, it would be worthwhile to add RAID-4 support to Vinum and run WAFL over it. We did not notice a significant difference in performance by running WAFL over Vinum (RAID-5 configuration). We expect this to change with the RAID-4 configuration of Vinum.

In ONTAP, WAFL logs NFS requests that occur between consistency points. In the event of a system crash, the system boots up the most recent consistency point and then replays all the requests stored in the log. This feature is not supported in our version. It would be beneficial to implement some form of NVRAM logging in FreeBSD to avoid losing information if there is an unclean shutdown. Implementing this feature in software would require a new design. We could add NVRAM hardware to the system but NVRAM is not affordable for a normal PC user.

Further reconstruction of WAFL in BSD (especially the buffer cache interaction) is necessary to make it more robust so that it can withstand a high and continuous load of filesystem activity. It
is difficult to assess and justify the fragility of the current implementation of WAFL. We attribute the fragility to the complexity of the buffer cache mechanism in WAFL and BSD, the difference in semantics between the two systems and the asynchronous behaviour of WAFL writes. Asynchronous behaviour of a system is not suitable for testing and debugging. This calls for a re-evaluation of the existing design, and rebuilding the entire system considering all the problematic factors.
Bibliography


[russ97] Mark Russinovich and Bryce Cogswell. "Examining the Windows NT Filesystem — Mark and Bryce open up the inner workings of the NT filesystem by describing how a filesystem request originates in a user's program and ends up as a disk access. They also present an application called Filemon that monitors and displays all filesystem activity". Dr. Dobb's Journal of Software Tools, Vol. 22, No. 2, pp. 42-??, February 1997.


