Storage Virtualization with a Stackable File System

A Thesis Presented
by
Sunil Satnur
to
The Graduate School
in Partial Fulfillment of the
Requirements
for the Degree of

Master of Science
in
Computer Science

Stony Brook University

Technical Report FSL-05-03
December 2005

Stony Brook University
The Graduate School

Sunil Satnur

We, the thesis committee for the above candidate for the
Master in Science degree,
hereby recommend acceptance of this thesis.

Dr. Erez Zadok, Thesis Advisor
Computer Science

Dr. Tzi-cker Chiueh, Chairperson of Defense
Computer Science

Dr. Alexander E. Mohr
Computer Science

This thesis is accepted by the Graduate School

Dean of the Graduate School

Abstract

Different files have different importance, sizes, and access patterns. Standard file systems allow only per-mount-point customizations of the underlying storage. This means that: (1) storage resources are wasted, (2) data reliability and performance are suboptimal, and (3) files are not stored conveniently for the users. Therefore, individual files require different ways of storing them.

We have designed a stackable file system called Redundant Array of Independent File systems (RAIF). It allows users to store individual files with greater flexibility than is available with traditional disk-level RAID systems. RAIF can be mounted on top of any combination of other file systems (called branches) including network, distributed, disk-based, and memory-based file systems. RAIF uses a pool of lower file systems to place individual files on all or a subset of the lower file systems. RAIF combines the data survivability properties and performance benefits of traditional RAIDs with the greatly increased flexibility of composition, improved security potential, and ease of development of stackable file systems. Existing encryption, compression, anti-virus, versioning, tracing, consistency checking, and other stackable file systems can be mounted above and below RAIF, to allow many more configurations of storage. Individual files can be distributed or striped across branches, replicated, stored with parity, or stored with erasure correction coding (ECCs) to recover from failures on multiple branches. RAIF uses dynamic load balancing and dynamic mechanisms based on file storage types to better utilize the storage capacity and bandwidth resources.

In this thesis we describe the current RAIF design, provide preliminary performance results and discuss current status and future directions.

To my mom and dad

Contents

List of Figures

Acknowledgments

I would like to thank my advisor, Dr. Erez Zadok, for his ideas, help, and encouragement. Nikolai Joukov was a great mentor for the entire duration of this project. His experience and guidance were invaluable. Thank you to Dr. Tzi-cker Chiueh and Dr. Alexander Mohr for taking the time to serve on my thesis committee and for their valuable feedback. I would also like to thank Gopalan Sivathanu, Sean ”Viggo” Callanan and Avishay Traeger for their reviews and endless amusements in the lab.

This work was partially made possible by an NSF CAREER award EIA-0133589, NSF award CCR-0310493, and HP/Intel gifts numbers 87128 and 88415.1.

Chapter 1

Introduction

Redundant Array of Independent Filesystems (RAIF) is the first file system which combines the data survivability of RAID-like storage systems at the file system level with the flexibility of being composed from any combination of underlying storage devices. It also features the ability to apply per-file storage policy configurations in an easy and transparent manner.

For close to two decades, grouping hard disks together to form RAIDs has been considered a key technique for improving storage survivability and increasing data access bandwidth [27]. However, most of the existing hardware and software RAID implementations require that the storage devices underneath be of one type. For example, several network stores and a local hard drive cannot be seamlessly used to create a single RAID.

Traditional RAID systems implemented either in hardware or software operate at the data-block level, where high level meta-data information is not available. It is useful to make storage policy decisions based on metadata like file type, user ID and file size, which achieves optimal utilization of the storage resources. For example, in a build environment, it is useful to store C and header files more reliably than intermediate object files. But in RAID systems, the same redundancy and recovery mechanisms are applied for all files irrespective of their importance, resulting in suboptimal storage utilization and efficiency.

There are several implementations of RAID-like file server systems that operate over a network [1, 10], including implementations that combine network and local drives [9]. However, past systems targeted some particular usage scenario and had a fixed architecture. Inflexibilities introduced at design time often result in sub-optimal resource utilization. RAIF leverages the RAID design principles at the file system level, and offers better configurability, flexibility and ease of use in managing data security, survivability, and performance. RAIF is highly portable because it can mount on lower file systems

The concept of applying different storage policies to different files is not new. There are driver-based approaches [9] with hierarchical RAID levels, including compressed RAID, to improve storage utilization and response time. There are hardware-based approaches [37] that build intelligence into the array controller to implement different policies.

However, the existing techniques do not rely on high level meta information to derive their per-file policies. Rather they rely on low level statistical data related to file accesses. This might not always produce the best policies. For example, having a different policy for the active and inactive data sets renders the system inefficient if the active data set keeps changing a lot.

A file system is the best place to implement flexible per-file storage policies, not only because it has access to various high level meta-data, but also because there is enough freedom to implement any kind of storage policy in software.

RAIF is a fan-out RAID-like stackable file system. Stackable file systems are a useful and well-known technique for adding functionality to existing file systems [45]. They allow incremental addition of features and can be dynamically loaded as external kernel modules. Stackable file systems overlay on another *lower* file system, intercept file system events and data bound from user processes to the lower file system, and in turn manipulate the lower file system's operations and data. A different class of file systems that use a one-to-many mapping (a fan-out) has been previously suggested [11, 29] and was recently included in the FiST [38, 45] templates.

RAIF derives its usefulness from three main features: flexibility of configurations, access to high-level information, and easier administration.

First, because RAIF is stackable, it can be mounted over any combination of lower file systems. For example, it can be mounted over several network file systems like NFS and Samba, AFS distributed file systems [13] , and local file systems at the same time; in one such configuration, fast local branches may be used for parity in a RAID4-like configuration. If the network mounts are slow, we could explore techniques such as data recovery from parity even if nothing has failed, because it may be faster to reconstruct the data using parity than to wait for the last data block to arrive. Stackable file systems can be mounted on top of each other. Examples of existing stackable file systems are: an encryption [40], data-integrity verification [17], an antivirus [23], and a compression file system [42]. These file systems can be mounted over RAIF as well as below it, among others.

Second, because RAIF operates at the file system level, it has access to high-level file system meta-data that is not available to traditional RAIDs operating at the block level. This meta-data information can be used to store files of different types using different RAID levels, optimizing data placement and readahead algorithms to take into account varying access patterns for different file types.

Third, administration is easier because of the following reasons.

- Files are stored on unmodified lower-level file systems, and RAIF allows a file to be stored in any subset of the lower-level branches. The advantage of storing files on unmodified file systems are twofold. First of all, lower branches can easily be expanded and shrunk, either offline by copying data to other devices, or in place by using logical volume managers because they give a standard file system interface. Existing backup software can be seamlessly integrated with RAIF because all lower operations are done on files, and not individual blocks of data.

- RAIF can transparently and concurrently use different redundancy algorithms for different files or file types. For example, RAIF can stripe large multimedia files across different branches for performance, and use two parity pages for important financial data files that must be available even in the face of two failures.

- RAIF provides a unified interface for managing multiple different subsets of branches, possibly with overlapping branches, each subset having its own set of policies like RAIF level and striping unit.

In this paper we introduce this new type of stackable RAID-like file system design, our current prototype, and its preliminary evaluation. We describe some general fan-out design principles that are applicable even beyond RAIF. The rest of the paper is organized as follows. project, some interesting implementation details, and outlines future directions. Section 5 discusses related work.

Chapter 2

Design

In this section we present the design of RAIF. The following are the design goals that we considered.

- **Flexibility** It is important to give a wide variety of options to the user, and administer the system in a transparent way. Also, a user must be able to put together a complicated system in a short time. The design of stackable file systems naturally lends itself to more flexibility.
- **Reliability** We use RAID semantics in order to make storage of files more reliable. We use different RAID levels with parity and ECC branches to recover from single or multiple failures.
- **Scalability** It is important to have low overheads in order to support a very large number of lower branches. We propose methods to reduce these overheads.

2.1 Stackable Fan-Out File System

Stackable file systems are a technique to add new features incrementally to existing file systems. As shown in Figure 2.1, the operations of a stackable file system are called by the Virtual File System (VFS) like other file systems. They in turn calls the operations associated with the lower level file system like ext2 or NFS, instead of performing operations directly on a storage device. A stackable file system can be viewed as a thin layer of code in the kernel. It intercepts the file system related calls from the VFS, applies a transformation on the data, like encryption or compression, and then invokes the corresponding operation(s) of the lower level file system before the data is finally written to disk, or returned to the user. Stackable file systems behave like normal file systems from the perspective of the VFS; from the perspective of the underlying file system they behave like the VFS.

FiST is a toolkit for building stackable file systems [45]. It has been extended to support fan-out file systems on Linux [38]. Fan-out stackable file systems differ from linear stackable file systems in that they operate on multiple underlying file systems, or *branches*. Figure 2.2 shows a RAIF file system mounted over several different types of file systems. RAIF intercepts the calls from VFS and passes it on to the lower branches. Data is encrypted by NCryptfs before being sent to an untrusted NFS server. Gzipfs compresses files to save space on an in-memory file system.

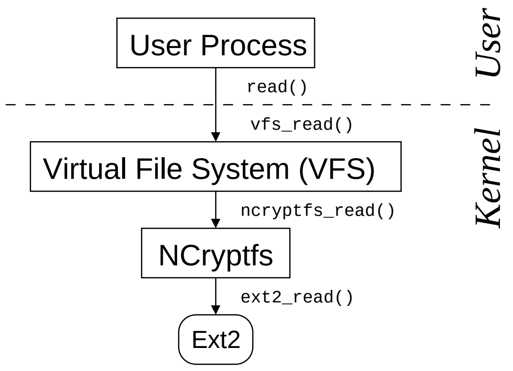

Figure 2.1: Linear stacking
Linear file systems stacking: files are transparently encrypted by NCryptfs before being written to the disk through Ext2.

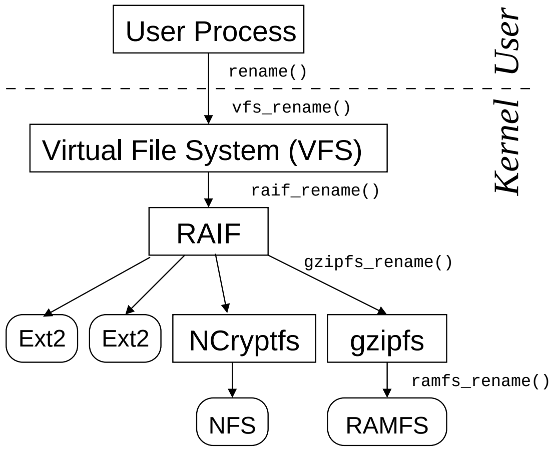

Figure 2.2: Fanout stacking
A possible combination of RAIF fan-out stacking and other file systems stacked linearly.

2.2 RAIF Levels

RAIF duplicates the directory structure on all the lower branches. The data files are stored using different RAIF operations that we call levels, analogous to standard RAID levels. **RAIF0** stripes a file over the lower file systems. The striping unit may be different for different files. This level distributes the accesses to the file among several lower branches. We define **RAIF1** slightly differently from the original RAID level 1 [27]. The original RAID level 1 definition corresponds to RAIF01 described below; RAIF1, on the other hand, duplicates a file on all the branches. In **RAIF4**, parities are calculated for every stripe and stored on a dedicated branch. This level is useful if the parity branch is much faster than the others. **RAIF5** is similar to RAIF4, but the parity branch changes for different stripes as shown in Figure 2.4. In **RAIF6**, extra parity branches are used to recover from two or more simultaneous failures. Some of these levels can be combined: for example, **RAIF01** is a combination of RAIF1 and RAIF0 arrays in such a way that a RAIF0 array is mirrored. (RAIF01 corresponds to the historical definition of RAID level 1 [27].)

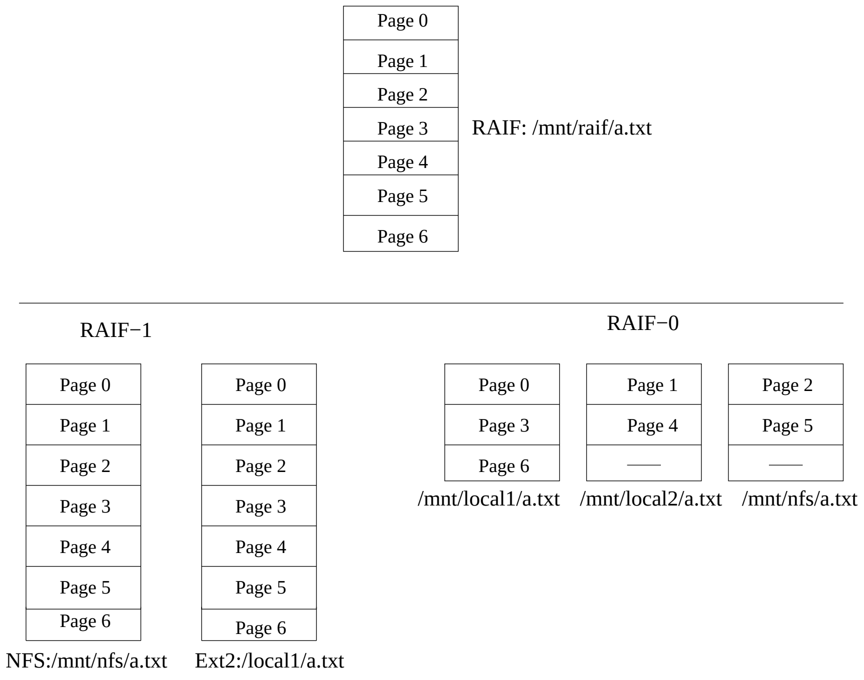

Figure 2.3: RAIF levels
A view of how files are stored on the lower branches in RAIF levels 0 and 1

2.3 Per-file Storage Policies

RAIF supports the notion of per-file storage virtualization by providing the ability to store each file using any combination of storage policies. It allows every file to be stored with any combination of the following rules:

- A file can be stored in any subset of the total configured branches. This feature provides maximum flexibility in assigning the proper branches to the respective files. For example, in a software development environment, all C files and header files can be stored on more reliable branches, probably with a versioning file system below, whereas temporary object files can be stored on less-reliable branches.

- A file can be stored with any RAIF level. For example, for good performance and reliability, multimedia files can be striped across any number of branches, whereas database files can be striped across branches which are in turn mounted on mirrored RAID devices, like RAID 01.

- A file can be stored with any striping unit size. This needs some knowledge of the data access pattern of the file. If the access pattern is mostly serial, like in multimedia files, a bigger striping unit size can be used for such files. The lower level file system usually does a good job of prefetching data, so subsequent sequential requests within the same branch will be serviced faster. However, for files with random I/O patterns, the striping unit size does not matter. However, having a smaller striping unit size overcomes the negative effects of readahead for a random workload to some extent.

2.4 Configuration file

RAIF features a configuration file to make the job of configuring per-file storage policies easy. In general, to achieve per-file storage virtualization, a configuration file can specify rules based on the file size, the file name, the user and group IDs, access frequency, read-write permissions and so on. Based on the nature of classification, there are two kinds of rules which can be specified in the configuration file.

- **Static rules** These are rules which can be applied based on relatively static attributes of a file, like file name and user/group IDs. Using hashing, we can match a large number of rules based on these values quite efficiently.
- **Dynamic rules** These are rules which have to be inferred over a period of time from observed system behavior. For example, the access patterns of files can be used to migrate their RAIF levels to the most optimal configuration. Alternately, for applications whose behavior is more or less fixed, the system could use a training set to infer rules.

2.5 RAIF Meta-Data

Small files may occupy only a portion of a single stripe. To distribute the space utilization and accesses among the branches, we start the stripes of different files on different branches. We call the branch where the file's first data page is located the *starting branch*. The meta information about every file includes the file's starting branch, RAIF level, and striping unit. To delay LOOKUP operations on all except one branch, file size and file attributes are stored together with the RAIF per-file meta-data. The RAIF per-file meta information is stored in the file's *authoritative branch*.

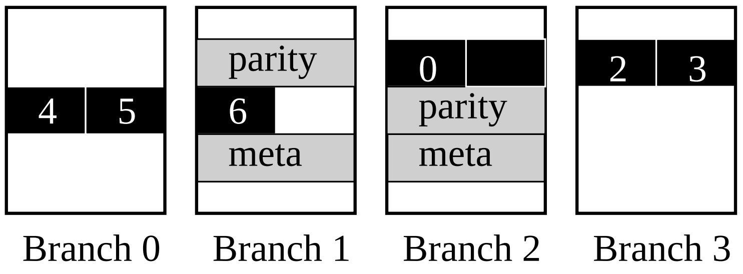

Figure 2.4: RAIF meta-data

RAIF5 file layout on a RAIF mounted over four branches. The file size is 7 pages. Each RAIF striping unit consists of 2 pages. The starting branch is number 2. The authoritative branch is the 1st. The meta-data copy is stored in the 2nd branch. The meta-data size is equal to one disk block (512 bytes) which is usually smaller than the stripe size.

Initially, the authoritative branch and the starting branch of a file are the same, calculated based on the file name. The authoritative branch number may change after a RENAME operation. Therefore, the corresponding meta information has to be moved appropriately. For example, if $hash(old\_file\_name) = 1$ and $hash(new\_file\_name) = 3$ then for a file stored using the RAIF level 4 the meta-data has to be moved from branches 0 and 1 to branches 2 and 3, respectively. Note that the meta-data still contains information that 1 is the starting branch for this file. Therefore, the file can be correctly composed from the stripe even after it is renamed.

Storing per-file metadata is optional. In the absence of metadata, the rules inferred from the configuration file are enough to determine the subset of branches and the starting branch associated with a file. However, upon a rename operation, the entire file has to be moved if the rule for the old and new names has changed. On the other hand, maintaining and updating the metadata information itself may cause overheads.

The problem of storing extra meta information on a per-file basis is well known. However, no universal solution is available up to date. Extended Attributes (EA) associate arbitrary data with files in a file system. Unfortunately, the working group to define an EA API within the POSIX family of standards was unable to reach a common decision and the entire effort was abandoned in 1,998. Some of the file systems that support EAs are compatible with the latest draft of the specification [15], while others are based on older drafts. This resulted in a number of subtle differences among the different implementations. NTFS's *streams* [30] and HFS Plus's *named forks* also associate additional data with files but their APIs are completely different.

2.6 Storage and Access Latency Balancing

RAIF imposes virtually no limitations on the file systems that form lower branches. Therefore, the properties of these lower branches may be substantially different. To optimize the read performance, we integrated a load-balancing mechanism into RAIF which leverages replication to dynamically balance the load. For heterogeneous configurations, the *expected delay* or *waiting time* is often advocated as an appropriate load metric [31]. RAIF measures the times for all read and write operations sent to lower level file systems, and uses their latencies to maintain a per-branch *delay estimate*. The delay estimate is calculated by exponentially averaging the latencies of page and meta-data operations on each individual branch. A good delay estimate can track lasting trends in file system load, without getting swayed by transient fluctuations. We ensure this by maintaining an exponentially-decaying average along with a deviation estimate.

Proportional share load-balancing distributes read requests to the underlying file system branches in inverse proportion to their current delay estimates. This way, it seeks to minimize the expected delay, and maximizes the overall throughput. For this, RAIF first converts delay estimates from each of the underlying branches into per-branch *weights*, which are inversely related to the respective delay estimates. A kernel thread periodically updates a randomized array of branch indexes where each branch's frequency of occurence is proportional to its weight. As RAIF cycles through the array, each branch receives its proportional share of operations.

2.7 Data recovery

RAIF has built-in routines for error correction and data recovery on a page by page basis. In RAIF level 4, a dedicated branch is used to store parity information, and in RAIF level 5, different parity branches are used for different stripes. Regardless of the RAIF level, when a `readpage` operation fails, an array of pages from a particular stripe, including the zeroed page from the failed branch, is presented to a recovery library, which applies the parity information to recover the page from the failed branch. When there is a read failure on a branch, we say that the RAIF mount is operating in degraded mode. In this case, RAIF declares the failed branch as read-only. Files can still be recovered using the parity branches.

Chapter 3

Implementation and current status

The current RAIF prototype consists of 8,293 lines of C code. Out of these, only 2,669 are RAIF specific and 5,624 lines are common for fan-out stackable file systems. The RAIF structure is modular so that new RAIF levels, parity and load balancing algorithms can be added without any changes to the main code. Currently RAIF supports levels 0, 1, 4, 5.

3.1 Fanout Stackable File System Implementation

Fanout stackable file systems transform VFS calls to operate on multiple lower level branches. A branch is nothing but a directory and its collection of files. Since RAIF operates at the file system level, it combines the view of multiple lower branches and provides a unified view through its own mountpoint.

The VFS provides a generic pointer in each of its objects, `file`, `dentry`, `inode` and `super_block` for every file system to store its own private data. RAIF uses of this pointer and allocates a private data block for each of the VFS objects. The following is a list of information that RAIF stores in the private data blocks to achieve its functionality.

- **Number of branches** This is the total number of underlying branches (mountpoints) that RAIF operates upon. This is stored in the superblock. Labels can also be associated with each branch, so that they can be used in the configuration file.

- **Storage rules** To support per-file storage policy configurations, all the rules specified in the `raiftab` configuration file are parsed and linked to the private data block of the superblock. Each rule specifies the subset of branches to use, the RAIF level, the striping unit size, and a list of file extensions to be associated with this rule, as shown in Figure3.1. Rule number 0 is special: it lists all the lower branches, and it is used to create directories.

- **Load balancing information** The latency of every read, write and lookup operation is measured and the running average is stored per lower branch. Since load-balancing can be applied at the granularity of a rule, the latency of a branch is updated in every rule of which it is a part of.

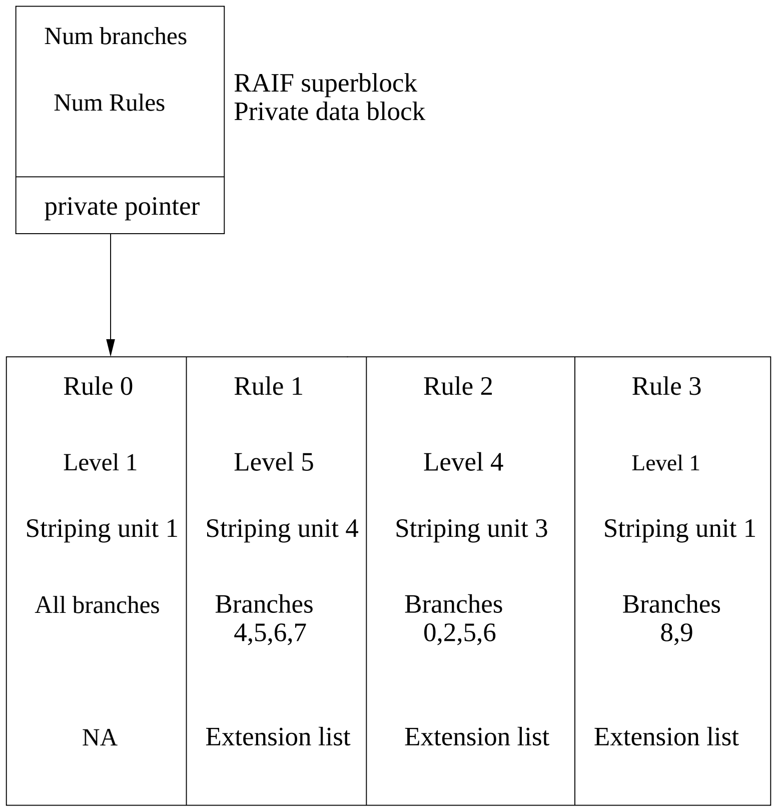

Figure 3.1: RAIF superblock private data
Superblock private data: All information needed for per-file storage policies is stored in it

3.2 Assigning a Policy to a File

Given that the super-block contains a table of rules with file extensions, all that is needed to apply a particular policy to a file is to initialize the private data fields of the VFS objects from the tables. The `lookup` method converts a filename to an inode, caching the `dentry` objects in the `dentry-cache` during the process. By inserting a pattern matching code at relevant places in the `lookup` method, we can determine the rule for that particular file, and initialize its VFS objects with the appropriate storage parameters. For faster lookups, we hash the file extension list associated with each rule into a table which is stored in the data block associated with that rule.

3.3 `raiftab` Configuration File

To make the job of configuring per-file policies easy, RAIF supports a configuration file which is similar to the `raidtab` configuration file for configuring RAID devices. The format of the `raiftab` file is as shown below.

```
raifmnt /mnt/raif
        nr-raif-branches 6
```

```
                raif-branch     fs1
                branch          /branch/nfs1
                raif-branch     ext2-1
                branch          /branch/local1
                raif-branch     ext2-2
                branch          /branch/local2
                raif-branch     ext3-1
                branch          /branch/local3
                raif-branch     versionfs1
                branch          /branch/version
                raif-branch     ncryptfs1
                branch          /branch/secure
nr-raif-rules                   3
raif-rule                       multimedia
                raif-level      1
                stripe-unit     4
                numbranch       2
                branch-ids      nfs1,local1
                files           ".mpeg", ".jpg", ".rm"
raif-rule                       Sw-dev
                raif-level      1
                stripe-unit     1
                numbranch       2
                branch-ids      ext3-1, versionfs1
                files           ".c", ".cpp", ".h"
raif-rule                       default
                raif-level      0
                stripe-unit     1
                numbranch       3
                branch-ids      ext2-1,ext2-2,ext3-1
```

The sample configuration shown above depicts a scenario where multiple file types can be managed easily with the `raiftab` configuration file. The keyword `raifmnt` specifies the RAIF mountpoint. This path is the only one that is visble to all the applications wanting to make use of RAIF as their storage system. The configuration file also specifies a number of rules, each one specified by the `raif-rule` keyword. Currently in our implementation, a rule is the smallest level of configuration granularity. For every rule, the user can specify the RAIF level, the striping unit size, the subset of branches to store the files on, and a list of file extensions or patterns to match against this rule. In our example configuration, the rule labeled `multimedia` is specified for multimedia files, which are mirrored on a local hard-drive and a more reliable NFS server. This gives performance advantages to the user, because there is a local copy of data as well. With the rule labeled `Sw-dev`, source code is mirrored on a local hard-drive and on a versionfs [26] branch. Any update that the user makes to source files will be automatically and transparently versioned by versionfs, providing the most up to date version of a file. The rule labeled `default` is matched against all those files that do not match any of the above rules, and it is configured for performance using RAIF0, but not necessarily reliability.

3.4 Asynchronous operation support

An area of concern specific for fan-out stackable file systems is the sequential execution of VFS requests. It dramatically increases latency of VFS operations that require synchronous accesses to several branches. For example, RAIF5 synchronously reads data and parity pages before a small write operation can be initiated. RAIF also buffers pages at its level. To avoid double buffering, the data pages should be shared not only between lower and upper file systems but also between several lower file systems and an upper one. We are currently working on solving this. However, it is important to understand that it only increases the latency of certain file system operations while this has little impact on the aggregate RAIF performance under a workload generated by many concurrent processes.

Our current development efforts are concentrated on the performance enhancements of the general fan-out templates. We are exploring the effects of delaying some VFS operations on non-authoritative branches. In the future, we plan to add support for EAs, dynamic adjustment of RAIF levels and other storage policies, and provide advanced data recovery procedures in the kernel.

Chapter 4

Evaluation

This section describes the performance of the current RAIF prototype. Performance measurements were made with different RAIF levels and different number of branches. Scenarios with a combination of local and network file systems were also tested for performance.

We conducted our benchmarks on a 1.7GHz Pentium 4 machines with 1GB of RAM. The machine was equipped with four Maxtor Atlas 15,000 RPM 18.4GB Ultra320 SCSI disks formatted with Ext2. It was running Linux kernel version 2.6.13.3. We used autopilot [39] to carry out the benchmarks. Autopilot was configured to format the lower file systems before every run. The lower file systems were remounted before every benchmark run to purge the page cache. We ran each test at least 3 times and obtained mean values for the elapsed, system, user and wait times. Wait time, the elapsed time less CPU time used, consists mostly of I/O, but process scheduling can also affect it. In each case, the half-widths of the confidence intervals were less than 5% of the mean.

We ran the following two benchmarks:

- Postmark [18] simulates the operation of electronic mail servers. It performs a series of file appends, reads, creations, and deletions. We configured Postmark to create 20,000 files, between 512–10K bytes, and perform 200,000 transactions. Create, delete, read, and write operations were performed with equal probability.

- RANDOM-READ is a benchmark designed to evaluate RAIF under a heavy load of random data read operations. It spawns 32 child processes and concurrently reads 32,000 randomly-located 512 byte blocks from 16GB files. The load on the lower branches fluctuates because of the randomness of the read pattern. In particular, a branch may be idle for some time, if all the reading processes have sent their requests to the other branches. Our experiments showed that 32 processes are sufficient to make these random fluctuations negligible.

We call a test configuration RAIF-*N*BR, where *N* is the number of branches. We call RAIF*L* a RAIF file system where all files are stored using RAIF level *L*.

Figure 4.1 shows the RANDOM_READ benchmark results for plain Ext2 and RAIF5 with 1,2, 3 and 4 branches. It is clearly an I/O intensive workload, with over 99% of the time being spent in I/O wait time. Since CPU time is so minimal, the elapsed times for Ext2 and RAIF5 with one branch are almost the same. The I/O time is dominated by the disk seek time. The elapsed time decreases well with the increase in number of branches. The benchmark runs 1.5 times faster with RAIF-2BR than with RAIF-1BR, 1.8 times faster with RAIF-3BR and 1.95 times faster with RAIF-4BR.

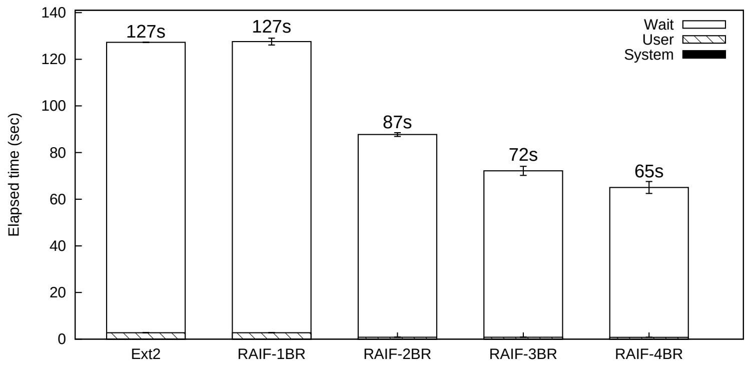

Figure 4.1: RAIF5 RANDOM_READ
RANDOM_READ *benchmark results for plain Ext2 and RAIF5*

Figure 4.2 shows the RANDOM_READ benchmark for two heterogeneous branches mounted with RAIF1. One branch is an Ext2 branch on a fast SCSI disk, and the other branch is an Ext2 branch on a slower IDE disk (Western Digital 30GB IDE disk, 7,000 rpm with 2MB on-disk cache). Without load-balancing, the read requests always go to the first disk, which is quite slow. Compared to RAIF5-1BR, RAIF1-2BR is 18.2% slower because the requests go to the slower disk. Compared to RAIF5-2BR, it is 72.5% slower. However, with load balancing turned on, the read requests are proportionally distributed among the two disks, and hence compared to RAIF5-2BR, the overhead drops to 12%. Since the RANDOM_READ benchmark issues 1,000 requests, and the IDE disk is 18% slower than the SCSI disk, the I/O wait times are more or less justified, because it is a random workload and there will be some variations.

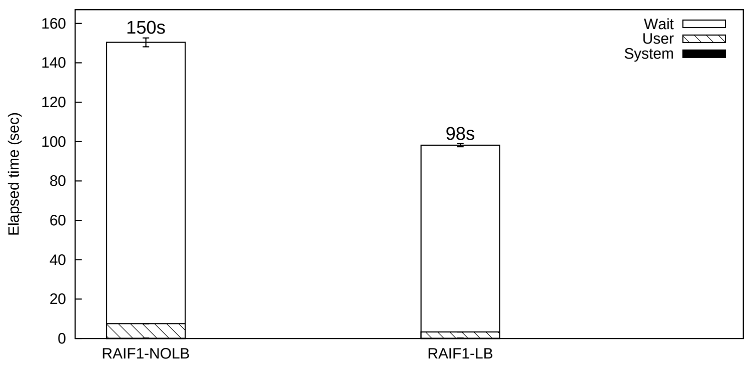

Figure 4.2: RAIF1 RANDOM_READ *with load balancing*
RANDOM_READ *for heterogeneous branches and with load-balancing*

Postmark is the second I/O-intensive benchmark we ran. Figure 4.3 shows the result for RAIF0.

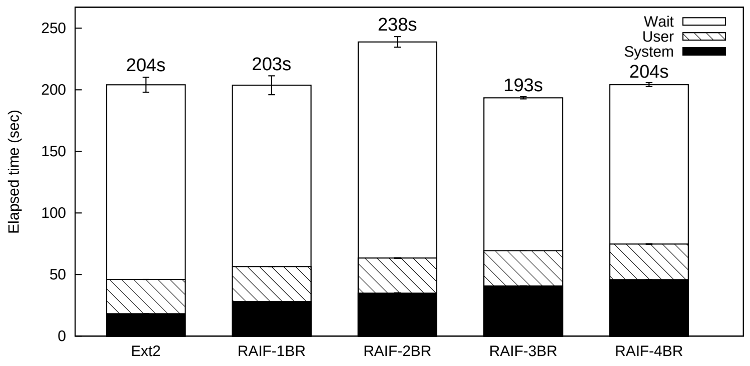

Figure 4.3: RAIF0 Postmark

Postmark benchmark results for plain Ext2 and RAIF0 mounted on 1,2,3 and 4 Ext2 branches

The system time overheads increase linearly because operations like OPEN, CLOSE and LOOKUP are performed serially on all branches. . Compared to a plain Ext2 branch, the system overhead for RAIF0-1BR is 30.5%, for RAIF0-2BR is 90%, for RAIF0-3BR is 122% and RAIF0-4BR is 138%.

For a small number of rules specified in the configuration file, the overheads are almost negligible. We are in the process of designing an extensible policy-matching scheme which can be used to include more attributes than just file names.

Our current RAIF prototype has modest system time overheads. It reduces elapsed time considerably for I/O-intensive workloads by balancing and distributing the load of lower branches. Especially for random I/O workloads, the benefits increase linearly with the number of branches that are added. However, system time optimizations are needed to improve scalability and decrease latency of individual file system operations.

Chapter 5

Related Work

Striping and replication of data are well known methods used to improve performance and fault-tolerance of secondary storage. RAID [27], which was introduced in the late 80's, had configurations for different combinations of striping, mirroring, and parity-based failure recovery. Traditional RAID is composed of homogeneous components. However, algorithms were later proposed for supporting heterogenuous components in RAID [7]. Striping is primarily used for parallelizing and load balancing read and write requests across different devices to improve performance. The original RAID was proposed as a hardware-level mechanism where the striping and mirroring logic was embedded into a special hardware device known as the RAID controller. Later, RAID was implemented at the device driver level [4] so that ad-hoc configurations were possible without requiring specialized hardware components. RAIF performs RAID-like operations at a layer above even device drivers: at the file system level.

There are several implementations of RAID-like file server systems that operate over a network [1, 10, 14], including implementations that combine remote and local drives [9]. However, past systems targeted particular usage scenarios and mostly had fixed architectures. Zebra is a distributed file system that uses standard network protocols for communications between its components [10]. Zebra uses only file-based and log-based striping with parity. In contrast, RAIF's stacking architecture allows it to utilize the functionality of existing file systems and to support a variety of configurations without any modifications to the source code. Media distribution servers use data striping and replication to distribute the load among servers [1, 6]. The stripe unit size and degree of striping have been shown to influence the performance of these servers [32]. RAIF effectively mixes striping with other data placement techniques as needed.

Data grids require high availability and efficiency [41]. The choice of data placement and management schemes plays a crucial role in realizing these goals [20, 33]. Data grids are typically composed of highly heterogeneous storage and network resources. Fault-tolerance and dynamic re-configuration are key for successful operation in such scenarios [21]. Grids also implement several mechanisms, like striping, streaming, and on-demand caching [24], to efficiently serve a wide range of access patterns. Even the early RAIF implementation realized these goals by bringing the rich set of RAID configurations to the file system level [16].

RAIF provides virtualization at a per-file level. Policies regarding the RAID method to be adopted can be chosen at the granularity of a file type. The idea of using different RAID levels for different data access patterns was used in several projects at the driver [9] and hardware levels [37]. However, the lack of higher-level information forced the developers to make decisions based solely on statistical information. D-GRAID [34] uses a Semantically-Smart Disk System [35] to infer file system semantics at the disk level and tailor its internal RAID schemes for different classes of file system data. It performs *selective metadata replication*, where metadata blocks are replicated to a higher degree than raw data blocks, and *fault-isolated data placement*, where semantically related blocks are placed within the storage array's unit of fault-containment. By doing this, D-GRAID achieves a graceful decline of availability in the event of multiple disk failures. Timothy et al. developed Exposed RAID (E$\times$RAID [8]) to enable file systems to tailor their storage management mechanisms by revealing information about parallelism and failure isolation boundaries, track performance, and failure characteristics to the file systems. They also built a file system called *Informed Log-structured file system* (I.LFS) that uses E$\times$RAID to provide functionalities like dynamic load balancing, user control of file replication, and delayed replication of files for improved performance.

RAIF offers file-level virtualization as a stackable file system and hence does not require explicit information exchange from the storage system to the file system. Initial RAIF Solaris ZFS [36] performs dynamic striping across disks to maximize throughput, using the notion of virtual storage pools. The components in a storage pool can be hetergenous and disks can be added or removed from the storage pools thereby dynamically expanding or shrinking the file system size.

RAIF is implemented as a stackable file system [43]. Stackable file systems are not new [44]. Originally introduced in the early 90s [29], stackable file systems were propsed to augment the functionality of existing file systems by transparently mounting a layer above them. Normal stackable file systems mount on top of a single lower level file system. Several stackable file systems exist today that provide functionalities like encryption [40], versioning [25], tracing [2], intrusion detection [17], and more. A class of stackable file systems known as *fan-out* mount on top of more than one file system to provide useful functionality [11, 29]. However, so far the most common application of fan-out has been unioning [3, 12, 19, 28, 38]. RAIF is a stackable fan-out file system that can mount on top of many underlying file systems, to provide RAID-like functionality.

Replication in RAIF uses proportional-share load balancing using the expected delay as the load metric. This approach is generally advocated for heterogeneous systems [31]. However, when the workload includes a mix of random and sequential operations, the number of I/O operations performed may be a more suitable load metric [22]. Traffic shaping systems allow better control over the bandwidth utilization and allow bandwidth dedication for particular users and applications [5].

Chapter 6

Conclusions

RAIF combines high flexibility, portability, reliability, and simplicity. This makes it a general solution to many existing file system architecture problems. Like RAID systems, RAIF can provide improved data survivability, data management and performance. For read requests, it can recover a corrupted file on-the-fly using parity. RAIF supports the notion of per-file storage virtualization by allowing users the freedom to store any file with any combination of storage parameters, including the set of branches, RAID level, and striping unit size. For example, RAIF can dynamically place more important data on more secure but possibly slower or smaller file systems. RAIF provides load balancing by directing write and read operations to the most appropriate lower file systems.

By default, RAIF uses a set of rules derived from its configuration file to assign storage properties to individual files. The rules can be based on the patterns for file names or locations, file owners, and attributes. Also, users can change storage properties on the per-file basis. In that case RAIF stores extra information called TAG for every file. This results in more flexible configurability but also decreases RAIF performance.

RAIF distributes requests among lower branches and thus decreases the average I/O time. However, it adds system time overheads that linearly increases with the number of branches. For a small number of branches, RAIF has modest overheads and frequently operates faster than single-storage file systems.

6.1 Future Work

In the future, we plan to improve performance and usability of RAIF:

- We are exploring a number of techniques to improve RAIF's efficiency, including delayed writes, simultaneous writes to all branches, zero-copying, and advanced page caching.
- We plan to implement hot-spare branch support. While a recovery mechanism, like kernel thread or a user daemon recovers data from the failed branch to the hot-spare branch, the file system can continue to service read requests using the recovery library. Write requests can be directed to the hot-spare branch.
- We plan to implement a checkpointing mechanism so that the recovery process does not have to be restarted if interrupted.

- We are designing better ways to store meta-data information. The overheads associated with renaming files and changing their storage policies can be reduced if we store per-file metadata in the form of tags or Extended Attributes.
- We are trying to make more informed storage decisions not only based on file names, but also based on other attributes like user IDs, permissions, and access patterns.

Bibliography

[1] S. Anastasiadis, K. Sevcik, and M. Stumm. Maximizing Throughput in Replicated Disk Striping of Variable Bit-Rate Streams. In *Proceedings of the Annual USENIX Technical Conference*, pages 191–204, Monterey, CA, June 2002.

[2] A. Aranya, C. P. Wright, and E. Zadok. Tracefs: a file system to trace them all. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 129–143, San Francisco, CA, March/April 2004. USENIX Association.

[3] AT&T Bell Laboratories. *Plan 9 – Programmer's Manual*, March 1995.

[4] A. Brown and D. Patterson. Towards Availability Benchmarks: A Case Study of Software RAID Systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 263–276, San Diego, CA, June 2000. USENIX Association.

[5] T. Chiueh, K. Gopalan, A. Neogi, C. Li, S. Sharma, S. Shan, J. Chen, W. Li, N. Joukov, J. Zhang, F. Hsu, F. Guo, and S. Doong. Sago: A Network Resource Management System for Real-Time Content Distribution. In *Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS'02)*, pages 557–562, National Central University, Taiwan, ROC, December 2002.

[6] C. Chou, L. Golubchik, and J. C. S. Lui. Striping doesn't scale: How to achieve scalability for continuous media servers with replication. In *International Conference on Distributed Computing Systems*, pages 64–71, Taipei, Taiwan, April 2000.

[7] T. Cortes and J. Labarta. Extending Heterogeneity to RAID level 5. In *Proceedings of the Annual USENIX Technical Conference (ATC)*, pages 119–132, Boston, MA, June 2001. USENIX Association.

[8] T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Bridging the information gap in storage protocol stacks. In *Proceedings of the Annual USENIX Technical Conference*, pages 177–190, Monterey, CA, June 2002. USENIX Association.

[9] K. Gopinath, N. Muppalaneni, N. Suresh Kumar, and P. Risbood. A 3-tier RAID storage system with RAID1, RAID5, and compressed RAID5 for Linux. In *Proceedings of the FREENIX Track at the 2000 USENIX Annual Technical Conference*, pages 21–34, San Diego, CA, June 2000. USENIX Association.

[10] J. Hartman and J. Ousterhout. The Zebra Striped Network File System. In *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 29–43, Asheville, NC, December 1993. ACM.

[11] J. S. Heidemann and G. J. Popek. File system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, February 1994.

[12] D. Hendricks. A Filesystem For Software Development. In *Proceedings of the USENIX Summer Conference*, pages 333–340, Anaheim, CA, June 1990.

[13] J. H. Howard. An Overview of the Andrew File System. In *Proceedings of the Winter USENIX Technical Conference*, February 1988.

[14] L. Huang, G. Peng, and T. Chiueh. Multi-dimensional Storage Virtualization. In *Proceedings of the 2004 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 14–24. ACM Press, June 2004.

[15] IEEE/ANSI. Information Technology–Portable Operating System Interface (POSIX)–Part 1: System Application Program Interface (API)—Amendment: Protection, Audit, and Control Interfaces [C Language]. Technical Report STD-1003.1e draft standard 17, ISO/IEC, October 1997. *Draft was withdrawn in 1997.*

[16] N. Joukov, A. Rai, and E. Zadok. Increasing distributed storage survivability with a stackable raid-like file system. In *Proceedings of the 2005 IEEE/ACM Workshop on Cluster Security, in conjunction with the Fifth IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2005)*, pages 82–89, Cardiff, UK, May 2005. IEEE. (**Won best paper award**).

[17] A. Kashyap, S. Patil, G. Sivathanu, and E. Zadok. I3FS: An In-Kernel Integrity Checker and Intrusion Detection File System. In *Proceedings of the 18th USENIX Large Installation System Administration Conference (LISA 2004)*, pages 69–79, Atlanta, GA, November 2004. USENIX Association.

[18] J. Katcher. PostMark: a new filesystem benchmark. Technical Report TR3022, Network Appliance, 1997. `www.netapp.com/tech_library/3022.html`.

[19] D. G. Korn and E. Krell. A New Dimension for the Unix File System. *Software-Practice and Experience*, 20(S1):19–34, June 1990.

[20] T. Kosar and M. Livny. Stork: making data placement a first class citizen in the grid. In *International Conference on Distributed Computing Systems*, March 2004.

[21] Erwin Laure. The Architecture of the European DataGrid. Technical report, The European DataGrid Project Team, March 2003. `www.twgrid.org/event/isgc2003/ISGC_pdf/The_Architecture_of_EDG.pdf`.

[22] Ixora Pty Ltd. Disk load balancing. `www.ixora.com.au/tips/tuning/disk_load.htm`.

[23] Y. Miretskiy, A. Das, C. P. Wright, and E. Zadok. Avfs: An On-Access Anti-Virus File System. In *Proceedings of the 13th USENIX Security Symposium (Security 2004)*, pages 73–88, San Diego, CA, August 2004. USENIX Association.

[24] R. W. Moore, I. Terekhov, A. Chervenak, S. Studham, C. Watson, and H. Stockinger. Data Grid Implementations. Technical report, Global Grid Forum, January 2002. `www.ppdg.net/docs/WhitePapers/Capabilities-grids.v6.pdf`.

[25] K. Muniswamy-Reddy. Versionfs: A versatile and user-oriented versioning file system. Master's thesis, Stony Brook University, December 2003. Technical Report FSL-03-03, `www.fsl.cs.sunysb.edu/docs/versionfs-msthesis/versionfs.pdf`.

[26] K. Muniswamy-Reddy, C. P. Wright, A. Himmer, and E. Zadok. A Versatile and User-Oriented Versioning File System. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 115–128, San Francisco, CA, March/April 2004. USENIX Association.

[27] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the ACM SIGMOD*, pages 109–116, Chicago, IL, June 1988. ACM Press.

[28] J. S. Pendry and M. K. McKusick. Union mounts in 4.4BSD-Lite. In *Proceedings of the USENIX Technical Conference on UNIX and Advanced Computing Systems*, pages 25–33, New Orleans, LA, December 1995. USENIX Association.

[29] D. S. H. Rosenthal. Evolving the Vnode interface. In *Proceedings of the Summer USENIX Technical Conference*, pages 107–118, Anaheim, CA, June 1990. USENIX Association.

[30] M. Russinovich. Inside Win2K NTFS, Part 1. `www.winnetmag.com/Articles/ArticleID/15719/pg/2/2.html`, November 2000.

[31] B. Schnor, S. Petri, R. Oleyniczak, and H. Langendorfer. Scheduling of parallel applications on heterogeneous workstation clusters. In *Proceedings of PDCS'96, the ISCA 9th International Conference on Parallel and Distributed Computing Systems*, pages 330–337, Dijon, France, September 1996.

[32] P. Shenoy and H. M. Vin. Efficient striping techniques for variable bit rate continuous media file servers. Technical Report UM-CS-1998-053, University of Massachusetts at Amherst, 1998.

[33] A. Shoshani, A. Sim, and J. Gu. Storage Resource Managers: middleware components for grid storage. In *Proceedings of the Nineteenth IEEE Symposium on Mass Storage Systems*, April 2002.

[34] M. Sivathanu, V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving storage system availability with D-GRAID. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 15–30, San Francisco, CA, March/April 2004. USENIX Association.

[35] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-smart disk systems. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 73–88, San Francisco, CA, March 2003. USENIX Association.

[36] Sun Microsystems, Inc. Solaris ZFS file storage solution. *Solaris 10 Data Sheets*, 2004. `www.sun.com/software/solaris/ds/zfs.jsp`.

[37] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID Hierarchical Storage System. *ACM Transactions on Computer Systems*, 14(1):108–136, February 1996.

[38] C. P. Wright, J. Dave, P. Gupta, H. Krishnan, E. Zadok, and M. N. Zubair. Versatility and Unix Semantics in a Fan-Out Unification File System. Technical Report FSL-04-01b, Computer Science Department, Stony Brook University, October 2004. `www.fsl.cs.sunysb.edu/docs/unionfs-tr/unionfs.pdf`.

[39] C. P. Wright, N. Joukov, D. Kulkarni, Y. Miretskiy, and E. Zadok. Auto-pilot: A Platform for System Software Benchmarking. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, pages 175–187, Anaheim, CA, April 2005. USENIX Association.

[40] C. P. Wright, M. Martino, and E. Zadok. NCryptfs: A secure and convenient cryptographic file system. In *Proceedings of the Annual USENIX Technical Conference*, pages 197–210, San Antonio, TX, June 2003. USENIX Association.

[41] William Yurcik, Xin Meng, Gregory A. Koenig, and Joseph Greenseid. Cluster Security as a Unique Problem with Emergent Properties: Issues and Techniques. In *5th LCI International Conference on Linux Clusters*, May 2004.

[42] E. Zadok, J. M. Anderson, I. Bădulescu, and J. Nieh. Fast indexing: Support for size-changing algorithms in stackable file systems. In *Proceedings of the Annual USENIX Technical Conference (ATC)*, pages 289–304, Boston, MA, June 2001. USENIX Association.

[43] E. Zadok and I. Bădulescu. A stackable file system interface for Linux. In *LinuxExpo Conference Proceedings*, pages 141–151, Raleigh, NC, May 1999.

[44] E. Zadok, R. Iyer, N. Joukov, G. Sivathanu, and C. P. Wright. On incremental file system development. *ACM Transactions on Storage (TOS)*, 2(2):161–196, May 2006.

[45] E. Zadok and J. Nieh. FiST: A language for stackable file systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 55–70, San Diego, CA, June 2000. USENIX Association.