Increasing Distributed Storage Survivability with a Stackable RAID-like File System

Nikolai Joukov, Abhishek Rai, and Erez Zadok
Stony Brook University

Appears in the 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). Won the Best Paper Award.

Abstract

We have designed a stackable file system called Redundant Array of Independent Filesystems (RAIF). It combines the data survivability properties and performance benefits of traditional RAIDs with the unprecedented flexibility of composition, improved security, and ease of development of stackable file systems. RAIF can be mounted on top of any combination of other file systems including network, distributed, disk-based, and memory-based file systems. Existing encryption, compression, antivirus, and consistency checking stackable file systems can be mounted above and below RAIF, to efficiently cope up with slow or unsecure branches. Individual files can be distributed across branches, replicated, stored with parity, or stored with erasure correction coding to recover from failures on multiple branches. Per-file incremental recovery, storage type migration, and load-balancing are especially well suited for grid storages.

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

1 Introduction

Redundant Array of Independent Filesystems (RAIF) is the first RAID-like storage system designed at the file system level that imposes virtually no restrictions on the underlying stores and allows per-file storage policy configuration.

For many years, grouping hard disks together to form RAIDs has been considered a key technique for improving storage survivability and increasing data access bandwidth [19]. 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 RAID. RAID configurations are fixed and are the same for all the files because hardware and software RAIDs operate at the data-block level, where high level meta-information is not available. This results in inefficient storage utilization when important data is stored with the same redundancy level as less-important data. Other common RAID limitations are related to long–term maintenance. For example, data recovery is slow and may require a restart from the very beginning if interrupted.

There are several implementations of RAID-like file server systems that operate over a network [7, 25], including implementations that combine network and local drives [6]. However, past systems targeted some particular usage scenario and had a fixed architecture. Inflexibilities introduced at design time often result in suboptimal 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 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 [34]. They allow incremental addition of features and can be dynamically loaded as external kernel modules. Stackable file systems overlay 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 [8, 23] and was recently included in the FiST [30, 34] 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, 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 [31], data-integrity verification [11], an antivirus [17], and a compression file system [33]. These file systems can be mounted over RAIF as well as below it over only some slow or untrusted branches.

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. Dynamic RAIF-level migration offers additional benefits. For example, a RAIF mounted over a RAM-based file system and several network file systems can serve as a caching file system if the file storage method changes depending on the file usage frequency.

Third, administration is easier because files are stored on ordinary unmodified lower-level file systems. Therefore, the size of these lower file systems can be changed, they can be easily backed up using standard software. The data is easier to recover in the case of failure because it is stored in a more accessible format.

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

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. Section 2 outlines the design of RAIF. Section 3 describes the current status of the project, some interesting implementation details, and outlines future directions. Section 4 presents an evaluation of the current RAIF prototype. Section 5 discusses related work. We conclude in Section 6.

2 Design

2.1 Stackable fan-out file system

Stackable file systems are a technique to layer new functionality on existing file systems. As seen in Figure 1, a stackable file system is called by the Virtual File System (VFS) like other file systems, but in turn calls another file system instead of performing operations on a backing store such as a disk or an NFS server [20]. Before calling the lower-level file system, stackable file systems can modify the operation, for example encrypting data before it is written to disk. 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.

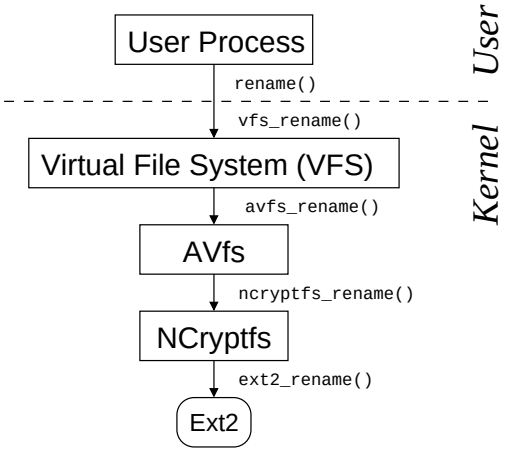

Figure 1: Linear file systems stacking: files are transparently checked for viruses by AVfs and encrypted by NCryptfs before being written to the disk through Ext2.

FiST is a toolkit for building stackable file systems [34]. Recently it has been extended to support fan-out file systems on Linux [30]. Fan-out stackable file systems differ from linear stackable file systems in that they call multiple underlying file systems, or *branches*. Figure 2 shows a RAIF file system mounted over several different types of file systems.

2.2 RAIF levels

RAIF duplicates the directory structure on all of the lower branches. The data files are stored using different RAIF methods 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 [19]. The original RAID level 1 definition corresponds to RAIF01 described below; RAIF1, on the other hand, duplicates the files on all of 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 3. In **RAIF6**, extra parity branches are used to recover from two or more simultaneous failures. Some of these levels can be combined together. For example, **RAIF01** is a combination of the RAIF1

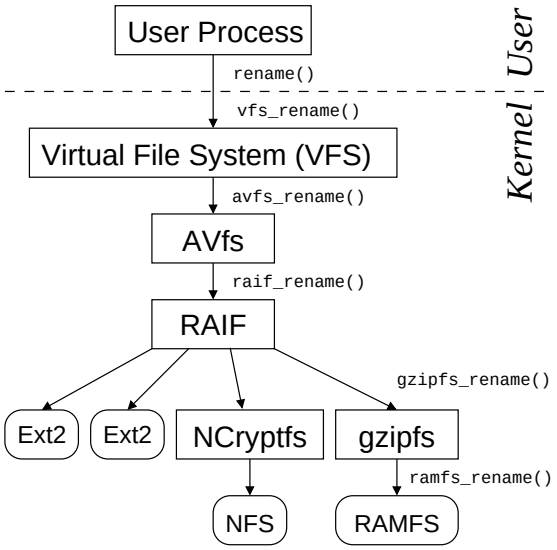

Figure 2: A possible combination of RAIF fan-out stacking and other file systems stacked linearly: files are checked for viruses by AVfs before they are stored by RAIF; data is encrypted by NCryptfs before being sent to an untrusted NFS server; and gzipfs compresses files to save space on an in-memory file system.

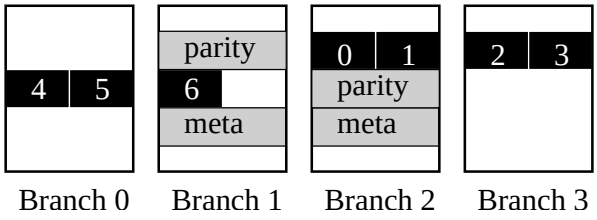

Figure 3: RAIF5 file layout on a RAIF mounted over four branches. The file size is 7 pages. Each RAIF striping unit consists of 2 pages. The starting branch is number 2. The authoritative branch is the 1st. The meta-data copy is stored in the 2nd branch. The meta-data size is equal to one disk block (512 bytes) which is usually smaller than the stripe size.

and RAIF0 arrays in such a way that a RAIF0 array is mirrored. (RAIF01 corresponds to the historical definition of RAID level 1 [19].)

2.3 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*. The only information available about the file for a file LOOKUP operation is the file name. Therefore, it is natural to calculate the authoritative branch number based on a hash of the file name. Because the RAIF meta information is essential to reading and writing the file, the meta-data is replicated according to the RAIF level. Thus, RAIF meta information is stored only on the authoritative branch for RAIF0, on all the branches for RAIF1, and on authoritative and parity branches for RAIF levels 4, 5, and 6. Note that 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.

The problem of storing extra meta information on a per-file basis is well known. However, no universal solution is available up to date. Thus, 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 1998. Some of the file systems that support EAs are compatible with the latest draft of the specification [10], while others are based on older drafts. This resulted in a number of subtle differences among the different implementations. NTFS's *streams* [24] and HFS Plus's *named forks* [1] also associate additional data with files but their APIs are completely different.

We have implemented two methods to store the extra per-file meta information. In one method, we maintain an additional file with meta information for each data file. However, we found that this method has a number of deficiencies arising from complicated meta file-related pointers management that results in non-trivial execution paths in the VFS. In addition, it results in a large number of open and created files. In an alternative approach, we store the RAIF meta information in the additional data block at the beginning of the data file itself in the corresponding branch or branches. Files in other branches have a one page hole at the beginning. In both cases, these extra files or data pages are not visible to the user. Currently, the second method is the default one. Finally, we also plan to use EAs at least on the file systems that support them.

2.4 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 differ-

ent. To optimize the read performance, we integrated a load-balancing mechanism into the RAIF that leverages replication to dynamically balance the load. For heterogeneous configurations, the *expected delay* or *waiting time* is often advocated as an appropriate load metric [26]. 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 maximize 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 has a representation in proportion to its weight. As RAIF cycles through the array, each branch receives its proportional share of operations.

Currently, we assign file storage properties at the file's creation time based on the file's type. Next, we plan to implement automatic RAIF level migration to dynamically balance the storage utilization and data access latency and bandwidth. For example, a file replicated among the memory file system and a network file system may be removed from memory if it is rarely used and there is no space left on the memory file system.

2.5 Data recovery

Currently, we have a user level *fsck* program that recovers the information offline in case of a branch failure. The program accesses the data on lower branches directly and generates the data on the replaced branch. In the future, we will implement a kernel recovery thread that will automatically start to regenerate the data on a hot-swap branch. It will recover the missing parts of the files and checkpoint the progress so that the recovery process does not have to be restarted if interrupted.

3 Implementation and current status

The current RAIF prototype consists of 7,273 lines of C code. Out of these, only 1,649 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, and a stub module for level 6.

RAIF has high performance and scalability demands. Therefore, we had to change the stackable file system templates accordingly. Traditionally, stackable file systems buffer the data twice. This allows us to keep both modified (e.g., encrypted or compressed) and unmodified data in memory at the same time and thus save considerable amounts of CPU time. However, RAIF does not modify the data pages. Therefore, double caching does not provide any benefits but makes the page cache size effectively half its original size. Unfortunately, the VFS architecture imposes constraints that make sharing data pages between lower and upper layers complicated. In particular, the data page is a VFS object that belongs to a particular inode and uses the information of that inode at the same time. We are still working on this problem. So far we have found a relatively simple but not a straightforward solution to the problem. We use data pages associated with the upper inode for *both* the lower and upper file system layers. Specifically, the data pages belong to the upper inode but are assigned to lower-level inodes for the short duration of the lower-level page-based operations. Here is an example of the modified VFS `readpage` operation:

```
page->mapping = lower_inode->i_mapping;
err = lower_inode->i_mapping->a_ops->
    readpage(lower_file, page);
page->mapping = upper_inode->i_mapping;
```

We analyzed the Linux kernel functions that directly or indirectly use inode and cache page connectivity and found that in all these cases, the above modification works correctly. We tested the resulting stackable file system on a single-CPU and multi-CPU machines under the compile and I/O-intensive workloads. No races or other problems were observed.

Another problem 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. This problem is related to the previous one because sometimes, 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 this problem. 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. In addition to the two problems of double buffering and sequential VFS operations execution described above, we are also working on the overall reduction of CPU overheads to increase the system scalability. 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.

4 Preliminary Evaluation

We have evaluated RAIF performance during different stages of the development process, to identify possible problems early and update the design as needed. In this section we describe the performance of the current RAIF prototype with data stored using levels 0 and 1.

We conducted our benchmarks on two 1.7GHz Pentium 4 machines with 1GB of RAM. The first machine was equipped with four Maxtor Atlas 15,000 RPM 18.4GB Ultra320 SCSI disks formatted with Ext2. The second machine was used as an NFS server. It had two 10GB Seagate U5 IDE drives formatted with Ext2. Both machines were running Red Hat 9 with a vanilla 2.4.24 Linux kernel and were connected via a dedicated 100Mbps link. We remounted the lower file systems before every benchmark run to purge the page cache. We ran each test at least ten times and used the Student-t distribution to compute 95% confidence intervals for the mean elapsed, system, user, and wait times. Wait time is the elapsed time less CPU time used and 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 [12] 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 and read/write operations were selected 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 of the different 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-NBR, where N is the number of branches. We call RAIFL a RAIF file system where all files are stored using RAIF level L.

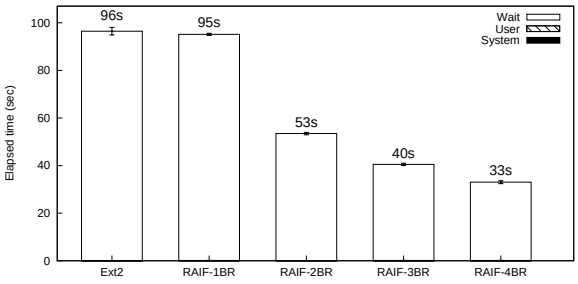

Figure 4: RANDOM-READ *benchmark results for plain Ext2 and RAIF0 mounted over 1, 2, 3, and 4 branches.*

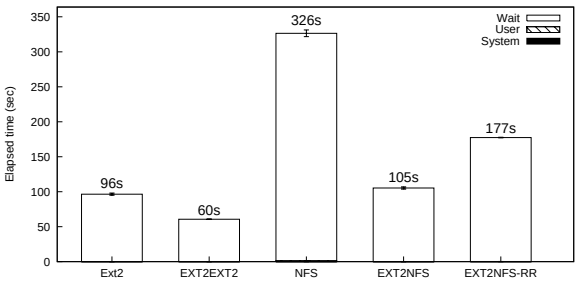

Figure 5: RANDOM-READ *results for RAIF1 mounted over NFS and Ext2. RAIF1 performance is compared to plain Ext2, plain NFS, and RAIF1 with a round-robin balancing policy (*EXT2NFS-RR*) configuration.*

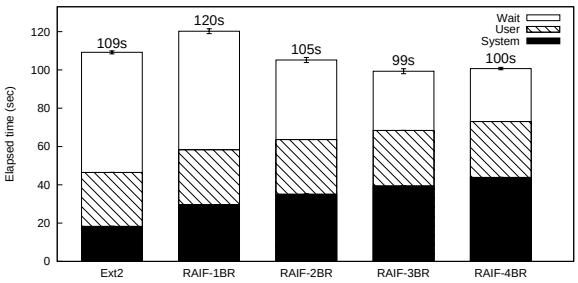

Figure 6: Postmark benchmark results for a plain Ext2 file system and RAIF0 mounted over 1, 2, 3, and 4 branches.

Figure 4 shows the benchmark results for plain Ext2 and RAIF0 mounted over 1, 2, 3, and 4 lower Ext2 file system branches. The I/O time accounts for more than 99.9% of the execution time. The execution times of Ext2 and RAIF-1BR are statistically indistinguishable. Since the total amount of data fetched from the disks is only 16MB, the SCSI and PCI buses were idle most of the time. The I/O time is mostly defined by the time of the disk seek operations. Therefore, the elapsed benchmark execution time decreases well with the increase of the number of branches. Compared to Ext2, the benchmark executes 1.8 times faster with RAIF-2BR, 2.4 times

faster with RAIF-3BR, and 3.0 times faster with RAIF-4BR.

In the case of RAIF1, the RANDOM-READ benchmark shows the effectiveness of proportional-share load-balancing in distributing read loads between identical replicas. The results can be seen in Figure 5. The EXT2EXT2 configuration, which has RAIF1 mounted over two EXT2 file systems, distributes read loads between two EXT2 branches and is 38% faster than Ext2. The EXT2NFS configuration has RAIF1 mounted over an EXT2 branch and an NFS branch, and is a case of RAIF1 over disparate file systems. Proportional share load-balancing beats naive round-robin in this case.

Postmark is the second I/O-intensive benchmark we ran. We ran it with a striping unit size of 8KB that is smaller than the optimal value but is about the average file size used during the test. This makes about half of the files span across multiple branches. Figure 6 shows the result for RAIF0. The I/O time decreases slightly slower than in the random-read case because operations such as LOOKUP, READDIR, OPEN, CLOSE etc. are performed sequentialy on all the branches. Compared to Ext2, the I/O time of RAIF0-2BR is 1.5 times less, 2.0 times less for RAIF0-3BR, and 2.3 times less for RAIF0-4BR. There is a 62% system time overhead of RAIF0-1BR for this workload. There is an additional overhead of 24% for any additional lower branch. Since the I/O time improves sublinearly and system time overhead grows linearly as a function of the number of branches, there is a branch number where these two effects compensate for each other. Under the Postmark workload, RAIF0-4BR starts to behave slower than the RAIF0-3BR. Compared to Ext2, RAIF0-2BR performs 3.7% faster, RAIF0-3BR performs 9.0% faster, and RAIF0-4BR performs 7.8% faster.

So far we demonstrated that our current RAIF prototype has modest system time overheads. It improves elapsed time considerably for I/O-intensive workloads by balancing and distributing the load of lower branches. However, further system time optimizations are needed to improve scalability and decrease latency of individual file system operations.

5 Related Work

Data grids have high availability and efficiency requirements [32]. The choice of data placement and management schemes plays a crucial role in realizing these goals [14, 28].

Data replication is a commonly-used strategy. It improves data survivability and response time, provides load balancing, and offers better bandwidth utilization [4, 22]. Replication in RAIF uses proportional-share load balancing using the expected delay as the load metric. This approach is generally advocated for heterogeneous systems [26]. 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 [16].

Grids also implement several other storage mechanisms, like striping, streaming, and on-demand caching [18], to efficiently serve a wide range of access patterns. Media distribution servers use data striping and replication to distribute the load among servers [5, 25]. The stripe unit size and degree of striping have been shown to influence the performance of these servers [27]. RAIF effectively mixes striping with other data placement techniques on a need basis.

Data Grids typically comprise of highly heterogeneous storage and network resources. Fault-tolerance and dynamic re-configuration are key to a successful operation in such scenarios [15]. RAIF realizes these goals by bringing the rich set of RAID configurations to the file system level through our stackable fan-out file system infrastructure. Fan-out file systems themselves were proposed before [8, 23]. However, so far the most common application of fan-out has been unioning [3, 9, 13, 21, 30].

Stackable file systems can be mounted on top of each other. Existing encryption [31], data integrity verification [11], antivirus [17], compression [33], and a tracing [2] stackable file systems can be mounted on top or below RAIF.

The idea of using different RAID [19] levels for different data access patterns was used in several projects at the driver [6] and hardware levels [29]. However, the lack of higher-level information forced the developers to make the decisions based solely on statistical information.

Zebra is a distributed file system that uses standard network protocols for communications between its components [7]. Zebra uses per-file RAID levels and striping with parity. In contrast, RAIF's stacking architecture allows it to utilize the functionality of existing file systems and to create a variety of configurations without any modifications to the source code.

6 Conclusions

Our RAIF architecture holds much promise. Its high flexibility, portability, and simplicity make it a general solution to many existing file system architecture problems. RAIF can provide improved data survivability, data management, and can be easily and efficiently integrated with existing and future security tools. We have designed it to be extensible with plugin formulas and parameters that determine the RAIF personality.

We are currently implementing several RAIF6 multi-failure recovery methods. We are designing an automatic per-file RAIF levels assignment and RAIF level

dynamic migration. For this purpose, we plan to use statistical information about file accesses and information about the current storage state. We are developing a more reliable in-kernel data recovery mechanism. We are working on the general performance and scalability improvements of the stackable fan-out infrastructure. 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.

Acknowledgments

We thank the committee for its comments. We would also like to thank all the developers of the basic stackable fan-out file system template: Jay Pradip Dave, Puja Gupta, Harikesavan Pathangi Krishnan, Gopalan Sivathanu, Charles P. Wright, and Mohammad Nayyer Zubair.

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

References

[1] Apple Computer, Inc. HFS Plus Volume Format. Technical Report Note TN1150, March 2004. `http://developer.apple.com/technotes/tn/tn1150.html`.

[2] A. Aranya, C. P. Wright, and E. Zadok. Tracefs: A File System to Trace Them All. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 129–143, San Francisco, CA, March/April 2004.

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

[4] William H. Bell, David G. Cameron, Ruben Carvajal-Schiaffino, A. Paul Millar, Kurt Stockinger, and Floriano Zini. Evaluation of an economy-based file replication strategy for a data grid. In *Proceedings of the 3rd International Symposium on Cluster Computing and the Grid*, pages 661–668. IEEE Computer Society, 2003.

[5] 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, 2000.

[6] 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.

[7] 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.

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

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

[10] 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.*

[11] 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.

[12] J. Katcher. PostMark: A New Filesystem Benchmark. Technical Report TR3022, Network Appliance, 1997. `www.netapp.com/tech_library/3022.html`.

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

[14] 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.

[15] 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`.

[16] Ixora Pty Ltd. `www.ixora.com.au/tips/tuning/disk_load.htm`. Technical report.

[17] 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.

[18] 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`.

[19] 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, June 1988.

[20] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS version 3 design and implementation. In *Proceedings of the Summer USENIX Technical Conference*, pages 137–52, Boston, MA, June 1994.

[21] 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, December 1995.

[22] K. Ranganathan and I. Foster. Identifying Dynamic Replication Strategies for a High Performance Data Grid. In *Proceedings of the International Grid Computing Workshop*, November 2001.

[23] D. S. H. Rosenthal. Evolving the Vnode interface. In *Proceedings of the Summer USENIX Technical Conference*, pages 107–18, Summer 1990.

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

[25] M. Stumm S. Anastasiadis, K. Sevcik. 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.

[26] 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, 1996.

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

[28] 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.

[29] 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.

[30] 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`.

[31] 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.

[32] 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.

[33] 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*, pages 289–304, Boston, MA, June 2001.

[34] 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.