RAIF: Redundant Array of Independent Filesystems

Nikolai Joukov[1,2], Arun M. Krishnakumar[1], Chaitanya Patti[1], Abhishek Rai[1],
Sunil Satnur[1], Avishay Traeger[1], and Erez Zadok[1]
[1] *Stony Brook University,* [2] *IBM T.J. Watson Research Center*

Appears in the proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies (MSST 2007)

Abstract

Storage virtualization and data management are well known problems for individual users as well as large organizations. Existing storage-virtualization systems either do not support a complete set of possible storage types, do not provide flexible data-placement policies, or do not support per-file conversion (e.g., encryption). This results in suboptimal utilization of resources, inconvenience, low reliability, and poor performance.

We have designed a stackable file system called Redundant Array of Independent Filesystems (RAIF). It combines the data survivability and performance benefits of traditional RAID with the flexibility of composition and ease of development of stackable file systems. RAIF can be mounted on top of directories and thus on top of any combination of network, distributed, disk-based, and memory-based file systems. Individual files can be replicated, striped, or stored with erasure-correction coding on any subset of the underlying file systems.

RAIF has similar performance to RAID. In configurations with parity, RAIF's write performance is better than the performance of driver-level and even entry-level hardware RAID systems. This is because RAIF has better control over the data and parity caching.

1 Introduction

The importance of files varies greatly. If lost, some files can be easily regenerated, some can be regenerated with difficulty, and some cannot be regenerated at all. The loss of some files, such as financial data, can be catastrophic. Therefore, files must be stored with varying levels of redundancy using data replication or erasure-correction codes. File storage is a complicated set of trade-offs between data protection, performance, data management convenience, and special infrastructure requirements. For example, some files must be shared by many users, whereas some are only available locally on read-only media; some files must be encrypted; and some can be efficiently compressed. At the same time, every possible storage subcomponent, such as a disk or a remote server, has a set of unique features. Therefore, an ideal storage-virtualization system must support most physical and logical storage devices, as well as support various per-file data placement policies and per-file special features such as compression and encryption.

Hardware-virtualization systems are limited to specific types of lower storage devices. Driver-level storage virtualization systems support more device types, but the number of local and SAN device types is still limited. Device-level and driver-level storage virtualization systems, such as hardware and software RAID [34], operate on raw data blocks without knowledge of meta-data. This means that they use the same storage policies for all files. At most, they try to use statistical information about data accesses in an attempt to increase performance [47]. This also means that information on individual disks in an array has no meaning and little or no information can be extracted if too many disks fail.

File-system–level storage virtualization systems still support only limited types of storage; many support just one. This is because they have to explicitly support specific NAS and other protocols as well as special features such as compression and encryption.

Stackable file systems are a useful and well-known technique for adding functionality to existing file systems [53, 54]. 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 use the lower file system's operations and data. Most stackable file systems in the past have assumed a simple one-to-one mapping: the stackable file system was layered on top of one lower directory on a single file system. A different class of stackable file systems that use a one-to-many mapping (fan-out) has previously been suggested [16, 38] and was recently developed from the FiST [48] templates.

We have developed a fan-out RAID-like stackable file system called Redundant Array of Independent Filesystems (RAIF). RAIF is a storage-virtualization system that imposes virtually no restrictions on the underlying stores and allows versatile per-file and per-directory storage policies. RAIF derives its usefulness from three main features: flexibility of configuration, access to high-level information, and easier administration.

First, because RAIF is stackable, it can be mounted over directories and, thus, over any combination of lower file systems. For example, it can be mounted over network file systems like NFS and Samba, AFS distributed file systems, and local file systems—all at the same time. RAIF leverages the well-maintained code and protocols of existing file systems. In fact, RAIF can even utilize *future* file systems and protocols. Stackable file systems can be mounted on top of each other. Existing stackable file systems offer features like encryption [50], data-integrity verification [27], virus checking [32], versioning [33], tracing [3], and compression [51]. These file systems can be mounted over RAIF as well as below it.

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 block-level storage virtualization systems. This meta-data allows the optimization of redundancy schemes, data placement, and read-ahead algorithms. For example, RAIF can stripe large multimedia files across different branches for performance, and concurrently use two parity pages for important financial data files that must survive even two failures. Dynamic RAIF-level migration allows one to optimize RAIF reliability, resource utilization, and performance.

Third, administration is easier because files are stored on unmodified lower-level file systems. Therefore, the size of these lower file systems can be changed, and they can be backed up easily using standard software. The data is easier to recover in the case of failure because it is stored in files with the same names as their upper-level counterparts. For example, if the files are replicated on several disks, then every such disk will contain an ordinary file system with the same files intact.

A common limitation of software RAIDs is that data recovery is slow and may require a restart from the very beginning if it is interrupted. RAIF performs storage recovery on a per-file basis and can even do it in the background, concurrently with normal activity. If irreparable inconsistencies are detected, RAIF can identify the specific files affected, which is more user-friendly than the lists of block numbers offered by block-level systems.

Like other data striping and replication systems, RAIF can improve performance for multi-process workloads. Even for I/O-intensive single-process workloads that require many synchronous operations, RAIF's performance is comparable with the performance of driver-level software RAID systems. For RAIF configurations similar to standard RAID4 and RAID5, RAIF can outperform driver-level and even some hardware implementations thanks to better control over file system caches. Moreover, most other storage-virtualization systems choose a single data-placement policy, which is not optimal for all files. RAIF can optimize performance for all files because it can use better data-placement policies for individual files or groups of files.

The rest of the paper is organized as follows. Section 2 outlines the design of RAIF. Section 3 describes some interesting implementation details. Section 4 presents an evaluation of RAIF. Section 5 discusses related work. We conclude and outline future directions in Section 6.

2 Design

Existing storage-virtualization systems are limited in three ways: (1) they only virtualize a limited class of storage devices; (2) they support one or a small number of data placement policies, such as replication on a subset of lower storage devices or striping with parity; and (3) they provide few (usually no) per-file data management features such as compression, consistency validation, or encryption. Even so, existing storage-virtualization systems are complex, expensive, and hard to maintain. This is because traditional storage-virtualization systems must *explicitly* support different types of lower storage types and features.

Stackable file systems are a technique for layering new functionality on top of existing file systems. As seen in Figure 1, a stackable file system's methods are called by the Virtual File System (VFS) as with other file systems, but in turn it uses another file system instead of performing operations on a backing store such as a disk or an NFS server [35, 53]. Before calling the lower-level file system's methods, a stackable file system can modify the operation or the data. 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. Fan-out stackable file systems differ from linear stackable file systems in that they call multiple underlying file systems, or *branches*.

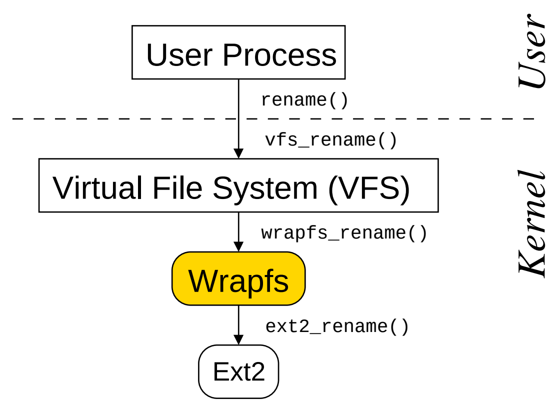

Figure 1: Linear file system stacking

Redundant Array of Independent Filesystems (RAIF) is a stackable fan-out file system. We designed it with flexibility in mind. RAIF can be mounted over *any* set of file systems. This means that it does not have to support all possible storage types and protocols explicitly: it can leverage the code of dozens of existing file systems and NAS protocols. RAIF will even be able to reuse *future* file systems and protocols without modification. If necessary, RAIF can still be mounted over standard RAID arrays or any other storage-virtualization system. In addition, existing and future stackable file systems can be mounted above RAIF or over any lower branches. This allows one to add important extra features such as compression or encryption in a selective way. Figure 2 shows an example RAIF file system mounted over several different types of lower storage. In this figure, letters below RAIF denote the branch labels as follows:

C A read-only CD-ROM file system.

E A set of Ext3 file systems mounted over local disks.

V A versioning stackable file system mounted over an NFS client.

N An encryption stackable file system mounted over an NFS client. All data stored on this remote branch is automatically encrypted.

G A TmpFS Linux file system that stores data in virtual memory or in the swap area. Because virtual memory may be scarce, the branch data is compressed with a stackable compression file system.

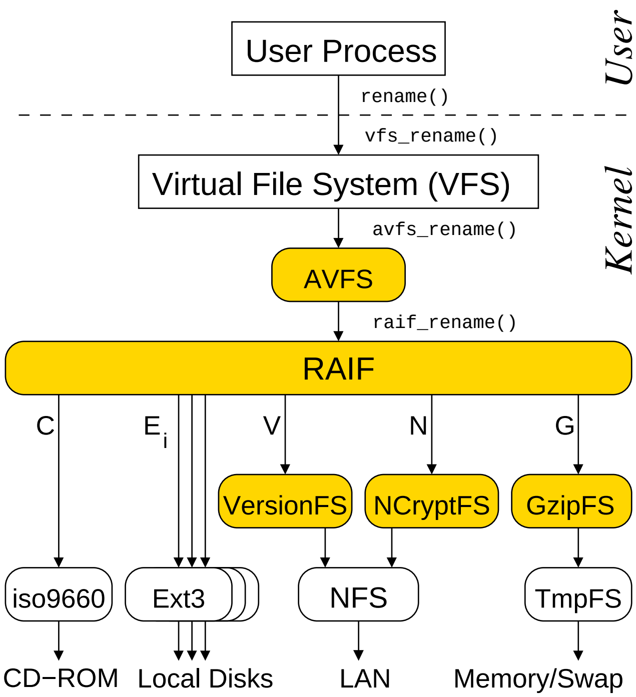

Figure 2: A possible combination of RAIF fan-out stacking and other file systems stacked linearly. Letters below RAIF denote the branch labels.

2.1 RAIF Data Placement

RAIF is designed at a high level of abstraction and leverages well-maintained, external code. Therefore, we were able to concentrate on storage virtualization issues rather than support for different types of lower storage devices, placement of individual blocks on these devices, and NAS or SAN protocols. RAIF duplicates the directory structure on all of the lower branches. The data files are stored using different RAIF modes that we call *levels*, analogous to standard RAID levels:

- **RAIF0** The file is striped over a set of lower file systems. The striping unit may be different for different files. As we describe below, this level also allows us to distribute entire files across branches.
- **RAIF1** The file is replicated over a set of branches. Note that we define RAIF1 slightly differently from the original RAID level 1 [34]. The original RAID level 1 definition corresponds to RAIF01.
- **RAIF4** The file is striped as in RAIF0 but a dedicated branch is used for parity. This level is useful for heterogeneous sets of branches, for example if the parity branch is much faster than the others and the workload has many writes, or if the parity branch is much slower and the workload is read-biased.
- **RAIF5** This level is similar to RAIF4, but the parity is rotated through different branches.
- **RAIF6** In RAIF6, extra parity branches are used to recover from two or more simultaneous failures.
- **RAIF01** This level is a mirrored RAIF0 array.
- **RAIF10** Similarly to RAIF01, a mirrored array is striped over a set of branches.

Small files may occupy only a portion of a single stripe. To distribute space utilization and accesses among branches, we start the stripes of different files on different branches. We use the term *starting branch* to denote the branch where the file's first data page is located. By default, the starting branch is derived from the file name using a hash function. Also, users may specify starting branches explicitly.

Another use of the starting branch is to randomize the data placement of entire files. In particular, one may use RAIF0 with a large striping unit size. In this case, every small file will be entirely placed on its starting branch whereas large files will be striped. For RAIF levels 4, 5, and 6, small files will be stored on their starting branches and their parity will be stored on parity branches. Essentially, small files will be automatically replicated on two or three branches whereas large files will be striped. Also, it is possible to specify a special striping width value (–1) to force files to be stored on their starting branch independently of their size. This is useful to replicate data on two random branches (level 5) or one random and one or two fixed branches (levels 4 and 6).

RAIF can assign levels and striping unit sizes to files and directories individually, or by applying regular expressions to file names. For example, if users' home directories are to be stored on a RAIF mounted over a set of branches as shown in Figure 2, one may use the following global rules:

- Use RAIF1 to replicate `*.c`, `*.h`, `*.doc`, and other important files on all local drives (branches $E_0 ... E_N$) and branch V. This way, important files are replicated locally, stored remotely, and also stored with versions at the same time.

- Use RAIF4 to stripe large multimedia files on all local drives and keep parity data on the remote NFS server (branch N). This allows us to save precious space on the local drives and still keep parity to recover files in case of a single disk failure. Note that multimedia files are usually not changed frequently. Therefore, the remote server is seldom contacted for common read operations.
- Use branch G for intermediate files (e.g., `*.o`). This way these files are available for repeated compilations but are purged if not used for a while.
- Use RAIF1 over branches C and G to provide the illusion of writing to a read-only file system. As we will describe in Section 2.3, RAIF provides a simplified unification functionality [48]. For example, if a CD-ROM contains source code, one may compile it directly from the CD-ROM. Resulting files will be put onto the G branch and users will see a merged set of files in a single directory.
- Use RAIF0 or RAIF5 on the E branches for all other files as they are either regenerable or unimportant.

In addition, individual users may be allowed to define special rules on a per-file or per-directory basis.

2.2 RAIF Rules

The meta-information about every file includes the file's starting branch, RAIF level, striping unit, and the set of branches used to store the file. Since the file name is the only information available for a file LOOKUP operation, RAIF needs to maintain a mapping of file names to meta-information to perform LOOKUP operations.

In an early RAIF prototype [23], we used a hash function to find the branch with per-file meta-information. However, this approach is not suitable for a storage-virtualization system in which any subset of branches can be used to store a file: RAIF needs to know the file storage configuration in order to determine the subset of branches used to store the file. RAIF's meta-information storage is based on two assumptions: first, we assume that RAIF meta-information does not change frequently; second, we assume that the number of storage policies is much smaller than the number of files. Therefore, every storage policy is associated with a regular expression to match file names. A regular expression, a RAIF level, a striping unit size, and a set of branches together define a RAIF *rule*. A `lookup`, then, consists of matching a file name with a rule and looking up the file in the returned set of branches. Rules are stored in special files on a subset of lower branches. These files are not visible from above RAIF and are read at directory lookup time. This approach allows RAIF to decrease both system time and I/O overheads. Directories with no rules contain no rule files, which is the common case. Also, storing rule files close to the files they relate to increases data locality on the lower branches and thus improves performance. RAIF supports the following three types of rules:

- **Global** rules are inherited by lower directories. They can be added or removed only manually.
- **Local** rules are not inherited by lower directories and have higher priority than global rules. Generally, these rules are necessary for exception handling. For example, they can be used to define special policies for an individual file. However, the most common use of local rules is to handle RENAME and rule-change operations, as we will describe in Section 2.2.2.
- **Meta** rules specify policies for storing RAIF rule files. This way, these special rule files can be replicated on all or only a subset of branches to balance reliability and performance. Also, meta rules are useful if some branches are mounted read-only. Finally, this allows one to dedicate branches for storing RAIF rules. For example, such branches can be networked file systems to allow sharing of the rules between servers. Dedicated branches are thus similar to dedicated meta-data servers in cluster storage systems [1]. Regular expressions in meta rules are matched only against lower directory names.

Let us again consider the example RAIF configuration shown in Figure 2, and assume that we want to store a small directory tree as shown in Figure 3. As we discussed in Section 2.1, we may want to have a global rule for the topmost directory, `raif`, to stripe all multimedia files on the local hard disks and keep their parity on a remote branch (RAIF4 configuration). However, one may want to add a special local rule to the `dir1` directory to replicate `birth.avi` on all the local drives and a remote server because that file contains especially precious data.

2.2.1 Rule Matching

Upon a file LOOKUP operation, RAIF matches the file name against the regular expressions in all relevant rule files. Rules in every directory are arranged sequentially according to their priority. Local rules have the highest priority and are tried first. Global rules in every directory have precedence over global rules of the parent directory. This way, every rule has a unique priority value. Therefore, the rule-matching process proceeds by scanning the rules sequentially, starting from the highest-priority local rule in the current directory and finishing at the lowest-priority global rule from the RAIF mount-point directory. Because we assume that rules are not changed frequently, RAIF can use finite-state machines (e.g., using the Aho-Corasick pattern-matching algorithm [2]) to avoid sequential scanning.

Let us now consider the LOOKUP operation in more detail. First, RAIF matches the file name against all relevant

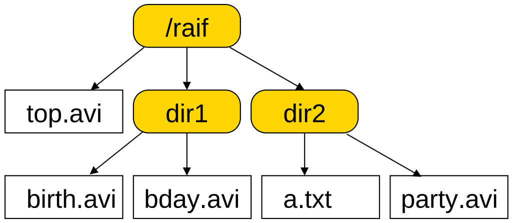

Figure 3: A sample directory tree.

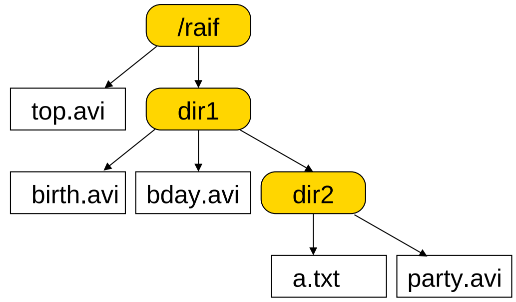

Figure 4: Same directory tree with `dir2` moved inside `dir1`.

rules using the finite automaton for the current directory. Out of the returned set of rules, RAIF selects the highest-priority rule and looks up the file on one of the branches according to that rule. After this LOOKUP operation, RAIF knows if the lower file exists and if it is a regular file or a directory. If the file exists, RAIF looks up the rest of the branches. In case of a directory LOOKUP, RAIF reads and parses its rules if they exist. If the file being looked up does not exist, RAIF returns from the LOOKUP operation. The rest of the branches are looked up later, upon calls to file or directory CREATE functions, because this is the earliest opportunity for RAIF to know the type of these files. This allows RAIF to minimize the number of LOOKUP calls sent to the lower file systems.

2.2.2 Rule Management

Rules for groups of files allow RAIF to perform with minimal overheads for common file system operations because only a minimal amount of extra meta-information requires maintenance in memory and on the lower branches. This is possible because RAIF rules are kept intact most of the time. Let us consider the three situations in which rule changes are required.

Rename operations change the name of the file. Therefore, the rule that matches a file may be different before and after it is renamed. Note that this is not likely because most renames do not change the logical meaning of the file being renamed. For example, it is unlikely that the source file `main.c` would be renamed to a video file `main.avi`. However, the location of the starting branch may change even in the common case, which is important if the file is striped. Therefore, if the old storage policy does not correspond to the new file name, RAIF has two options: (1) add a local rule for the new file (the regular expression field is the actual new file name), or (2) store the file according to its new name. RAIF adds a local rule for large files and moves the data for small files. The file-size threshold is configurable. Upon an UNLINK operation of a file, the corresponding local rule is deleted if it exists.

Moving a directory may require an update of the rules for that directory. For example, Figure 4 shows the directory tree of Figure 3 with directory `dir2` moved into `dir1`. We can see that if `dir1` has any rules, then these rules could conflict with the placement of the files in the old `dir2`. In that case, RAIF would add the necessary global rules to `dir2`. Note that the number of rules to add is usually small. In the worst case, the global rules from the directories above may require an addition to a single directory.

Manual addition, removal, and modification of rules by users is not a frequent operation. However, it may result in conflicts if existing files match the new rule and the new rule does not correspond to the old storage policy. Again, RAIF offers two possible options: (1) moving the files' data; and (2) adding local rules for affected files. The first option may be useful for storage reconfiguration. Rule changes usually correspond to administrators' desire to modify the storage configuration. For example, an administrator may want to increase the storage capacity and add another local disk to the RAIF configuration shown in Figure 2. In that case, administrators will change the existing rule to stripe data across more branches, and may want to migrate the data for all existing files as well. The second option, adding local rules, corresponds to the standard technique of lazy storage migration. In particular, new files will be stored according to the new rules whereas old files will be gradually migrated if deleted or completely overwritten.

Rename and directory-movement operations both require changes to rules or files in only one directory. Rule additions, removals, and modifications may require recursive operations and are therefore performed partially in user mode. In particular, a user-level utility traverses the directory tree bottom-up and gradually updates it by calling the kernel-level directory rule-update routines. Per-directory updates are not recursive, and RAIF performs them in the kernel to avoid races. As an example rule-update scenario, consider the directory tree shown in Figure 3. Let us assume that we want to add a global rule to directory `raif` with the regular expression `a.txt`. First, the user-level program for rule addition will issue an `ioctl` call to ask the kernel component to add the rule to directory `dir1`. Because that directory has no files that match the regular expression `a.txt`, RAIF will just add that global rule to `dir1` and return. Second, the user-level program will do the same for `dir2`, which has a file that matches the new regular expression. Depending on the file size or additional program flags, RAIF either adds a local rule for that file or moves its data. (Storage policies must be changed only in that case.) If RAIF moves the data, it first creates local rules for every matched file in a directory to ensure that the files' data is always consistent with the rules. Only after doing this, RAIF removes these temporary local rules, adds a new global rule to `dir2`, and returns. Third, the user-level program calls RAIF to add a rule to directory `raif`. Finally, RAIF adds the global rule to `raif` and removes the temporary global rules from `dir1` and `dir2`.

2.3 Merging and Unification Semantics

RAIF levels that stripe the files can be used to merge several small file systems together to form a single, larger file system. Similar to Chunkfs [19], such merging decreases the total consistency checking time. RAIF can merge any number of file systems without having to modify their source code or on-disk data layouts. RAIF can also store parity locally or remotely to increase data reliability.

RAIF's design ensures that files stored on several branches are visible as a unified name-space from above. This allows RAIF to join several file systems together similar to Unionfs [48]. Thus, one can run programs that require read-write access to the directory structure from a read-only CD-ROM or a DVD-ROM. For example, one may create rules to store rule files and all the new files on the writable branches only. This way, one can compile programs from read-only media directly, or temporarily store useful files on RAM-based file systems.

2.4 RAIF Administration

RAIF's rule-changing process is useful in many cases, such as when adding or removing RAIF branches, or when increasing or decreasing available storage space. However, the rule-changing process may be lengthy so it is wise to estimate the storage space changes beforehand. RAIF has user-level utilities for this purpose.

RAIF has a user-level Perl script that accepts high-level descriptions of file types and information about the relative importance of files. This script can change rules and even migrate data to increase or decrease storage space utilization given this information. For example, an administrator could use the script to free N gigabytes of space when necessary [55]. This script can perform automatic storage reconfiguration if periodically started by a `cron` job. For example, RAIF can store all files with replication while space is plentiful but gradually change the reliability of data up to the lowest allowed level as space becomes scarce.

Also, user-mode tools can initiate delayed conversion of file storage policies. For example, RAIF can create local rules for files affected by renaming or rule-changing operations during high system load times. Later, the `cron` daemon can initiate conversion of these files during periods in which the system is underutilized.

This design allows RAIF to be simple in the kernel and operate with clear and deterministic rules. At the same time, more sophisticated and abstract policies can be processed and defined at the user level.

2.4.1 RAIF Backup and Security

RAIF stores files on lower file systems as regular files. This provides three main benefits. First, it allows incremental backups on a per-branch basis. These backups can be restored on individual failed branches. Second, RAIF files have the same owners and attributes as their counterparts on lower branches. Similarly, rule files have the same owners and access permissions as their parent directories. This means that users not allowed to write to a directory are not allowed to create rules in that directory because they cannot create rule files there. Third, administrators can set up quotas for the lower file systems and control the usage of these individual branches. This could be used to discourage users from setting up rules to replicate all their files.

2.5 Performance

RAIF is a file system, and as such is located above file system caches. Therefore, it can efficiently use all available memory for data and meta-data caching. For example, both data and parity are automatically cached in the page cache [13]. This allows RAIF to outperform driver-level and even some hardware RAID controllers (with small caches) under a write-oriented workload.

RAIF takes advantage of its access to high level information to perform per-file performance optimizations. For example, it only calculates parity for newly-written information. Block-level RAID systems do not have sufficient information and have to calculate parity for whole pages or blocks. Worse yet, they have to read the old parity even for small file appends.

Standard read-ahead algorithms on layered storage systems can be not just suboptimal but can even degrade performance [43]. RAIF can optimize its read-ahead policies for different RAIF levels and even files.

2.5.1 Load-Balancing

RAIF imposes virtually no limitations on the file systems that form the lower-level branches. Therefore, the properties of these lower branches may differ substantially. To deal with this, RAIF supports homogeneous and heterogeneous storage policies. For example, RAIF4 assumes that a dedicated parity branch is different from the others, whereas RAIF5 assumes that all lower-level branches are similar. Starting-branch selection can be randomized to balance load for all branches or fixed to one branch to force small files be placed on that branch.

To optimize read performance, we integrated a dynamic load-balancing mechanism into RAIF1. The *expected delay* or *waiting time* is often advocated as an appropriate load metric [39] for heterogeneous configurations. The per-branch *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 and a deviation estimate.

Proportional share load-balancing distributes read requests to underlying file system branches in inverse proportion to their current delay estimates. This way, it seeks to minimize expected delay and maximize overall throughput. To implement this, RAIF first converts delay estimates from each underlying branch 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 is represented in proportion to its weight. As RAIF cycles through the array, each branch receives its proportional share of operations.

2.6 Reliability

If one or several branches fail, RAIF can either operate in a degraded mode or recover the data on-line or off-line.

On-line recovery is performed similarly to the rule-management procedures. That is, a user-mode program recursively traverses directories bottom-up and invokes the kernel-mode per-directory recovery procedure. During a file's recovery, RAIF either migrates the file's storage policy to store the file on fewer branches or regenerates missing information into a new branch. During this recovery process, RAIF can serve requests and recover missing data pages on demand.

Off-line recovery is performed by a user-level *fsck* program. The program accesses the lower branches directly and generates the data on the replacement branch. The program can also recover individual files if necessary.

In both cases, RAIF reports files that cannot be recovered instead of less meaningful block numbers as reported by block-level RAIDs.

2.6.1 Resynchronization

A notorious problem of most software and hardware RAID systems that do not use non-volatile memory is the write hole problem. In particular, concurrent writes (i.e., writes of the same information to several branches for RAID1 or writing of data and associated parity in the case of RAIF5) reach permanent storage in an undefined order unless performed synchronously and sequentially. Synchronous sequential writes significantly degrade performance and are not used in practice. Unordered writes may result in an inconsistent state after a power failure if, say, a parity block is written to the disk but the associated data update is not.

RAIF creates a hidden file called `.raif_status` in every branch when RAIF is mounted and removes that file on unmount. This file is not exposed to users. This allows RAIF to identify power-failures or other events that can potentially result in data inconsistencies by checking for the existence of that file during the RAIF mount operation. If the `.raif_status` file is detected, RAIF starts the data resynchronization process in the background. It recalculates parity for all files and, for RAIF1 it makes the data on all the branches identical. This procedure guarantees that, after it is complete, the data and the parity on all the branches are consistent. However, (similar to other software RAIDs) it does not guarantee that the latest write completely or partially succeeds.

Like rule management, per-directory resynchronization is performed in the kernel whereas recursive operations are performed by our user-level tool. During the resynchronization process, RAIF works as usual except that all operations result in resynchronization of the requested data and meta-data if they are not yet resynchronized. On-line resynchronization is possible because RAIF is mounted over lower file systems which are already in a consistent state by the time RAIF is mounted. For example, journals on lower file systems are used by their respective journalling file systems to restore a consistent state. The `.raif_status` file is not removed on unmount if the RAIF self-verification is not finished. The `.raif_status` file contains the current RAIF verification position. This allows RAIF to continue its self verification the next time it is mounted, or restart it in case of another power failure.

RAIF performs the verification on-line because it operates over other consistent-state file systems. In practice, this is a useful property, given that self-verification can take days or even longer for large arrays [10].

2.6.2 Reliability of RAIF vs. RAID

One of the major differences between RAIF and lower-level (driver or hardware) RAID systems is that RAIF operates above file system caches. One may think that this delays writes in memory and results in lower reliability. However, existing file systems mounted over RAID systems delay writes the same way before the RAID is called to write the information. Therefore, in both cases writes are delayed the same way, if one considers the whole file system stack.

Simultaneous power failure and a branch failure may results in *silent data corruption* [4] for parity-based redundancy schemes. For this to happen (1) a parity write must succeed; (2) the data write must fail; and (3) some other branch that is used to store this file must fail. In that case, the parity does not correspond to the data any more and will produce invalid data if used to recover the data on the failed branch. Similarly, as described above, big writes (if interrupted) may result in the situation when some blocks of a file contain previous information and some new. These problems exist for most driver-level RAID systems and even hardware RAID controllers not equipped with non-volatile memory. Nevertheless, such RAID controllers are widely used because these problems do not happen too often.

Out of several existing solutions for the problem, journalling of data and meta-data writes in a non-volatile memory card is most suitable for RAIF. It does not require major architectural changes compared to say, ZFS's whole stripe writes in RAID-Z [45]. Also, the addition of a journal can dramatically decrease RAIF resynchronization times [10].

3 Implementation

Generally, RAIF can be implemented for most OSs that support stacking [53]. We implemented RAIF as a loadable Linux kernel module that can be compiled for both Linux 2.4 and Linux 2.6. RAIF reuses the code of other file systems. As a result, its entire source consists of only 8,375 lines of C code. This is roughly three times smaller than the Linux RAID driver's source code. The RAIF structure is modular so that new RAIF levels, as well as parity and load balancing algorithms, can be added without any significant changes to the main code.

3.1 RAIF6

Our RAIF6 implementation uses the Row-Diagonal Parity Algorithm (RDP) [7] which protects against double disk failures. The RDP algorithm is computationally less intensive than most other ECC algorithms. It stores data in the clear and uses only XOR operations to compute parity. Each data block belongs to one row-parity set and one diagonal-parity set. Because of this, RAIF sometimes has to read multiple stripes when writing data.

3.2 Single vs. Double Buffering

RAIF must meet exacting performance and scalability requirements. We changed the common stackable file system templates accordingly.

Stackable file systems create and maintain their own versions of file system objects. For example, Figure 5 shows Linux kernel data structures representing an open file on RAIF mounted over three lower branches. Among other things, the file structure (`F`) encapsulates per-process current file position (users can set it with the `llseek` system call). RAIF creates its file structure when a file is opened; lower file systems create their own versions of file structures. RAIF's file structure keeps pointers to these lower file structures. Similarly, RAIF and lower file systems have their own directory entry structures (`D`), inode structures (`I`) that among other things keep file size information, address spaces (`A`), and cached data pages (`P`). This way, every ob-

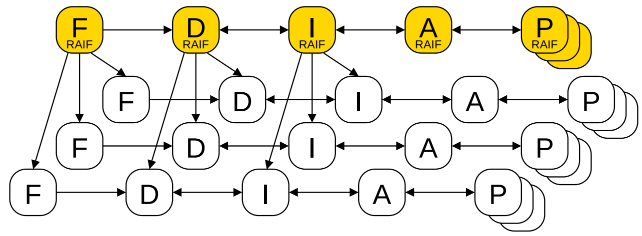

Figure 5: Kernel data structure duplication for every open file. RAIF structures are shown on the top with three lower file systems below. Arrows show important pointers between the structures. `F`, `D`, `I`, `A`, and `P` are the file, directory entry, inode, address space, and the memory page structures respectively.

ject stores the information specific for that file system's object. For example, if a file is striped using RAIF level 0, its file size (stored in the inode structure) is a sum of the sizes stored in the lower branches' inode structures.

Most of these file system structures are small and do not add significant memory overheads. However, the page structures are usually mapped one-to-one with the associated memory pages. Therefore, stackable file systems on Linux duplicate the data. This allows them to keep both modified (e.g., encrypted or compressed) data on lower file systems and unmodified (e.g., unencrypted or uncompressed) data in memory at the same time and thus save considerable amounts of CPU time. However, stackable file systems that do not modify data suffer from double data caching on Linux [22]. For them, double caching does not provide any benefits but makes the page cache size effectively a fraction of its original size. For linear stackable file systems it divides the available cache size in two [25]. For RAIF, the reduction factor depends on the RAIF level and ranges from two for level 0 to the number of lower branches plus one for level 1. In addition, data is copied from one layer to the other, unnecessarily consuming CPU resources.

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 as shown in Figure 5. The stackable file system developer community is considering a general solution to this problem [17, 22, 25, 41]. Meanwhile, we have designed several solutions applicable in different situations.

No upper-level caching. Upon a `read` or a `write` call, RAIF does not call the page-based VFS operations. Instead, it calculates the positions of the data on the lower branches and calls the `read` or `write` methods of the lower file systems directly. These lower methods copy the data between the user-level buffer and the lower data pages. Therefore, the data is never cached at the RAIF level and no data copying between the layers is necessary. Unfortunately, this method still requires in-memory data replication for all the lower branches for RAIF level 1. More importantly, however, this method does not guarantee cache coherency between layers if the file is memory-mapped: in that case, the data is double buffered again. Therefore, we have to disable this RAIF optimization for memory-mapped files.

Temporary low-level caching. For RAIF level 1 and memory-mapped files, RAIF allocates cache pages for its data and copies that data to and from cache pages of the lower branches. However, RAIF attempts to release these lower pages as soon as possible. In particular, it marks lower pages with the `PG_reclaim` flag every time it issues a write request on a lower page. Later, upon completion of a write request, Linux releases such pages automatically. This way, lower pages are only allocated for a short duration of the pending write requests' time. Read requests are satisfied from the RAIF page cache directly or forwarded below for I/O. This deallocation of the lower pages allowed us to reduce the running times of RAIF1 mounted over several lower branches by an order of magnitude under I/O intensive workloads.

3.3 Other Optimizations

Another problem of all Linux fan-out stackable file systems is the sequential execution of VFS requests. This increases the latency of VFS operations that require synchronous accesses to several branches. For example, RAIF deletes files from all branches sequentially. However, it is important to understand that, while this increases the latency of certain file system operations, it has little impact on aggregate RAIF performance under a workload generated by many concurrent processes. Several attempts to perform at least the `unlink` operations asynchronously on Linux Ext2 and Ext3 file systems proved that it is difficult in practice [5].

For RAIF levels that stripe data, RAIF delays file `create` operations. RAIF initially creates files only on the starting and parity branches. If the file eventually grows bigger, then RAIF creates files on the other branches.

During the `readdir` operation, RAIF merges the lists of files from different branches. To optimize this operation, RAIF analyzes the rules in a given directory and derives the minimal subset of branches that contains all the files. For example, consider a RAIF configuration with three branches and two rules: store files matching the pattern `*.mpg` on the first two branches and store all other files on the last two branches. It is sufficient to read the list of files from the middle branch because all the files stored on RAIF are also stored on that lower branch. This optimization allows RAIF to avoid issuing `readdir` requests for some branches. The problem of finding the minimal subset of branches that contains a set of files is NP-complete. Therefore, RAIF uses heuristics to check for simple and common cases first.

3.4 Rule Representation

All rules visible from the user-level have a unified text-based representation of the form:

T:P:R:L:U:S:B, where:

- ***T*** is the rule type, and can be a global rule (`G`), a local rule (`L`), or a meta rule (`M`);
- ***P*** is the rule priority relative to other rules of the same type in the current directory;
- ***R*** is the regular expression (e.g., `*.mpg`);
- ***L*** is the RAIF level;
- ***U*** is the striping unit size (–1 to store files on the starting and parity branches only);
- ***S*** is the starting branch (–1 to use name-hash value);
- ***B*** is the list of branch names to store files on.

For example, if one issues a command to add a rule:

```
raifctl -d /homes \
    add "G:1:*.mpg:0:-1:-1:E0,E1,E2,N"
```

then RAIF will write the same rule string to the rule files on the lower branches. Therefore, user-level tools and custom scripts can read and parse these files when RAIF is not mounted.

4 Evaluation

We have evaluated RAIF's performance using a variety of configurations and workloads. We conducted our benchmarks on two 2.8GHz Intel Xeon (2MB cache, 800MHz bus) machines with 2GB of RAM. The first machine was equipped with four Maxtor Atlas 15,000 RPM 18.4GB Ultra320 SCSI disks formatted with Ext2. For benchmarking hardware RAID, we replaced the SCSI adapter with an Adaptec 2120S SCSI RAID card with an Intel 80302 processor and 64MB of RAM. We used the second machine as an NFS client. Both machines ran Red Hat 9 with a vanilla 2.6.11 Linux kernel and were connected via a dedicated 100Mbps link.

We used the Auto-pilot benchmarking suite [49] to run all of the benchmarks. The lower-level file systems were remounted 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.

For the remainder of the evaluation, RAIFL-B refers to RAIF level L with B branches. Similarly, RAIDL-B refers to the Linux software RAID driver and HWRAIDL-B refers to hardware RAID. We use EXT2 to refer to a configuration with an unmodified Ext2.

4.1 Random-Read

RANDOM-READ is a custom micro-benchmark designed to evaluate RAIF under a heavy load of random data read operations. It spawns 32 child processes that concurrently read 32,000 randomly located 4,096-byte blocks in a set of files. We ran this benchmark with RAIF0, RAIF1, and RAIF5 using a stripe size of 64KB with up to four Ext2 branches. We did not include results for RAIF4, because RAIF0 with N branches is the same as RAIF4 with $N + 1$ branches. In addition, we did not include results for RAIF6 because they are similar to RAIF0 with dedicated ECC branches, or to RAIF5 with rotating ECC branches. We compared RAIF to RAID with similar configurations. Before running the benchmark, we created eight 2GB files on each of the four disks. We wrote the files in a round-robin fashion for each set of eight files, so that each file spanned almost the entire physical disk. Because the files were created in this way, the benchmark always spans almost the full size of each disk, regardless of the number of branches we used.

Figure 6 shows a comparison of RAID and RAIF under this workload, as well as reference results for Ext2. User and system times were small and statistically indistinguishable for all configurations. Additionally, the elapsed times for all RAID0 and RAIF0 configurations were statistically indistinguishable. This is because they use the same number of disks, and RAIF passes the requests directly to the underlying Ext2 branches. As we added more disks to RAIF0, the I/O requests were distributed among more disks, and the I/O time dropped significantly. RAIF1 performed slightly better than RAID1, except for RAIF1-3, which had a 6.4% overhead. The performance of RAIF5 was slightly worse than RAID5, with 4.6% overhead for three branches, and 1.7% for four branches. Overall, we can conclude that RAIF has comparable performance to driver-level software RAID for read workloads.

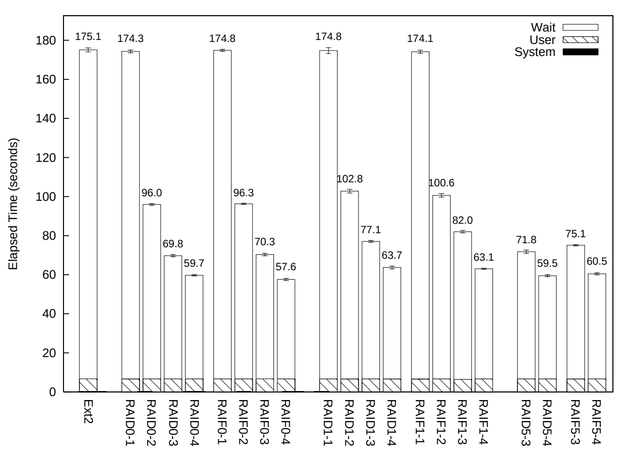

Figure 6: RANDOM-READ benchmark results using RAIF0, RAIF1, and RAIF5 with varying numbers of branches. Ext2 results are shown as a reference.

4.2 Postmark v1.5

Postmark [28] simulates the operation of electronic mail servers. It performs a series of file appends, reads, creations, and deletions, showing how RAIF might behave in an I/O-intensive environment. We chose a Postmark configuration to stress I/O: it creates 60,000 files of size between 512–10K bytes in 600 directories, and performs 600,000 transactions. All operations were selected with equal probability. Every other Postmark's operation is either a CREATE or an UNLINK. Due to VFS restrictions, these RAIF operations are executed sequentially on lower branches and are CPU-intensive. This makes Postmark a challenging benchmark for RAIF.

We ran Postmark for all applicable numbers of branches under RAIF levels 0, 1, 4, 5, and 6. We compared the results to both driver-level and hardware RAID implementations for their respective available RAID levels. Figure 7 shows these results, as well as results for Ext2 for reference.

RAIF0. As we can see from Figure 7, RAIF0-2 was 18.6% slower than RAID0-2, due to a 59.1% increase in system time and a 32.4% increase in wait time. The performance degraded slightly when more branches were added (overheads of 25.6% and 20.7% for three and four branches, respectively). This was due to the increased system time associated with extra branches.

RAIF1. The results for RAIF1 were similar to those for RAIF0, with similar increases to system time overhead as more branches were added. In this case, the elapsed time overheads were 26.4%, 30.5%, and 29.3% for configurations with 2, 3, and 4 branches, respectively. For RAIF1, we release cache pages of lower file systems after all write operations. This allowed us to decrease RAIF1-3 and RAIF1-4 overheads by approximately ten times.

RAIF4 and RAIF5. Whereas the system time of RAIF4 was higher than RAID4, the wait time was significantly lower, resulting in overall better performance. The system time of RAIF4-3 was 2.2 times that of RAID4-3, but it had 67.6% less wait time, resulting in a 46.1% improvement. Similarly, the system time of RAIF4-4 was 2.3 times that of RAID4-4, but the wait time was reduced by 75.8%, yielding an overall improvement of 54.0%. We saw similar results for RAIF5 and RAID5. For RAIF5-3, system time was 2.1 times that of RAID5-3, wait time was 63.8% lower, and there was an overall improvement of 44.5%. The system time of RAIF5-4 was 2.4 times that of RAID5-4, the wait time improved by 74.3%, and the elapsed time improved by 53.0%.

To understand this behavior, we profiled EXT2 mounted over RAID using OSprof [24]. We noticed that the EXT2 WRITEPAGE operation had disproportionally high latency due to disk head seeks. In particular, the Linux RAID driver experienced 6,569 long disk head seeks while writing dirty buffers to the disk. RAIF, on the other hand, waited for

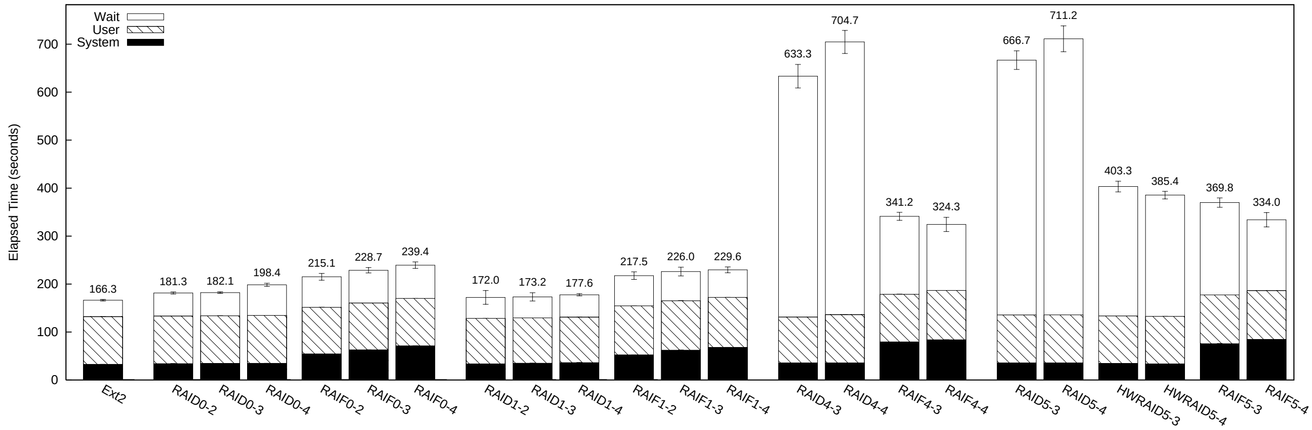

Figure 7: Postmark results for RAIF and RAID with varying levels and numbers of branches. Also shown are the results for an Adaptec 2120S RAID5 card. Results for Ext2 are shown as a reference.

only 211 long disk head seek operations. This is because RAIF operates above file system caches and parity pages are cached as normal data pages. Linux RAID, however, operates with buffers logically below file system caches. Therefore, the Linux RAID driver has to read existing parity pages for most write operations, whereas RAIF has them cached.

We also benchmarked one of the hardware implementations of RAID5, and RAIF5 was faster than that implementation as well. Hardware and driver-level implementations had similar system time overheads. RAIF5-3 had 28.6% less wait time than HWRAID5-3, and was 8.3% faster overall. RAIF5-4 had 41.5% wait time improvement over HWRAID5-3, and the elapsed time improved by 13.3%. This is because even though the RAID controller card has 64MB of RAM available for caching parity, RAIF has access to main memory.

RAIF6. RAIF6-4 (not shown in Figure 7 because of the large value difference compared to the other bars) performed 70% slower than RAID6-4, with a total elapsed time of 1,183.8 seconds. This shows that our current RAIF6 implementation requires further optimization for write-oriented workloads (part of our future work).

In summary, RAIF, like RAID, improves performance of read-oriented workloads. For writes, RAIF5 performs better than a driver-level software RAID5 system and even better than a hardware RAID5 system, both of which support only fixed configurations. Due to improved caching policies, RAIF caches parity information and can avoid many of the parity-read requests necessary to update existing parity pages. Furthermore, CPU speeds are increasing at a much faster pace than network and disk I/O. Therefore, these results will become even more significant in the future. Also, we used one of the fastest hard drives available today. Results with commodity SCSI or even ATA hard disks would favor RAIF more.

5 Related Work

Block-level RAID. Striping and replication of data possibly combined with the use of error-correction codes on homogeneous [34] and heterogeneous [8] configurations of local hard drives has been commonly used for decades to improve data survivability and performance. Modern block-level virtualization systems [21] rely on Storage Area Network (SAN) protocols such as iSCSI to support remote block devices. The idea of using different RAID levels for different data access patterns at the block level was used in several projects at the driver [14] and hardware [47] levels. However, the lack of higher-level information forced the developers to make decisions based either on statistical information or semantics of particular file systems [42]. Exposed RAID (E×RAID [9]) reveals information about parallelism and failure isolation boundaries, performance, and failure characteristics to the file systems. Informed Log-structured file system (I.LFS) uses E×RAID for dynamic load balancing, user control of file replication, and delayed replication of files. RAIF already operates at the file system level and possesses all the meta information it needs to make intelligent storage optimization decisions [9]. Solaris's ZFS is both a driver and a file system [45]. Despite having all the necessary information, it supports storage policies on a storage-pool basis only. This means that whole devices and whole file systems use the same storage policies. RAIF provides more versatile virtualization on a per-file basis.

File-system–level RAID. Higher level storage virtualization systems operate on files [11]. Their clients work as file systems that send networked requests to the servers [1, 15]. Clients find files on servers using dedicated meta-servers or hash functions [20]. Ceph and Zebra are distributed file systems that use per-file RAID levels and striping with parity [15, 46]. They have dedicated meta servers to locate the data. Ursa Minor's networking protocol supports special properties for storage fault handling [1]. Coda's client file system is a wrapper over a pseudo-device that directly communicates with a user-mode caching server [30]. The server replicates data on other similar servers.

File-system–level storage virtualization systems can support per-file storage policies. However, they still have fixed and inflexible architectures. In contrast, RAIF's stacking architecture allows it to utilize the functionality of existing file systems transparently and to support a variety of configurations without any modifications to file systems' source code. Thus, RAIF can use any state-of-the-art and even future file systems. Also, any changes to the built-in protocols of any fixed storage virtualization system will require significant time and effort and may break compatibility between the storage nodes.

Storage utilization. Storage resizing is handled differently by different storage-virtualization systems. For example, ZFS starts by lazily writing new data to newly added disks (which are initially free) and old disks [45]. Similarly, specially designed hash functions can indirectly cause more data to be written to the newly added disk [20]. This approach works only if all old data eventually gets overwritten, which is not the case in many long-term storage systems. An alternative approach is to migrate the data in the background [6, 12]. RAIF supports both lazy and background data migration.

Load balancing. Media-distribution servers use data striping and replication to distribute the load among servers [6]. The stripe unit size and the degree of striping have been shown to influence the performance of these servers [40]. Replication in RAIF uses proportional-share load balancing using the expected delay as the load metric. This approach is generally advocated for heterogeneous systems [39].

RAID survivability. Remote mirroring is a popular storage disaster recovery solution [29]. In addition to data replication, distributed RAID systems [44] use m/n redundancy schemes [37] to minimize the extra storage needs. RAIF can use a number of remote branches to store remote data copies or parity.

Two popular solutions to the silent data corruption problem [4] are journalling [10] (e.g., using non-volatile memory) and log-structured writes [9, 45]. RAIF can support journalling with no major design changes.

Stackable file systems. RAIF is implemented as a stackable file system [52]. Stackable file systems are not new [53]. Originally introduced in the early '90s [38], stackable file systems were proposed 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. Today, existing stackable file systems provide functionalities like encryption [50], versioning [33], tracing [3], data corruption detection [27], secure deletion [26], and more. Similarly to stacking, it is also possible to add new functionality to other file systems directly [26].

A class of stackable file systems known as *fan-out* file systems mount on top of more than one file system to provide useful functionality [16, 38]. However, so far the most common application of fan-out has been unification [18, 31, 36, 48]. Unification file systems present users with a merged view of files stored on several lower file systems. RAIF is a stackable fan-out file system that can mount on top of several underlying file systems (branches) to provide RAID-like functionality.

6 Conclusions

This paper has three primary contributions: (1) we present a storage virtualization file system we call RAIF, which supports many possible compositions of existing and future file systems; (2) RAIF supports per-file storage policies; and (3) it offers new possibilities for users and administrators.

1. RAIF is designed at a higher level of abstraction than other file systems and, in fact, reuses other file systems without modification. This allows RAIF to remain simple and maintainable while having the flexibility of many existing file systems combined. RAIF has general support for all NAS and local storage types without any backend-specific code. Therefore, RAIF not only supports existing file systems but also support future file systems and types of storage. In addition, other stackable file systems can be inserted above RAIF, or below it (but mounted over several lower branches) to provide extra features (e.g., encryption, compression, and versioning).

   RAIF currently supports all standard RAID levels and their advanced versions. We designed RAIF to be extensible with easy-to-modify parameters and with modular support for new RAIF levels that determine the RAIF personality. This makes it easy to add data placement policies.
2. Usually, files in a data store can be subdivided into groups with similar purposes, importance, and access patterns. RAIF supports the notion of rules that allow storage configuration for every such group of files. A group can be as small as a single file. These rules can also be applied for a single directory or a branch of a file system tree. Additionally, meta-rules allow the use of RAIF over read-only file systems and the implementation of rudimentary name-space unification.
3. RAIF eases storage administration significantly because many administration tasks such as security policy assignment, data backup and restoration, storage device replacement, and file system crash recovery can be performed for individual lower branches. Simple

user-level scripts allow administrators to change data storage policies, modify storage configurations, and automatically migrate data.

RAIF's flexibility does not increase administration complexity. RAIF allows its administrators to choose appropriate rules and lower file systems. However, we anticipate that in most configurations RAIF will have just a few rules and will be mounted over two or three types of different file systems.

Performance. RAIF has modest system-time overheads. Similar to RAID, it improves performance for multi-threaded workloads. RAIF has better control over the caches and has access to the high-level meta-information. In spite of RAIF's extra functionality, RAIF5 performs better than a state-of-the-art software block-level RAID5 and even a hardware RAID5 for some write-oriented workloads. In the future, RAIF's relative performance is expected to improve even more.

Cost-Efficiency. High-end RAID controllers would most likely outperform RAIF because of their extra caches and CPUs. However, the extra hardware cost spent on the controller can buy even more commodity memory and CPU power for the system running RAIF. In that case, the extra resources will be available for other system activity when not needed by RAIF.

Future Work. We are working on RAIF journal support to increase the reliability and decrease the array resynchronization time [10].

We are investigating how to improve RAIF6's performance, possibly by adding other error-correction algorithms—as well as supporting RAIF6 with more than two degrees of redundancy.

We plan to augment RAIF to provide quality-of-service guarantees; we plan to utilize the fact that RAIF operates at the file system level and has access to all meta-information [21].

We plan to port RAIF to other OSs such as FreeBSD, Solaris, and Windows. This is easier than it might appear because stackable file systems on many OSs operate in a similar way [53].

RAIF is released under the GPL at:
`ftp://ftp.fsl.cs.sunysb.edu/pub/raif/`

Acknowledgments

We would like to acknowledge contributions of Adam David Alan Martin for help with RAIF level 6, Jeff Sipek for discussions about the Linux RAID driver, and Charles P. Wright for his help with the experiments.

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] M. Abd-El-Malek, W. V. Courtright II, C. Cranor, G. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. Ursa Minor: Versatile Cluster-based Storage. In *Proc. of the Fourth USENIX Conf. on File and Storage Technologies*, pp. 59–72, San Francisco, CA, December 2005.

[2] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, June 1975.

[3] A. Aranya, C. P. Wright, and E. Zadok. Tracefs: A File System to Trace Them All. In *Proc. of the Third USENIX Conf. on File and Storage Technologies*, pp. 129–143, San Francisco, CA, March/April 2004.

[4] N. Brown. Re: raid5 write performance, November 2005. `http://www.mail-archive.com/linux-raid@vger.kernel.org/msg02886.html`.

[5] M. Cao, T. Y. Tso, B. Pulavarty, S. Bhattacharya, A. Dilger, and A. Tomas. State of the art: Where we are with the ext3 filesystem. In *Proc. of the Linux Symposium*, Ottawa, ON, Canada, July 2005.

[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 Conf. on Distributed Computing Systems*, pp. 64–71, Taipei, Taiwan, April 2000.

[7] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. Row-Diagonal Parity for Double Disk Failure Correction. In *Proc. of the Third USENIX Conf. on File and Storage Technologies*, pp. 1–14, San Francisco, CA, March/April 2004.

[8] T. Cortes and J. Labarta. Extending Heterogeneity to RAID level 5. In *Proc. of the Annual USENIX Technical Conf.*, pp. 119–132, Boston, MA, June 2001.

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

[10] T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Journal-guided Resynchronization for Software RAID. In *Proc. of the Fourth USENIX Conf. on File and Storage Technologies*, pp. 87–100, San Francisco, CA, December 2005.

[11] G. A. Gibson, D. F. Nagle, W. Courtright II, N. Lanza, P. Mazaitis, M. Unangst, and J. Zelenka. NASD Scalable Storage Systems. In *Proc. of the 1999 USENIX Extreme Linux Workshop*, Monterey, CA, June 1999.

[12] T. Gibson. *Long-term Unix File System Activity and the Efficacy of Automatic File Migration*. PhD thesis, Department of Computer Science, University of Maryland Baltimore County, May 1998.

[13] B. S. Gill and D. S. Modha. WOW:Wise Ordering for Writes—Combining Spatial and Temporal Locality in Non-Volatile Caches. In *Proc. of the Fourth USENIX Conf. on File and Storage Technologies*, pp. 129–142, San Francisco, CA, December 2005.

[14] 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 *Proc. of the FREENIX Track at the 2000 USENIX Annual Technical Conf.*, pp. 21–34, San Diego, CA, June 2000.

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

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

[17] J. S. Heidemann and G. J. Popek. Performance of cache coherence in stackable filing. In *Proc. of the Fifteenth ACM Symposium on Operating Systems Principles*, pp. 3–6, Copper Mountain Resort, CO, December 1995.

[18] D. Hendricks. A Filesystem For Software Development. In *Proc. of the USENIX Summer Conf.*, pp. 333–340, Anaheim, CA, June 1990.

[19] V. Henson, A. Ven, A. Gud, and Z. Brown. Chunkfs: Using Divide-and-Conquer to Improve File System Reliability and Repair. In *Proc. of the Second Workshop on Hot Topics in System Dependability (HotDep 2006)*, Seattle, WA, November 2006.

[20] R. J. Honicky and E. Miller. Replication Under Scalable Hashing: A Family of Algorithms for Scalable Decentralized Data Distribution. In *Proc. of the 18th International Parallel and Distributed Processing Symposium*, April 2004.

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

[22] N. Joukov. Re: [RFC] Support for stackable file systems on top of nfs, November 2005. `http://marc.theaimsgroup.com/?l=linux-fsdevel&m=113193082115222`.

[23] N. Joukov, A. Rai, and E. Zadok. Increasing distributed storage survivability with a stackable raid-like file system. In *Proc. 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)*, pp. 82–89, Cardiff, UK, May 2005

[24] N. Joukov, A. Traeger, R. Iyer, C. P. Wright, and E. Zadok. Operating system profiling via latency analysis. In *Proc. of the 7th Symposium on Operating Systems Design and Implementation*, pp. 89–102, Seattle, WA, November 2006. ACM SIGOPS.

[25] N. Joukov, T. Wong, and E. Zadok. Accurate and efficient replaying of file system traces. In *Proc. of the Fourth USENIX Conf. on File and Storage Technologies*, pp. 337–350, San Francisco, CA, December 2005.

[26] N. Joukov and E. Zadok. Adding Secure Deletion to Your Favorite File System. In *Proc. of the third international IEEE Security In Storage Workshop*, pp. 63–70, San Francisco, CA, December 2005.

[27] A. Kashyap, S. Patil, G. Sivathanu, and E. Zadok. I3FS: An In-Kernel Integrity Checker and Intrusion Detection File System. In *Proc. of the 18th USENIX Large Installation System Administration Conf.*, pp. 69–79, Atlanta, GA, November 2004.

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

[29] K. Keeton, C. Santos, D. Beyer, J. Chase, and J. Wilkes. Designing for disasters. In *Proc. of the Third USENIX Conf. on File and Storage Technologies*, pp. 59–72, San Francisco, CA, March/April 2004.

[30] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. In *Proc. of 13th ACM Symposium on Operating Systems Principles*, pp. 213–225, Asilomar Conf. Center, Pacific Grove, CA, October 1991.

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

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

[33] K. Muniswamy-Reddy, C. P. Wright, A. Himmer, and E. Zadok. A Versatile and User-Oriented Versioning File System. In *Proc. of the Third USENIX Conf. on File and Storage Technologies*, pp. 115–128, San Francisco, CA, March/April 2004.

[34] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proc. of the ACM SIGMOD*, pp. 109–116, June 1988.

[35] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS version 3 design and implementation. In *Proc. of the Summer USENIX Technical Conf.*, pp. 137–152, Boston, MA, June 1994.

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

[37] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: The OceanStore Prototype. In *Proc. of the Second USENIX Conf. on File and Storage Technologies*, pp. 1–14, San Francisco, CA, March 2003.

[38] D. S. H. Rosenthal. Evolving the Vnode interface. In *Proc. of the Summer USENIX Technical Conf.*, pp. 107–118, Anaheim, CA, June 1990.

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

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

[41] J. Sipek, Y. Pericleous, and E. Zadok. Kernel Support for Stackable File Systems. In *Proc. of the 2007 Ottawa Linux Symposium*, volume 2, pp. 223–227, Ottawa, Canada, June 2007.

[42] M. Sivathanu, V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving Storage System Availability with D-GRAID. In *Proc. of the Third USENIX Conf. on File and Storage Technologies*, pp. 15–30, San Francisco, CA, March/April 2004.

[43] L. Stein. Stupid File Systems Are Better. In *Proc. of the 10th Workshop on Hot Topics in Operating Systems (HotOS X)*, Santa Fe, NM, June 2005.

[44] M. Stonebreaker and G. A. Schloss. Distributed raid—a new multiple copy algorithm. In *Proc. of the 6th International Conf. on Data Engineering (ICDE'90)*, pp. 430–437, February 1990.

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

[46] S. Weil, S. Brandt, E. Miller, D. Long, and C. Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. In *Proc. of the 7th Symposium on Operating Systems Design and Implementation*, pp. 307–320, Seattle, WA, November 2006. ACM SIGOPS.

[47] 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.

[48] C. P. Wright, J. Dave, P. Gupta, H. Krishnan, D. P. Quigley, E. Zadok, and M. N. Zubair. Versatility and unix semantics in namespace unification. *ACM Transactions on Storage (TOS)*, 2(1):1–32, February 2006.

[49] C. P. Wright, N. Joukov, D. Kulkarni, Y. Miretskiy, and E. Zadok. Auto-pilot: A platform for system software benchmarking. In *Proc. of the Annual USENIX Technical Conf., FREENIX Track*, pp. 175–187, Anaheim, CA, April 2005.

[50] C. P. Wright, M. Martino, and E. Zadok. NCryptfs: A secure and convenient cryptographic file system. In *Proc. of the Annual USENIX Technical Conf.*, pp. 197–210, San Antonio, TX, June 2003.

[51] E. Zadok, J. M. Anderson, I. Bădulescu, and J. Nieh. Fast Indexing: Support for size-changing algorithms in stackable file systems. In *Proc. of the Annual USENIX Technical Conf.*, pp. 289–304, Boston, MA, June 2001.

[52] E. Zadok and I. Bădulescu. A stackable file system interface for Linux. In *LinuxExpo Conf. Proc.*, pp. 141–151, Raleigh, NC, May 1999.

[53] 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.

[54] E. Zadok and J. Nieh. FiST: A Language for Stackable File Systems. In *Proc. of the Annual USENIX Technical Conf.*, pp. 55–70, San Diego, CA, June 2000.

[55] E. Zadok, J. Osborn, A. Shater, C. P. Wright, K. Muniswamy-Reddy, and J. Nieh. Reducing Storage Management Costs via Informed User-Based Policies. In *Proc. of the 12th NASA Goddard, 21st IEEE Conf. on Mass Storage Systems and Technologies*, pp. 193–197, College Park, MD, April 2004