The *Legend* File System: Better Reliability through Implicit Replicas

Robert Roy, Britten Dennis, An-I Andy Wang
Computer Science Department
Florida State University

Peter Reiher
Computer Science Department
University of California, Los Angeles

Abstract

We have designed, prototyped, and evaluated the *Legend* file system, which exploits the ability to regenerate lost files to improve reliability. Unlike RAID-5 and RAID-6, *Legend* degrades gracefully as the number of failed storage devices increases. *Legend* also mitigates the high storage capacity overhead imposed by n-way replications. Combined with existing methods, *Legend* can form another line of defense against data loss.

1. Introduction

The quest for ever higher storage density and capacity has placed a growing strain on current reliability mechanisms. Greater storage demand leads to more devices in a system, increasing the chance of the system encountering device failures [PAT98].

Common solutions rely on some form of data redundancy to mask failures. RAID-5 and RAID-6 exploit partial data redundancy and can survive failures if the number of failed storage devices is below a certain threshold (e.g., two for RAID-6), assuming independent device failures. However, real-world device failures are not independent [SCH07; MIL09]. Higher storage capacity also increases the chance of a device hosting corrupted bits [HAR07; GRU12], undetected until the recovery time. Thus, enterprise deployments combine the use of RAIDs with 2- or 3-way full data replications [SHV10; PAR13]. However, full replication imposes high storage overhead and can be cost-prohibitive for small- to mid-scale deployments.

We introduce the *Legend* file system, which exploits file regeneration to guard against data loss. The key observation is that a file set used to generate a file can serve as an *implicit replica* of the regenerated file. Through our design, implementation, and evaluation, *Legend* shows that it can degrade gracefully as the number of failed storage devices increases, without imposing high storage overheads.

2. Observations

The following observations led to the design of *Legend*.

2.1 Missed Opportunities in File Relationships

Common reliability schemes do not exploit the relationships among files, overlooking potentially embedded data redundancy. For example, suppose a file `X.jpg` is derived from `X.bmp`. Replication of both files under traditional schemes would provide two copies of `X.bmp` and effectively four copies of `X.jpg` (since `X.bmp` contains redundant information of `X.jpg`). Alternatively, the system can replicate only `X.bmp` twice to provide three copies of `X.bmp` and effectively four copies of `X.jpg`, reducing the efforts while increasing the overall replication factor.

2.2 File Regeneration as a Form of Data Redundancy

With the running example, suppose `X.jpg` is generated via running `bmp2jpg X.bmp`. All files associated with `bmp2jpg` (e.g., library files) and `X.bmp` can form a file set `Y`, which is effectively an *implicit replica* of `X.jpg`, since the loss of `X.jpg` can be regenerated via `Y`. These types of opportunities can be found in workflow-related workloads (e.g., compilation, simulation, and general multi-staged data processing).

2.3 Practical to Recall Past System States

Recent systems, such as Arnold [DEV14], have shown the feasibility of recalling all system states, including all file versions, by logging all input dependency information (e.g., input files, external input such as X Windows events, network input, IPCs, etc.). Arnold could achieve this by logging at a rate of 2.8GB/day, with a performance penalty under 8%. Thus, the implicit-replica approach can potentially be applied system-wide.

2.4 Many-core Trends

While trading computation with storage capacity for better reliability may seem expensive, the continuation of the Moore's Law in the number of CPU cores [BUR14], along with the availability of massive parallelism, may make this tradeoff favorable [TIM10].

3. The *Legend* File System

The *Legend* File System exploits the notion of implicit replicas to improve reliability. Unlike Arnold's focus on complete lineage history tracking and queries, *Legend* focuses on reliability. To reduce tracing overhead, *Legend* currently targets files that can be regenerated without dependencies on external input and IPCs. To reduce consistency-related overhead, *Legend* is designed as a speculative reliability layer. That is, if regeneration is not successful or possible, *Legend* falls back to the next layer of reliability measure (e.g., backup). This semantics is still useful, since once RAID-6 encounters two disk failures, it has to fall back to the reliability layer. On the other hand, *Legend* is designed to fail gracefully and continue service proportional to the number of surviving devices, and it can be used as another line of defense.

The following subsections discuss our major design components. We will use software compilation as a running example, due to readers' familiarity with this type of computation and its richness in corner cases.

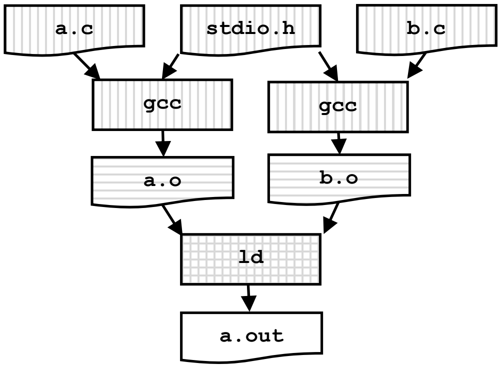


Figure 3.1.1: A simplified file-creation dependency graph for a compilation example. Rectangles are executables, and the document boxes are input and output files. The boxes containing vertical bars are one implicit replica of `a.out`. The boxes containing horizontal bars are the second implicit replica of `a.out`. `ld` is a member of both implicit replicas and contains both vertical and horizontal bars.

3.1. Identifying Implicit Replicas

File-creation dependency graph: We identify implicit replicas by gathering traces on process- and file-related system calls. Files referenced under the same process group ID form a file-creation dependency graph, where nodes are files or processes associated with executables. The reading of an input file *I* by an executable's process *E* forms an inbound edge *I*→*E*. The writing of an output file *O* by an executable's process *E* forms an outbound edge *E*→*O*. If a `mmap` call has its writable flag set, it is treated as a write; otherwise, it is treated as a read. An input file can also be the output file. For example, an editor reads in a file, modifies it, and overwrites it with the new version. Similarly, a script file can be an input, output, and executable file. However, our system considers files with only inbound edges as regenerable. Thus, a dynamically generated script file is considered regenerable. When the script is executed, a separate process node associated with the script is created. The creation time of the dependency graph is linear to the number of trace entries.

Graph trimming: A file-creation dependency graph is typically large. We trim this graph by grouping files from the same library package into a single node. Files under `/tmp` are also omitted, as discussed later.

Implicit replicas: Figure 3.1.1 shows a simplified file-creation dependency graph for a compilation example. From the bottom up, `a.out` can be generated with the presence of the `ld` executable (represented in a rectangle box), taking in the input files of `a.o` and `b.o` (represented in document boxes). Thus, `ld`, `a.o`, and `b.o` (boxes containing horizontal lines) are an ***implicit replica*** of `a.out`, resulting in the following regeneration rule:

```
ld, a.o, b.o → a.out                    (3.1.1)
```

Similarly, `a.o` and `b.o` can be regenerated with the following rules:

```
gcc, stdio.h, a.c → a.o                 (3.1.2)
gcc, stdio.h, b.c → b.o                 (3.1.3)
```

By expanding `a.o` and `b.o` in (3.1.1) with the left-hand side of (3.1.2) and (3.1.3), and with duplication removed, we can derive the following rule:

```
gcc, ld, stdio.h, a.c, b.c → a.out (3.1.4)
```

Now, `gcc`, `ld`, `stdio.h`, `a.c`, and `b.c` (represented with boxes containing vertical bars) can serve as a second implicit replica of `a.out`.

Duplicate file memberships for implicit replicas: Since memberships are determined by rules, a file can be replicated to be the member of multiple implicit replicas (unless the maximum replication factor is capped). In this case, the `ld` box contains both vertical and horizontal bars, denoting that it belongs to the first and second implicit replicas of `a.out`.

Graph consistency: Since our reliability layer is speculative, a file generated based on out-of-date dependencies or upper stream input file updates may have a mismatching checksum. Once a mismatch occurs, we fall back to the next layer of reliability mechanism. One implication is that we do not need to maintain the consistency of our graphs as aggressively, which can incur high overhead. While traces need to be gathered continuously, daily updates of the dependency graph would be sufficient.

Additionally, we used file names instead of their content hashes to build the dependency graph. Thus, as long as updates lead to the same checksum, our recovery is considered successful.

Nondeterminism: Files under `/tmp` often have randomly generated file names from different processes to avoid name collisions. Suppose `a.o` and `b.o` are located under `/tmp`, our system omits them. The net effect is the same as replacing the file regeneration rule (3.1.1) with (3.1.4).

The same technique can be applied to other intermediary regenerated files that contain nondeterministic content (e.g., embedded timestamps).

Currently, we handle regenerated files with nondeterministic content located at leaf nodes of the dependency graph. A mismatched checksum simply means falling back to the next reliability mechanism.

3.2. Implications of Forming Implicit Replicas

The use of implicit replicas has several implications. First, to avoid correlated failures, implicit replicas should be placed on separate storage devices.

Second, spreading files within an implicit replica across multiple storage units increases the chance of losing a file within the implicit replica due to a device failure. On the other hand, it may increase the CPU parallelism to regenerate multiple implicit replicas that are stored across devices.

Third, regenerable files deeper in the dependency graph have more opportunities to form implicit replicas. Spreading these files rather than the files near the top of the graph across devices will improve parallelism.

Fourth, while implicit replicas can be formed with files from different depths of the dependency graph, as described in (3.1.5) and (3.1.6), spreading out files with fewer implicit replicas increases the chance of losing one of them, which would require us to fall back on an underlying reliability layer to recover.

```
gcc, ld, stdio.h, a.c, b.o → a.out (3.1.5)
gcc, ld, stdio.h, a.o, b.c → a.out (3.1.6)
```

Device	Content
0	gcc, ld, stdio.h, a.c, b.c
1	ld, a.o, b.o
2	a.out

Figure 3.3.1: An example of implicit replica assignment.

3.3. Device Assignment

Assigning implicit replicas to storage devices resembles the n-coloring problem, where each implicit replica within the same regeneration tree has a unique color (or storage device). Otherwise, two implicit replicas residing on the same device can fail together.

Since n-coloring is NP-complete, we used a greedy scheme. Each implicit replica is assigned a depth based on the file-creation dependency graph. Each replica then is assigned to storage devices in a round-robin fashion. If a file belongs to multiple implicit replicas (e.g., a user-created script), we duplicate it to avoid correlated implicit replica failures.

If there are more storage devices than implicit replicas, we can spread out the files of some implicit replicas across multiple devices to improve parallelism and recovery speed, considering several constraints. First, we should not spread out the files of an implicit replica that cannot be regenerated (e.g., .c files), since doing so increases its chance of failure. Second, for regenerable replicas, we used a peak-to-mean-ratio threshold of the maximum implicit replica size to the average replica size to determine whether we should balance the storage capacity. Third, we used a peak-to-mean ratio threshold of the maximum implicit replica access frequency to the average replica access frequency to determine whether to balance the loads.

3.4. Name Space Management

Having assigned a file to a device, the path leading to the file is created or replicated. Thus, having implicit replicas grouped by depths encourages spatial grouping of files and limits the extents of replications

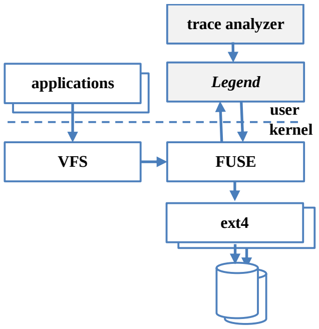


Figure 4.1: *Legend* components (shaded) and the data path from applications to the underlying ext4.

4. Implementation

The two major components of *Legend* are the trace analyzer and the file system itself. We prototyped *Legend* in the user space via FUSE (v2.9.3) [Szeredi 2005] (Figure 4.1) running atop Linux 3.13.11.

The trace analyzer gathers system-call traces related to process executions and file system accesses from strace. We captured all parameters (e.g., environmental variables) for file regeneration. The analyzer then generates file-creation dependency graphs, identifies implicit replicas, and maps files to storage devices, per-file checksums, and corresponding regeneration methods, to be used by *Legend*.

To enable file allocation to individual storage devices without re-engineering the data layout, we modified the Unionfs (v2.6) [2016], which can combine multiple file systems into a single name space. To illustrate, if /dir/file1 resides on device 0 and /dir/file2 resides on device 1, /dir is replicated on both devices. In our case, we ran ext4 (v1.42.9) on each storage device.

We modified Unionfs to use our file-to-device mapping to determine and speed up lookups. In addition, in the case of failures, *Legend* would back trace the file-creation dependency graph and trigger regeneration. In our running example, should the device containing a.out fail, *Legend* would try to regenerate based on the first implicit replica, or `ld`, `a.o`, and `b.o`. If the first implicit replica fails to regenerate, *Legend* would try regenerate based on the second implicit replica, or `gcc`, `ld`, `stdio.h`, `a.c`, and `b.c`.

Since the file-creation dependency graph does not capture timing dependencies, we had to make conservative timing estimate when reissuing individual execution system calls. Thus, our system tries to reissue top-level user commands when applicable during recovery to better exploit concurrency.

The trace analyzer is written in Perl (863 semicolons); the *Legend* file system is written in C (2,447 semicolons).

5. Evaluation

We evaluated the *Legend* file system's reliability using trace replays on a simulation, and measured recovery cost and performance overhead based on the actual prototype. Each experiment was repeated 5 times, and results are presented at 90% confidence intervals.

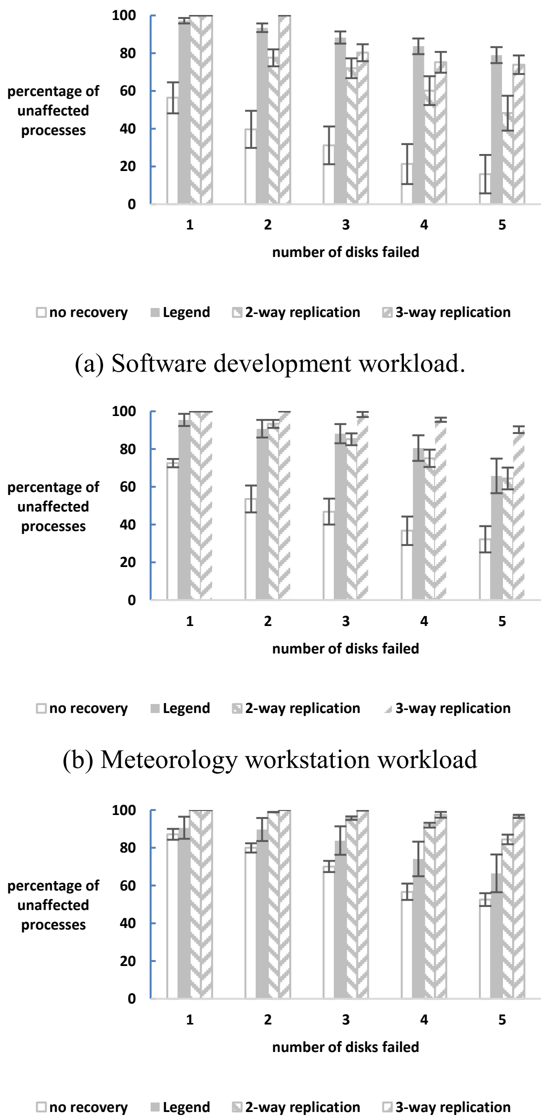
5.1. Reliability

A 15-disk simulator was used to explore interactions between n-disk failures with reliability strategies, including no recovery (RAID-0) and *Legend*, as well as 2- and 3-way replication.

The simulator was populated with contents based on three traces. The first 355MB trace was gathered from a software development environment (10/6/2015-10/13/2015). The trace contains 400K references to 29 million unique files from 1,440 execution calls. The second 2.2GB trace was gathered from a meteorology workstation running radar plotting, image generation, and statistical analysis (8/14/2015-8/24/2015). The trace contains 900K references to 63 million unique files from 47K execution calls. The third 43GB trace was gathered from a meteorology student server running statistical analysis workloads (8/14/2015-8/24/2015). The trace contains 2.6K references to 192K unique files from 4.4K execution calls.

Given that *Legend* regenerates files based on past reference patterns, a longer deployment will yield a greater coverage of files. For the simulation, we assume oracle knowledge to explore the full potential of our approach for long-term deployment. That is, all references are processed first, prior to the replay of the traces. We varied the number of randomly chosen failed storage devices at the beginning of the replay and compared the percentage of processes that can complete successfully [SIV04a].

Figure 5.1 compares the reliability of different schemes as the number of failed storage devices increases. The reliability provided by various schemes varies depending on the workloads. The software development trace contains the most generative dependencies among file groups (13K edges), followed by the meteorology workstation trace (9.5K edges). The student server trace only contains 569 edges. *Legend* performs better with more generative dependencies (Figure 5.1a). With 3 disk failures, *Legend* alone can outperform the no recovery option by 50%, and *Legend* even outperforms 3-way replication.



(a) Software development workload.

(b) Meteorology workstation workload

(c) Meteorology student server workload.

Figure 5.1: Percentage of unaffected processes vs. number of failed disks.

On the other hand, other schemes thrive with fewer generative dependencies. This inverse correlation suggests that *Legend* can potentially be combined with other approaches to achieve a greater overall reliability. For instance, *Legend* can be combined with 2-way replication. As *Legend* identifies files with implicit replicas, the space and effort used for their explicit replicas can be redirected for other files to increase the effective replication factor.

5.2. Recovery

We compared the performance of *Legend* stacked atop of 4-disk, per-disk ext4 via FUSE with that of the baseline ext4-based RAIDs (no FUSE). Table 5.2.1 shows our experimental platform.

Given that RAID recoveries are typically performed in the background and throttled while serving foreground requests, we measured the bandwidth ranges that can be achieved through regeneration. For the low bandwidth bound, we measured the regeneration of missing `.o` files of Dungeon Crawl (v0.13.0) [2016], and for the high bandwidth bound, we measured the regeneration of one of our trace files from a `.gz` file (Table 5.2.2). We compared our numbers with the bandwidth ranges of a RAID-5 and a RAID-6 with the typical 10MB/s low-end cap and achievable high bandwidth bound based on measurement.

The high bandwidth bound of *Legend* shows that regeneration can recover at a rate faster than the common throttled threshold. While *Legend*'s low bandwidth bound is lower than the threshold, prioritized regeneration of files in need can still be more responsive than waiting for the recovery of the entire device content for RAIDs. In addition, *Legend* can continue service beyond 2-disk failures. Finally, *Legend* can leverage a growing number of CPUs to parallelize recovery.

Table 5.2.1: Experimental platform.

Processor	4x2GHz Intel® Xeon® E5335, 128KB L1 cache, 8M L2 cache
Memory	8GB 667Mhz, DDR2
Disk	4x73GB, 15K RPM, Seagate Cheetah®, with 16MB cache

Table 5.2.2: Recovery bandwidth.

System configurations	Bandwidth (±1%)
FUSE + *Legend*	0.3 – 42 MB/s
RAID-5 1-disk recovery	10 - 96 MB/s
RAID-6 2-disk recovery	10 - 50 MB/s

Table 5.3.1: Filebench with file-server personality, configured with default flags and 100K files.

System configurations	Bandwidth (±1%)
ext4 + RAID-0	84.1 MB/s
ext4 + RAID-5	35.0 MB/s
FUSE + Unionfs + ext4	22.8 MB/s
FUSE + *Legend*	20.7 MB/s
FUSE + *Legend* + tracing	16.6 MB/s

5.3. Overheads

We first ran Filebench (v.1.4.9.1) [2014] with a file server personality, configured with default flags and 100K files. Table 5.3.1 shows that moving from RAID-0 to RAID-5 incurs ~60% of the overhead. Since *Legend* is built on Unionfs, which is built on the user-level FUSE, we found that the majority of the overhead arises from FUSE+Unionfs, whereas current non-optimized *Legend* incurs a 9% overhead when compared to FUSE+Unionfs. The overhead of `strace` is highly load dependent. In this case, it reduces the bandwidth by another 20%.

The next experiment involves compiling Dungeon Crawl. The elapsed time of *Legend* (595 seconds) is within 1% compared to FUSE+Unionfs (594 seconds). `strace` introduces 10% more overhead (656 seconds).

The current non-optimized single-threaded trace analyzer processes uncompressed traces at 90MB/min. Optimizing the *Legend* framework and exploring other lightweight trace techniques will be future work.

6. Related Work

The closest work to *Legend* is D-GRAID [SIV04a], which groups a file and its related blocks into isolated fault units (mostly in terms of directories) and aligns them to storage devices. By doing so, a file, its metadata, and parent directories (replicated at times) are less affected by the failure of other disks. *Legend* replicates file paths as well, as needed, but it uses dynamic generative dependency information to identify implicit replicas, and it leverages regeneration for recovery.

Generative dependencies have been used to recover files from malicious attacks [GOE05; HSU06; KIM10]. However, these solutions are not designed for graceful degradation in the face of device failures.

Legend can be viewed as an example of applying specific types of provenance [MUN06] to improve reliability. However, provenance gathering has been resource intensive, until the Arnold File System [DEV14] demonstrates how system-wide information can be gathered efficiently. *Legend* sees this advance as a potential springboard to improve reliability via file regenerations.

7. Conclusions

We have presented the design, implementation, and evaluation of the *Legend* file system, which exploits the use of implicit replicas to improve reliability. Our results show that, under certain workloads, the use of implicit replicas can achieve a better process completion rate than 2-way replication in the face of multiple disk failures. While the recovery speed depends on the effort to regenerate files, it can be mitigated with prioritization and parallelization via the growing availability of cheap CPU cycles. *Legend* can complement existing approaches for overcoming multiple storage device failures and achieving graceful degradation.

References

[BUR14] Burt J. eWEEK at 30: Multicore CPUs Keep Chip Makers in Step with Moore's Law. *eWeek*, February 20, 2014.

[CAR15] Carbonite Pro: Nonstop Workstation Backup for Nonstop Business. www.carbonite.com, 2015.

[DEV14] Devecsery D, Chow M, Dou X, Flinn J, Chen P. Eidetic Systems. *Proceedings of the 2014 USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[DUN16] Dungeon Crawl, https://crawl.develz.org, 2016.

[FIL14] Filebench. http://filebench.sourceforge.net/wiki/index.php/Main_Page, 2014.

[GOE05] Goel A, Po K, Farhadi K, Li Z, de Lara E. The Taser Intrusion Recovery System. *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

[GRU12] Grupp LM, Davis JD, Swanson S. The Bleak Future of NAND Flash Memory. *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*, 2012.

[HAR07] Harris R. Why RAID 5 Stops Working in 2009. ZDNet, July 2007. http://www.zdnet.com/article/why-raid-5-stops-working-in-2009, 2007.

[HSU06] Hsu F, Chen H. Ristenpart T, Li J, Su Z. Back to the Future: A Framework for Automatic Malware Removal and System Repair. *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC)*, 2006.

[KIM10] Kim T, Wang, X, Zeldovich N, Kaashoek MF. Intrusion Recovery Using Selective Re-execution. *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[MIL09] Mills E. Carbonite Sues Hardware Maker, Reseller, CNET, http://www.cnet.com, March 25, 2009.

[MUN06] Muniswamy-Reddy KK, Holland DA, Braun U, Seltzer M. Provenance-Aware Storage Systems. *Proceedings of the USENIX Annual Technical Conference*, May 2006.

[MUR14] Muralidhar S, Lloyd W, Roy S, Hill C, Lin E, Liu W, Pan S, Shankar S, Sivakumar V, Tang L, Kumar S. F4: Facebook's Warm BLOB Storage System. *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[OUS10] Ousterhout J, Agrawal P, Erickson D, Kozyrakis C, Leverich J, Mazieres D, Metra S, Narayanan A, Parulkar G, Rosenblum M, Rumble SM, Stratmann E, Stutsman R. The Case for RAMClouds: Scalable High-Performance Storage Entirely in RAM. *ACM SIGOPS Operating Systems Review*, 43(4):92-105, 2010.

[PAR13] Thomas Park, Data Replication Options in AWS. Amazon Web Services, http://awsmedia.s3.amazonaws.com/ARC302.pdf, 2013.

[PAT98] Patterson DA, Gibson G, Katz RH. A Case for Redundant Arrays of Inexpensive Disks (RAID). *Proceedings of 1988 ACM SIGMOD International Conference on Management of Data*, 1998.

[PIN07] Pinheiro E, Weber WD, Barroso LA. Failure Trends in a Large Disk Drive Population, *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)*, 2007.

[SCH07] Schroeder B, Gibson GA. Disk Failures in the Real World: What Does an MTTF of 1,000,000 Hours Mean to You? *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)*, 2007.

[SHV10] Shvachko K, Kuang H, Radia S, Chansler R. The Hadoop Distributed File System. *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010.

[SIV04a] Sivathanu M, Prabhakaran V, Arpaci-Dusseau AC, Arpaci-Dusseau RH. Improving Storage System Availability with D-GRAID. *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, 2004.

[SIV04b] Sivathanu M, Bairavasundaram LN, Arpaci-Dusseau AC, Arpaci-Dusseau, RH. Life or Death at Blocklevel. *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation (OSDI)*, 2004.

[SZE05] Szeredi M. Filesystem in Userspace. http://userspace.fuse.sourceforge.net, 2005.

[TIM10] Timor A, Mendelson A, Birk Y, Suri N. Using Underutilized CPU Resources to Enhance Its Reliability. *IEEE Transactions on Dependable and Secure Computing*, 7(1):94-109, 2010.

[WEL08] Welch B, Unangst M, Abbasi Z, Gibson G, Mueller B, Small J, Zelenka J, Zhou B. Scalable Performance of the Panasas Parallel File System. *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, 2008.

[UNI16] Unionfs: A Stackable Unification File System. http://unionfs.filesystems.org, 2016.