An Exploration of Targeted Provenance for Reliability

Robert Roy
Computer Science Department, FSU
Tallahassee, FL 32306
USA
roy@cs.fsu.edu

Erika Dennis
Computer Science Department, FSU
Tallahassee, FL 32306
USA
dennis@cs.fsu.edu

An-I Andy Wang
Computer Science Department, FSU
Tallahassee, FL 32306
USA
awang@cs.fsu.edu

Peter Reiher
Computer Science Department, UCLA
Los Angeles, CA 90095
USA
reiher@cs.ucla.edu

Sarah Diesburg
Computer Science Department, UNI
Cedar Falls, IA 50614
USA
diesburg@cs.uni.edu

ABSTRACT

Recent advances show that system-wide provenance gathering can be performed with reasonable overhead. In addition, generative dependencies have been used to recover files for non-reliability purposes. We pose the following research question: Can provenance be used to regenerate lost files in the event of storage failures to improve reliability?

We have designed, prototyped, and assessed the feasibility of the *Legend* file system, which exploits targeted provenance to regenerate lost files in order to improve reliability. Under a number of workloads, we have observed *Legend* when combined with 2-way replication can outperform or match 3-way replication. The regeneration rate exceeds the throttled recovery bandwidth for common RAIDs.

CCS CONCEPTS

• **Computer systems organization → Reliability**

KEYWORDS

storage, provenance, replication, performance, reliability

ACM Reference format:

R. Roy, E. Dennis, A. Wang, P. Reiher, and S. Diesburg. 2020. In *Proceedings of ACM SAC Conference, Brno, Czech Republic, March 30- April 3, 2020 (SAC'20),* 7 pages. DOI: 10.1145/3341105.3373860

SAC'20, March 30 –April 3, 2020, Brno, Czech Republic

https://doi.org/10.1145/3341105.3373860

1 INTRODUCTION

The quest for higher storage capacity has placed a growing strain on reliability mechanisms. Greater storage demand leads to more devices in a system, increasing the chance of the system encountering device failures [6].

Common solutions rely on some form of data redundancy to mask failures. For example, RAID-5 exploits partial data redundancy and can survive a single device failure, assuming independent device failures. However, real-world device failures are not independent [4, 8]. Higher storage capacity also increases the chance of a device hosting corrupted bits [15, 19], undetected until the recovery time. Thus, enterprise deployments sometimes resort to the use of 2- or 3-way full data replications [13, 18], which impose high storage overhead and can be cost-prohibitive for small-to-mid-scale deployments.

We have observed that file regeneration could be used as a form of data redundancy. A form of targeted provenance [14] can be used to log the dependency information required for file regeneration. Based on these observations, we ask the following research question: Can targeted provenance and file regeneration show promise to improve reliability?

To test this question, we introduce the *Legend* file system, which exploits file regeneration to guard against data loss. The key observation is that a combination of data and/or program file(s) may be used to generate other files. A file set used to generate a file can serve as an implicit replica of the regenerated file. Through the evaluation of our system, we have observed the most promise with a software development workload due to higher numbers of implicit file replicas. We also show that *Legend* degrades gracefully as the number of failed storage devices increases. *Legend* additionally mitigates the high storage capacity overhead imposed by n-way replications.

2 OBSERVATIONS AND MOTIVATION

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

2.1 Efficient Replication Using File Relationships

Common reliability schemes do not exploit the provenance-based relationships among files, overlooking potentially embedded data redundancy. For example, suppose `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 Data Redundancy

Suppose `X.jpg` is generated via running `bmp2jpg X.bmp`. All files associated with `bmp2jpg` 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 such as simulation or general multi-stage data processing, where data files are statistically or visually transformed and analyzed [7].

2.3 Practical to Recall Past System States

Recent systems, such as Arnold [5], have shown the feasibility of recalling all system states, including all file versions, by logging and compressing all dependency information (e.g., input files, X-Window events). Arnold logs about 3GB/day, with a performance penalty under 8%. Thus, the implicit-replica approach can potentially be applied system-wide.

2.4 Tradeoff with Computation and Storage

While trading computation with storage capacity for reliability may seem expensive, the continuation of Moore's Law in the number of CPU cores [12], along with the availability of massive parallelism, may make this tradeoff possible [3].

3 THE *LEGEND* FILE SYSTEM

The *Legend* File System exploits the notion of implicit replicas to improve reliability. As a proof of concept, *Legend* currently targets files that can be regenerated without dependencies on external inputs, while future inclusion of such inputs can broaden the benefit coverage. However, even using files without dependencies, we have found improvements in overall reliability. Unlike existing RAIDs, *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 an example, due to readers' likely familiarity with this type of computation and its richness in corner cases.

3.1 Identifying Implicit Replicas

3.1.1 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 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 inbound edges, such as writeable files, as potentially 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.

3.1.2 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.

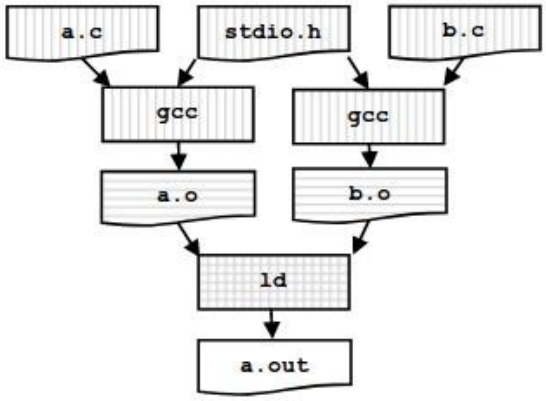


Figure 1. A simplified file-creation dependency graph for a compilation example. Rectangles are executables, and the document boxes are input and output files. The boxes containing vertical bars are one implicit replica of `a.out`. The boxes containing horizontal bars are the second implicit replica of `a.out`. `ld` is a member of both implicit replicas and contains vertical and horizontal bars.

3.1.3 Implicit Replicas: Figure 1 shows a simplified file creation dependency graph for a compilation example. `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`.

3.1.4 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. 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`.

3.1.5 Graph Consistency: A file generated based on out-of-date dependencies or upper stream input file updates may have a mismatching checksum for the output file. Therefore, it is important that *Legend* recognize when an updated file is no longer an implicit replica. Should a mismatch occur, however, we may fall back to the next layer of reliability mechanism (e.g., combining *Legend* with 2-way replication or other backup). One implication of combining *Legend* with another backup scheme is that we do not need to maintain our graph consistency as aggressively. While traces are gathered continuously, daily updates of the dependency graph would be sufficient. That also means that we can afford to replicate the dependency graph on all devices for fault tolerance.

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.

3.1.6 Nondeterminism: Files under `/tmp` often have randomly generated file names 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.

3.2 Implications of Using Implicit Replicas

The use of implicit replicas has two 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. However, it may increase the CPU parallelism to regenerate implicit replicas that are stored across devices.

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.

Currently, 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 (Figure 2). If a file belongs to multiple implicit replicas, we duplicate it to avoid correlated failures.

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

Figure 2: An example of an implicit replica assignment.

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 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 threshold of the maximum implicit replica access frequency to the average replica access frequency to determine whether to balance the loads.

3.4 Namespace 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 path replications. For example, files `/dir/A` and `/dir/B` can be assigned to devices 1 and 2, respectively. However, this arrangement would require `/dir` to be replicated on both devices 1 and 2. Should those two files be stored on the same device, `/dir` no longer needs to be replicated.

4 IMPLEMENTATION

The two major components of *Legend* are the trace analyzer and the file system itself. We prototyped *Legend* via the Filesystem in User Space framework (FUSE) [16] running atop ext4 in Linux (Figure 3). FUSE allowed us to prototype our system quickly at the user level, with associated performance overheads. As future work, we will port *Legend* into the kernel.

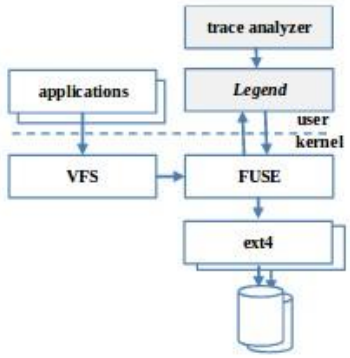


Figure 3: *Legend* components (shaded) and the data path from applications to the underlying ext4.

The trace analyzer gathers system call traces related to process executions and file-system-related calls 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 Unionfs [1] as the code base for *Legend*, which can combine multiple file systems into a single namespace. To illustrate, if `/dir/file1` resides on device 0 and if `/dir/file2` resides on device 1, `/dir` is then replicated on both devices, so that either device fails, `/dir` is still accessible. In our case, we ran ext4 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 backtrace 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 to 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 conservatively estimate the time to reissue individual execution system calls. Thus, our system tries to reissue top-level user commands (e.g., make) when applicable during recovery to better exploit concurrency.

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 as 90% confidence intervals.

5.1 Reliability

We built and used a 15-disk simulator 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. We omitted the results for RAID-5 and RAID-6 since they can either serve 100% of the requests or 0% of the requests after a threshold number of storage devices fails (1 for RAID-5 and 2 for RAID-6).

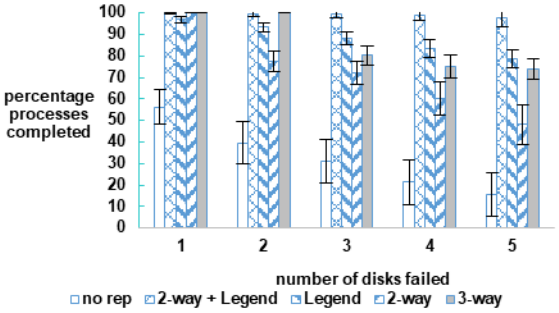
The simulator was populated with contents based on various traces. The first seven-day 6.6MB compressed trace was gathered from a software development setting. The trace contains 400K references to 29 million unique files (9.6 GB unique bytes) from 1,440 execution calls. Note that the trace can be processed incrementally during off-peak hours each day, and captured dependencies can be represented in a more compact table. In this case, the 6.6MB trace can be represented with a 679KB dependency table. The second 10-day 1.0GB compressed trace was gathered from a meteorology workstation running radar plotting, image generation, and statistical analysis. The trace contains 900K references to 63 million unique files (18 GB unique bytes) from 47K execution calls. The resulting dependency table size is 342KB. The third 10-day 189MB compressed trace was gathered from a meteorology student server running statistical analysis workloads. The trace contains 2.6K references to 192K unique files (180 MB unique bytes) from 4.4K execution calls. The resulting dependency table size is 599KB.

Given that *Legend* regenerates files based on past reference patterns, a longer deployment will yield a greater coverage of files. For the simulation, 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 [17].

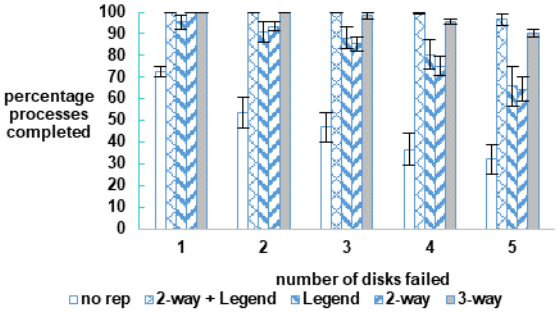
Figure 4 compares the reliability of different schemes as the number of failed storage devices increases. The reliability provided by various schemes varies depending on the workloads. The software development trace contains the most generative dependencies among file groups (13K edges), followed by the meteorology workstation trace (9.5K edges). The student server trace only contains 569 edges. *Legend* performs better with more generative dependencies (Figure 4(a)). For the software development trace, with 3 disk failures, *Legend* can outperform the no recovery option by 50%, and *Legend* even outperforms 3-way replication. For workloads with fewer generative

dependencies, *Legend* performs less well compared to 2- and 3-way replication.

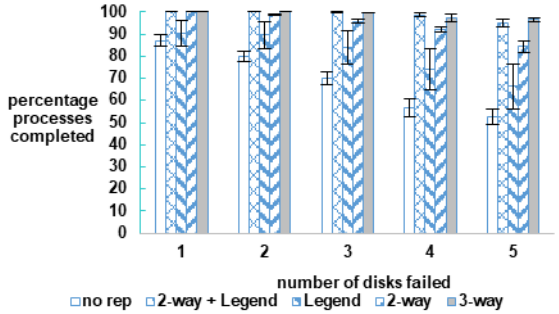
As an alternative, we combined *Legend* with 2-way replication, which means in the case of device failures, a file can be recovered either through the second replica or through regeneration. Figure 4(b) and 4(c) show that the reliability of this combined setting can exceed or match the reliability of 3-way replication without the storage overhead of 3-way replication. Another possibility is to redirect the space used by files with implicit replicas to increase the effective replication factor of other files. We will explore it as future work.



(a) software development workload.



(b) meteorology workstation workload.



(c) meteorology student server workload.

Figure 4: Percentage of unaffected processes vs. number of failed disks.

5.2 Recovery

We compared the recovery performance of *Legend* stacked atop of 4-disk, per-disk ext4 via FUSE with that of the no-FUSE baseline ext4-based RAIDs.

Table 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

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 the game Dungeon Crawl (v0.13.0, with 13,157 files) [11], and for the higher bound, we measured the regeneration of one of our trace files from a 15MB `.gz` file (Table 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.

Table 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

The high bandwidth bound of *Legend* shows that regeneration can recover at a rate faster than the common throttled threshold. While *Legend*'s lower 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.

5.3 Overheads

We first compare the storage overhead of *Legend* to store dependency information to the overhead to store redundant data for RAID-0, RAID-5, 2-way replication, and 3-way replication (Table 3). *Legend's* storage overhead for the dependency table is small (<0.1%). If the size of compressed traces is considered as well, the overhead can be as high as 10%, assuming the traces are converted into the table form each day.

Table 3: Storage Overhead.

System configurations	Storage overhead
RAID-0	1x
RAID-5	1.25x
2-way replication	2x
3-way replication	3x
Legend	<1.01x - 1.1x
Legend + 2-way replication	2x to 2.1x

We then ran Filebench [10] with a file server personality, configured with default flags and 100K files. This workload personality contains creates, deletes, appends, reads, writes and attribute operations on a directory tree. Table 4 shows that moving from RAID-0 to RAID-5 incurs 60% of the overhead. Since *Legend* is prototyped on Unionfs, which is built on user-level FUSE, we found that 73% of the overhead is from FUSE+Unionfs, and *Legend* incurs a 9% overhead when compared to FUSE+Unionfs. The overhead of strace is highly load-dependent. In this case, strace reduces bandwidth by another 20%. Exploring other lightweight trace techniques will be future work.

Table 4: Filebench with File-Server Personality, Configured with Default Flags and 100K Files.

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

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 1.5MB/sec. Optimizing this analyzer is future work. For example, to speed up real-life deployments, the trace analyzer could run daily during non-peak hours.

6 DISCUSSIONS

In developing *Legend*, we identified several additional research questions raised by targeted provenance.

6.1 Prioritized File Recovery

As a reliability mechanism, targeted provenance introduces context by working at the file granularity. With the added context, it is possible to perform prioritized recovery, ensuring that essential processes have their necessary files restored as quickly as possible, leaving less essential files to be restored later. This leaves the open question of what kind of priority algorithm to use. In other words, which files are most important to be recovered first? How is this determined? Using a working temporal time slice as a guide, it might be possible for *Legend* to identify likely candidates for prioritized recovery [21].

Exchanging computational resources for storage resources allows us to take advantage of the variance in computational resource utilization. While the utilization of storage is persistent and fixed, CPU utilization is more periodic. This allows us to take advantage of periods of low CPU utilization for the purpose of recovery.

6.2 Workloads

We believe that the added context allows our approach to be generalized to additional forms of recovery. Graphic and media design, document editing, system maintenance, and any process or workflow that can be described in terms of dependency graphs might benefit from recovery using targeted provenance.

7 RELATED WORK

The closest work to *Legend* is D-GRAID [17], 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 [2, 9, 20] and to reduce the amount of network transmitted data [7]. However, these solutions are not designed for graceful degradation in the face of device failures.

Within the context of high-performance computing, Spark leverages the concept of resilient distributed data sets to allow data to be recomputed in the event of failures [22].

Legend can be viewed as an example of applying specific types of provenance [14] to improve reliability. However, provenance gathering has been resource-intensive, until the Arnold File System [5] demonstrates how system-wide information (including external inputs, X-Window events, and IPCs) can be gathered efficiently for lineage tracking and queries. *Legend* sees this advance as a potential springboard to improve reliability via file regenerations.

Legend isolates files on disks based on their level in a process hierarchy. This is done to eliminate the possibility that a given disk failure impacts multiple stages in a given file's generation, maximizing the possibility that we can regenerate a lost file. Similar methods for isolating explicit replicas demonstrate the

viability of this approach and are generalizable to our notion of implicit replicas [24].

We recognize that other implicit sources of redundancy may exist across machine boundaries (e.g. git-cloned repositories, cloud-synced files, web caches, etc.). We will investigate such possibilities in the future in order to develop an integrated solution.

8 CONCLUSIONS

We have presented the design, prototype, and feasibility assessment 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. When combined with 2-way replication, we can potentially exceed or match the reliability of 3-way replication without the storage overhead. 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

[1] 2016. Unionfs: A Stackable Unification File System. In *http://unionfs.filesystems.org*.

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

[3] Timor A, Mendelson A, Birk Y, and Suri N. 2010. Using Underutilized CPU Resources to Enhance Its Reliability. In *IEEE Transactions on Dependable and Secure Computing*. 94–109.

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

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

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

[7] Dou X, Chen PM, and Flinn J. 2017. Knockoff: Cheap versions in the cloud. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*.

[8] Mills E. 2009. Carbonite Sues Hardware Maker, Reseller. In *CNET*.

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

[10] Filebench, https://github.com/filebench/filebench/wiki

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

[12] Burt J. 2014. *eWEEK* at 30: Multicore CPUs Keep Chip Makers in Step with Moores Law. In *eWeek*.

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

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

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

[16] Szeredi M. 2005. Filesystem in Userspace. In *http://userspace.fuse.sourceforge.net*.

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

[18] Park T. 2013. Data Replication Options in AWS. In *Amazon Web Services*.

[19] Harris R. 2007. Why RAID 5 Stops Working in 2009. In *ZDNet*.

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

[21] Kuenning GH, Popek GJ. 1997. Automated Hoarding for Mobile Computers. In *USENIX Symposium on Operating Systems Design and Implementation*.

[22] Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I. Spark: Cluster Computing with Working Set. *Proceedings of the 2nd USENIX Workshop on Hot Topics in Cloud Computing*, 2010.

[23] SNIA, Storage and Networking Industry Association, https://www.snia.org/, 2019.

[24] Cidon A, Rumble S, Stutsman R, Katti S, Ousterhout J, Rosenblum M. Copysets: Reducing the Frequency of Data Loss in Cloud Storage. *Proceedings of the 2013 USENIX Annual Technical Conference*, 2013.