Cloud-Native File Systems

Remzi H. Arpaci-Dusseau and
Andrea C. Arpaci-Dusseau
Computer Sciences Department
University of Wisconsin–Madison

Venkat Venkataramani
Rockset, Inc.

Abstract

We present the case for cloud-native system design, focused on the creation of CNFS, a local file system built specifically for the cloud era. We first present numerous storage and CPU design principles that any cloud-native storage system should consider; we demonstrate the utility of these principles through the design of CNFS. CNFS is a hierarchical, copy-on-write file system that migrates data and metadata across cloud storage volumes to meet user objectives, and harnesses remote CPU workers to perform critical background work such as migration and compression.

1 Introduction

The landscape of computer system design and implementation is undergoing a disruptive sea-change. The advent of cloud computing [4] has transformed the basic substrate for systems building: instead of a physical infrastructure of machines, developers now can implement services upon a sophisticated virtualized platform [8,18], utilizing well-tested and heavily used distributed services to realize their end goals.

The first generation of services and systems built in the cloud are called *cloud-enabled systems*. In this version of the cloud, systems from the pre-cloud world are "ported" to the cloud, but run quite similarly to their pre-cloud selves. As a result, these systems do not take advantage of the scalable and robust services provided within clouds, nor do they take advantage of the fundamental benefits of copious (rented) compute cycles and storage.

We believe the next generation of services and systems must take a critical step forward, away from simply being cloud-enabled to become *cloud-native systems* [37]. Cloud-native systems are designed not just to take advantage of the rentable nature of computing infrastructure, but intrinsically utilize now-standard cloud services to realize their end goals. For example, scalable, reliable distributed storage (e.g, Amazon's S3 [2], Google's Cloud Storage [19], Azure's Blob Storage [27]) is now ubiquitous; these services form a strong storage base upon which to build systems, instead of building upon simple collections of raw storage resources (e.g., disk drives [5]). Similarly, new serverless compute platforms [20, 28] (such as Amazon's Lambdas [9] and Google's Cloud Functions [29]) enable users to launch small pieces of computation on demand, scaling up or down readily, all without considering issues such as server provisioning or maintenance. Cloud-native systems exploit these basic cloud services to realize new, more flexible, high-performance, reliable systems and services more readily than ever before. An excellent current example is the Snowflake data warehouse [12], which is realized entirely atop Amazon services while providing high performance, reliability, security, and elasticity to clients.

In this paper, we focus our discussion on a specific type of cloud-native system, the *cloud-native local file system*. Currently, when running Linux on a virtualized instance in the cloud, one normally mounts a standard file system (such as ext4 or XFS) on a virtual block storage system, such as Amazon's EBS [35]. Each EBS volume can be configured, perhaps choosing high performance (e.g., a high-IOPS, costly SSD-backed replicated partition) or low cost (e.g., a low-IOPS, inexpensive disk-backed partition). The file system itself remains unchanged, unaware that is even running upon a virtual storage system.

Our goal is simple: to reconsider how such a local file system should be built, given the presence of virtualized block storage such as provided by Amazon EBS and Google Persistent Disk. We thus discuss, in Section 3, the design of *CNFS*, a ground-up rethinking of the local file system which takes a hard dependency on these cloud services.

The CNFS architecture currently has the following form. CNFS is a hierarchical, copy-on-write file system that uses remote cloud workers to perform background tasks such as migration and deduplication; background work on remote machines improves foreground performance (by offloading it to distant CPUs) and also can harness multiple CPUs in parallel to perform such work quickly. CNFS migration is at the heart of the cost/performance trade-off presented to users; specifically, CNFS moves data and metadata across differently configured storage volumes to meet user cost and performance goals.

The CNFS design is rooted in numerous cloud-native principles, which we discuss and present in the next section. We consider two large classes of principles (storage and CPU) and one overarching principle (the cost/performance trade-off). The storage principles, in short, summarize what is important about building systems upon modern cloud storage infrastructure, including critical aspects such as reliability, capacity, cost, performance, and hierarchy; the CPU principles focus on similarly important concerns regarding the com-

putation needed within systems, focusing upon parallelism, capacity, scaling, remote work, and hierarchy. Finally, the overarching principle centers around the cost/performance trade-off that cloud-native systems must make. By taking all such concerns into account, CNFS can deliver local file system service that meets the performance, reliability, and cost needs of the client.

In this paper, we present the following, looking both to spark discussion in the area of cloud-native systems as well as to solicit feedback on the specifics of CNFS. We begin by presenting our cloud-native principles in Section 2. We follow this with a discussion of CNFS in Section 3. Finally, we then present related work in Section 4 and conclude in Section 5.

2 Cloud-Native Principles

We now outline a number of *cloud-native principles* that underly this vision, building on early thinking in this space by Venkataramani [37]. The principles, when applied correctly, can highlight new points in the systems design space which are directly enabled by the modern cloud. We group these principles into three major groups: storage principles, CPU principles, and one overall principle.

2.1 Storage Principles

Our first focus is on storage principles: how should developers of cloud-enabled systems view the storage services offered within the cloud?

Storage reliability principle: *Highly replicated, reliable, and available storage is widely available.* One obvious principle of cloud-native systems is the ubiquity of reliable (highly durable) storage (11 "9s" according to Amazon [3]). Much of the work of creating and managing replicas (for durability) can be pushed down into the infrastructure; as building replicated storage is challenging [16], utilizing a stable and widely tested system instead of rolling one's own is likely a wise option.

These systems exist in many forms, and thus one choice that must be made is how best to utilize them. For example, in some cases an object-based interface, such as that provided by Amazon S3, will likely be the best choice; in other cases, using a lower-level block-based interface such as Amazon's EBS will be best. In both cases, taking advantage of the reliability characteristics provided by these replicated services is key.

Storage cost principle: *Storage space is generally inexpensive.* There are two important perspectives that arise from the low cost of cloud storage. The first, and perhaps most important, is that storage, for most use cases, can largely be thought of as "free". For example, if one has 1 TB of data, it will cost only $4 per month to store this in archival storage (Glacier), and only somewhat more to store it in higher performance tiers.

Nearly-free storage has strong implications for higher-level systems. For example, if making a specific type of index over data can improve performance, the space costs of doing so are so low that paying the cost for the space the index uses is likely well worth it. More generally, one should consider all the possible places in a system's design where using more space can improve system behavior.

The second important point is that, despite nearly free storage, storage is not absolutely free. Thus, optimizations to put as much "cold" data into cheaper tiers and only "hot" data in higher-cost tiers are worth considering. One can't simply make thousands of replicas and associated indexes, put them in the highest-cost SSD tier, and expect to build a cost-effective system.

Storage capacity principle: *Large amounts of storage space are available*. In cloud storage, there are seemingly no limits on how much space users can use. For example, the S3 website states "The total volume of data and number of objects you can store are unlimited."

This principle has strong implications for systems design. For example, issues such as *space amplification* are no longer a central concern: there are plenty of bytes available. Similarly, extra space presents a performance opportunity: more indices, pre-computed answer caches, and other space-consuming optimizations, can all be used to speed execution, as stated above. Finally, there is no need for capacity planning; just use what is needed when it is needed.

Storage bandwidth/latency principle: *Storage services provided by cloud providers are generally high bandwidth; however, they have varying levels of latency.* The cloud substrate does not provide a perfect storage system, with effectively infinite bandwidth and incredibly low latency. However, bandwidth generally is scalable, whereas latency depends on the storage tier.

This principle also has strong implications. Because bandwidth is generally available, scaling out is relatively easy, and should be realizable wherever needed. However, realizing low-latency storage requires extensive consideration; placement of metadata/data into different performance tiers (including memory-only storage layers) may be required to achieve desired latency goals.

Storage bandwidth-cost principle: *Access to data is low cost.* This principle once again highlights two points. The first point is that for most use-cases, bandwidth costs should not be of primary concern; one should simply access data as need be, as the costs are low.

The second point is also important: when data is partitioned across tiers, access to "less expensive" capacity tiers is generally more expensive. This point further suggests the need for careful management across tiers.

Storage hierarchy principle: *Storage is available in many forms, with noticeable differences in performance and cost across each level.* When combining the above principles, one realizes this hierarchy principle: data must be managed across levels of the cloud hierarchy, with more popular data in more expensive but faster storage, and less popular in slower cheaper storage.

Hierarchy management is thus central to cloud-native storage. Without it, a system will always reside in one (non-optimal) extreme, either paying too much for consistent high performance, or saving cost while delivering consistent poor performance.

2.2 CPU Principles

Our next set of principles focus upon computation. With the cloud-native mindset, systems can take advantage of the vast fleets of CPUs now available as need be to achieve their ends. How best to use them, of course, depends upon the system at hand.

CPU parallelism principle (or $A \cdot B = B \cdot A$): *It should cost roughly the same to execute on A CPUs for B time units as it does to execute on B CPUs for A units of time.* The ramifications of this principle are clear: if one can do work in parallel, one generally should, as it will complete faster and cost roughly the same.

In current clouds, this principle is true down to a certain time scale. For example, Amazon's Lambda service charges in 100ms increments; thus, if one can partition work into N 100ms (or more) units, one will pay a similar amount as running a single, longer running Lambda. Further, as Lambda services improve, this minimum time unit will likely decrease, enabling even finer-grained parallelism to be achieved.

CPU capacity principle: *Large numbers of CPUs are available.* What we saw with storage, we also see with CPU: there are essentially an unbounded number of CPUs available for computational tasks. Thus, if they are needed, they should be used. There is no need for the cloud-user or cloud-developer to worry excessively about exhausting CPU resources; that is the worry of the cloud provider.

The same corollary applies here as well: while CPUs are nearly free, they are not absolutely free. Thus, a framework for considering costs versus benefits would be useful towards making each utilization decision.

CPU scale-up/scale-down principle: *One should only use as many CPUs as needed for a task, and not more*. The parallelism principle suggests doing work in parallel, to the greatest extent possible; the capacity principle makes this possible. However, CPUs are not free. Thus, while systems should be willing to scale *up* to get work done quickly, they must also scale *down* to avoid paying for resources that are not needed at the moment.

The implication for system design is clear here as well. Systems should not take static or simplistic approaches to parallelism. Rather, careful monitoring of usage and adaptation is required to best utilize the cloud. One interesting possibility for scale up is found in "spot" instances, which are cheap virtual CPUs that can be quickly recruited, but can be reclaimed by the provider as needed, thus requiring robust systems design to best make use of their capabilities.

CPU remote-work principle: *When possible, use remote CPU resources to do needed work.* With shared storage, it is easy to perform work over data on remote machines, as data is available throughout the datacenter. This separation generally leaves front-end machines free to focus on serving current workload demands; background work can be done elsewhere.

CPU hierarchy principle: *CPU is available in different forms, with differences in performance, cost, and reliability across each level.* When renting CPU cycles, different levels of performance are available at different costs, thus creating a new knob for systems to tune; should a service use a few more powerful CPUs, or many less powerful ones?

2.3 The Overall Principle

As hinted at in both the storage and CPU principles, there is an underlying driving force behind the creation of truly native cloud systems, the cost/performance trade-off:

Overall performance/cost principle: *Every decision in cloud-native systems is ultimately driven by a cost/performance trade-off.* A cost/performance trade-off exists at the heart of many decisions in cloud-native systems and thus is a first-class concern. Not considering this factor leads to systems that are either *cost-oblivious* or *performance-oblivious*. In the former, the system may use excessive resources to get a task done quickly but at an exorbitant price; in the latter, a system may save pennies but return an answer sluggishly. Ideal cloud-native systems take both into account.

3 Case Study: Linux CNFS

Current Linux file systems were developed mostly for hard drives (e.g, ext4, XFS, ZFS) and occasionally for more modern media such as flash-based SSDs (e.g., f2fs). However, there is not yet a file system designed to operate effectively upon a cloud-based virtual storage platform such as Amazon's EBS or Google Persistent Disk. We now describe the Linux *Cloud-Native File System (CNFS)*, a first step in this new direction.

3.1 Underlying Assumptions

The CNFS design presumes the following from the cloud, and tailors its design towards maximally exploiting this environment. First, it assumes the presence of block-based virtualized storage such as Amazon's EBS or Google's Persistent Disk (PD), with the option of using a local scratch SSD or hard drive as needed.

Second, CNFS assumes that not only can a single client access the virtual volume, but that other machines in the system can also access the volume (for reasons explained below). Although Amazon does not allow such access for EBS, and Google does only in limited (i.e., read-only) fashion for PD, we believe there are excellent reasons to enable remote concurrent volume access, and thus assume it for the remainder of the discussion.

Third, CNFS assumes that the block store offers a number of different performance/cost configurations. Amazon, for example, provides two different SSD volumes and two different HDD volumes, with different performance/cost trade-offs. One CNFS focus will be to understand and exploit these different configurations to optimize performance, cost, or both for a given client.

3.2 Basic Design

Our basic design for CNFS derives its design from the following cloud-native principles:

- The storage hierarchy principle. CNFS inherently understands that it is built atop virtualized block-based storage systems, with volumes that can be configured at different cost/performance levels.
- The CPU remote-work principle. CNFS wishes to reserve local CPU for foreground performance. Thus, CNFS itself is structured to push as much file-system work as possible to the background; CNFS can then take advantage of remote resources to perform the necessary background work.
- The overall performance/cost principle. Because CNFS is built to run in the cloud, it must intrinsically understand that decisions it makes have cost and performance ramifications. Thus, CNFS must have a cost-performance framework as part of its algorithmic core, enabling sensible trade-offs based on user desires; it must also export interfaces to enable user control over these features.

While these principles are at the core of CNFS (v1.0), other principles also underly our work. For example, using many CPUs to do background work is natural within CNFS (CPU parallelism), and CNFS presumes reliable replicated storage as its substrate (Storage reliability).

We thus envision the following design for CNFS. At its core, CNFS is a copy-on-write (COW) file system [7, 24, 25, 30], never over-writing data or metadata in place but rather always writing data to unused storage.

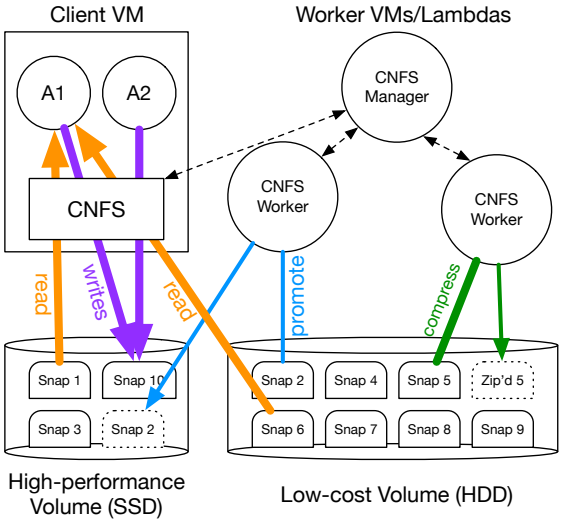

Figure 1: **CNFS I/O Flows.**

COW techniques are critical within CNFS, as they fundamentally enable remote work to be done upon read-only snapshots of the file system, thus obviating the need for complex synchronization between the foreground client and background workers.

CNFS is also a hierarchical file system, building upon classic [14, 26] and more recent [22] work in this area. CNFS actively moves files and directories across underlying storage volumes with different cost/performance properties, seeking to optimize usage based on user cost and performance requirements.

In addition, CNFS aggressively performs compression/deduplication [39] as needed, saving storage space and cost in some cases, reducing bandwidth demands in others. This work is performed by remote workers, each of which can access portions of a read-only snapshot and transform it into a smaller, compressed format.

3.3 Example I/O Flow

To give a better idea of how CNFS operates, we describe an example of its possible usage. Figure 1 shows its basic operation. One the left, a client VM running two applications mounts CNFS and enables access to a CNFS virtual volume. Internally, CNFS maps data from this volume onto two cloud volumes, one a high-performance SSD (left) and the other a low-cost hard drive (right). All writes are directed to a current write point (Snapshot 10), whereas reads may refer to any active snapshot.

CNFS workers run on other machines, utilizing the parallelism of the cloud to enact background tasks such as compression or migration between volumes. In the figure, Snapshot 2 is being promoted to the SSD (because, perhaps, it has been accessed frequently in the recent past) and Snapshot 5 is being compressed (perhaps because it has been inactive).

A CNFS manager orchestrates all of this behavior, communicating with the CNFS file system and remote workers to ensure proper synchronization. The manager also serves as a policy engine, utilizing access statistics and policy goals gathered from CNFS to decide how to best place data and other related decisions.

3.4 Status and Research Issues

CNFS is in early design phase, and thus we share it to spark new thoughts in the creation of cloud-native systems, as well as solicit feedback on its key elements. While building the core infrastructure, we are currently focusing upon the following research issues:

- **Cost/Performance APIs.** CNFS needs to export new APIs to specify cost/performance goals at many different granularities. How expressive should these controls be? What granularity is needed by higher-level systems and users? How should such information be tracked within the file system, and then shared with remote workers?
- **Cost and Performance Speculation.** CNFS monitors workloads and must make decisions about migration, compression, and other features, all while meeting user cost/performance goals. Thus, CNFS must be able to predict how various changes will affect future performance. What data must be collected to inform such decisions? What type of internal simulation and optimization framework should be built to make the decisions?
- **Client and Remote Worker Synchronization.** Because CNFS assumes the presence of cloud workers as part of its normal operation, correct and efficient synchronization between a client and workers is essential. How should synchronization be realized? How are worker faults detected and acted upon?
- **Scalable Access Monitoring.** CNFS decision-making, as described above, requires data on file and directory access. CNFS must thus track such information effectively and compactly, even for large file systems. Which information should be tracked? How often should it be communicated to the external manager? How should the manager store and use such data?
- **Synchronous Workload Performance.** In some cases, applications require frequent synchronization to durable storage for recovery purposes [11]. However, such workloads do not run well upon copy-on-write file systems (e.g., ZFS, while utilizing COW in its basic design, relies upon an intent log [7]). Does CNFS require logging for high performance? Can a replicated memory tier be used instead to provide durability and high performance for these styles of workloads?

4 Related Work

CNFS builds upon a long line of research. For example, numerous block-based systems internally reorganize data layout to improve performance or save space [1, 6, 17, 21, 34, 38]. An alternative to CNFS would be to build a smarter block layer, and thus realize similar benefits under an unchanged Linux file system. There are two primary reasons that adding this functionality at the file-system level is the better option. First, the file system has high-level information about semantically meaningful items such as files and directories; thus, providing fine-grained controls would be challenging (though perhaps not impossible [31, 32]) at block level. Second, migration of data at block level requires another layer of the system to implement crash-consistency machinery, thus complicating the system further [13]; CNFS provides a unified, simpler approach to crash consistency by handling such activity itself.

File systems that manage data and metadata across hierarchies have been studied for many years. Many early works focused upon migration of files from slow tape-storage systems onto higher-performing hard drives [14, 23, 33]. CNFS builds upon this work but on a modern substrate of SSD- and HDD-based volumes.

Most recently, Kwon et al. introduced Strata [22], a system that places data upon NVM, Flash-based SSD, and hard drive in a unified hierarchical storage system. CNFS differs in its focus on utilizing cloud resources (remote workers and a manager) to perform background tasks, using more extensive analysis to migrate files across tiers, and its integration with existing cloud storage offerings. It would be an interesting exercise to transform Strata into a cloud-native version of itself.

Others have noted the potential impact of the cloud on systems design. Notably, Dewitt and Lang speculate that cloud-based approaches will lead to the end of "shared nothing" architectures [15].

Finally, industry interest in cloud-native systems is on the rise. The Snowflake data warehouse [12] is a fully functional cloud-native data warehouse, serving as a pioneering example of what is possible. More recent efforts include RocksDBCloud [10], which places lower tiers of the RocksDB LSM tree into Amazon S3 buckets, and Kasten, a new venture investigating data management for cloud-native applications [36].

5 Conclude

We have presented CNFS, a first generation cloud-native file system. While many facets of its design and implementation are yet to be realized, we hope that its design, and the principles upon which it is built, can help move the field forward in this exciting new direction, thus enabling a new generation of high-performance, cost-effective storage systems to be realized.

References

[1] S. Akyurek and K. Salem. Adaptive Block Rearrangement. *ACM Trans. Comput. Syst.*, 13(2):89–121, May 1995.

[2] Amazon. Amazon Simple Storage Service (Amazon S3). `http://aws.amazon.com/s3/`, 2012.

[3] Amazon. Durability and Data Protection. `https://aws.amazon.com/s3/faqs/#Durability_.26_Data_Protection`, 2018.

[4] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. U.C. Berkeley Technical Report, 2009.

[5] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.9 edition, 2014.

[6] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. Liptak, R. Rangaswami, and V. Hristidis. BORG: Block-reORGanization for Self-optimizing Storage Systems. In *Proceedings of the 7th USENIX Symposium on File and Storage Technologies (FAST '09)*, San Francisco, California, Feb. 2009.

[7] J. Bonwick and B. Moore. ZFS: The Last Word in File Systems. `http://opensolaris.org/os/community/zfs/docs/zfs_last.pdf`, 2007.

[8] E. Bugnion, S. Devine, and M. Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 143–156, Saint-Malo, France, Oct. 1997.

[9] B. Butler. What Is Amazon Cloud's Lambda And Why Is It A Big Deal? Network World, Apr. 2016.

[10] I. Canadi, S. Dong, and Others. RocksDB-Cloud: A Key-Value Store for Cloud Applications. `https://github.com/rockset/rocksdb-cloud`, 2017.

[11] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Optimistic Crash Consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Nemacolin Woodlands Resort, Farmington, Pennsylvania, Oct. 2013.

[12] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, A. W. Lee, A. Motivala, A. Q. Munir, S. Pelley, P. Povinec, G. Rahn, S. Triantafyllis, and P. Unterbrunner. The Snowflake Elastic Data Warehouse. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data (SIGMOD '16)*, San Francisco, California, June 2016.

[13] T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Journal-guided Resynchronization for Software RAID. In *Proceedings of the 4th USENIX Symposium on File and Storage Technologies (FAST '05)*, pages 87–100, San Francisco, California, Dec. 2005.

[14] M. B. Deshpande and R. B. Bunt. Dynamic File Management Techniques. Proceedings of the 7th IEEE Phoenix Conference on Computers and Communication, 1988.

[15] D. J. DeWitt and W. Lang. Data Warehousing in the Cloud: The End of 'Shared-Nothing'. North East Database Day 2017, Jan. 2017.

[16] A. Ganesan, R. Alagappan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17)*, Santa Clara, California, Feb. 2017.

[17] R. Geist, R. Reynolds, and D. Suggs. Minimizing mean seek distance in mirrored disk systems by cylinder remapping. Performance Evaluation. Volume 20, Issues 1–3, May 1994.

[18] R. Goldberg. Survey of Virtual Machine Research. *IEEE Computer*, 7(6):34–45, 1974.

[19] Google. Google Storage for Developers. http://code.google.com/apis/storage/, 2011.

[20] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Serverless Computation with OpenLambda. In *The Eighth USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'16)*, Denver, Colorado, June 2016.

[21] W. W. Hsu, A. J. Smith, and H. C. Young. The Automatic Improvement of Locality in Storage Systems. *ACM Trans. Comput. Syst.*, 23(4):424–473, 2005.

[22] Y. Kwon, H. Fingler, T. Hunt, S. Peter, E. Witchel, and T. Anderson. Strata: A Cross Media File System. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)*, Shanghai, China, Oct. 2017.

[23] D. H. Lawrie, J. M. Randal, and R. R. Barton. Experiments with Automatic File Migration. *Computer*, 15(7), July 1982.

[24] R. Lorie. Physical Integrity in a Large Segmented Database. *ACM Transactions on Databases*, 2(1):91–104, 1977.

[25] C. Mason. The Btrfs Filesystem. `oss.oracle.com/projects/btrfs/dist/documentation/btrfs-ukuug.pdf`, Sept. 2007.

[26] B. McNurr. Background Data Movement in a Log-structured Disk Subsystem. *IBM Journal of Research and Development*, 38(1):47–58, 1994.

[27] Microsoft. Azure Blob Storage. azure.microsoft.com/en-us/services/storage/blobs/, 2017.

[28] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC '18)*, Boston, Massachusetts, July 2018.

[29] A. Odewahn. Serverless on Google with Cloud Functions and React. www.oreilly.com/learning/serverless-on-google-with-cloud-functions-and-react, July 2017.

[30] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Trans. Comput. Syst.*, 10(1):26–52, Feb. 1992.

[31] M. Sivathanu, V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving Storage System Availability with D-GRAID. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST '04)*, pages 15–30, San Francisco, California, April 2004.

[32] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-Smart Disk Systems. In *Proceedings of the 2nd USENIX Symposium on File and Storage Technologies (FAST '03)*, pages 73–88, San Francisco, California, April 2003.

[33] A. J. Smith. Long Term File Migration: Development and Evaluation of Algorithms. *Commun. ACM*, 24(8), Aug. 1981.

[34] A. J. Smith. Optimization of I/0 Systems by Cache Disks and File Migration: A Summary. Performance Evaluation, Volume 1, 1981.

[35] The AWS Team. Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region. http://aws.amazon.com/message/65648/, Apr. 2011.

[36] N. Tolia, V. Kamra, G. Matev, J. Lopez, T. Manville, A. Vorbau, I. Kislenko, and I. Mitra. Kasten. `https://kasten.io`, 2018.

[37] V. Venkataramani. Challenges, Economics, and Principles of Cloud Native Systems. Talk at University of Wisconsin–Madison, Oct. 2017.

[38] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID Hierarchical Storage System. *ACM Trans. Comput. Syst.*, 14(1):108–136, February 1996.

[39] B. Zhu, K. Li, and H. Patterson. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST '08)*, San Jose, California, Feb. 2008.