Using Disk Add-Ons to Withstand Simultaneous Disk Failures with Fewer Replicas

Eitan Rosenfeld Nadav Amit Dan Tsafrir
Technion – Israel Institute of Technology

Abstract

Contemporary storage systems that utilize replication often maintain more than two replicas of each data item, reducing the risk of permanent data loss due to simultaneous disk failures. The price of the additional copies is smaller usable storage space, increased network traffic, and higher power consumption. We propose to alleviate this problem with SIMFAIL, a storage system that maintains only two replicas and utilizes per-disk "add-ons", which are simple hardware devices equipped with relatively small memory that proxy disk I/O traffic. SIMFAIL can significantly reduce the risk of data loss due to temporally adjacent disk failures by quickly copying at-risk data from disks to their add-ons. SIMFAIL can further eliminate the risk entirely by maintaining *local* parity information of disks on their add-ons (such that each add-on holds the parity of its own disk's data chunks). We postulate that SIMFAIL may open the door for cloud providers to reduce the number of data replicas they use from three to two.

1 The Price of Faster Recovery

Storage systems often employ data replication to defend against disk failures. The data is partitioned into small contiguous units, and each unit is stored on more than one disk. If one replica happens to reside on a failing disk, the system can utilize another replica to recover.

A key parameter of replicating systems is the number of copies to maintain for each data unit. This parameter reflects a tradeoff between safety and efficiency. Storing fewer replicas increases the risk of simultaneously losing all the copies of a data unit due to several temporally adjacent disk failures. Conversely, storing additional replicas has the following drawbacks: (1) it reduces the effective storage size, as k replicas translate to $1/k$ usable space; (2) it induces higher network traffic, as more disks have to be synchronized upon data changes; and (3) it translates to greater energy consumption, because write operations induce additional disk/network activity, and because systems must utilize a higher number of disks to attain the same effective storage space.

In practice, maintaining only two replicas for each data unit might not be enough: (1) because disk failures exhibit temporal correlation, so if one disk fails, there is a higher chance a second disk will soon fail too [22]; (2) because there is a nonnegligible probability to encounter bad disk sectors while reading the surviving replicas during the recovery [18]; and (3) due to the manner by which contemporary replicating systems recover from disk failures, which further increases the risk, as discussed next.

Reconstructing a failing disk by copying all its data from replicas to a new disk takes a long time. For example, assuming a typical disk throughput of 50 MB/s, reconstructing a 2 TB disk would take more than 11 hours. During this interval, the system is vulnerable to additional failures that might cause data loss. Contemporary systems therefore employ a different recovery scheme, whereby, instead of reconstructing a new disk, they replicate the vulnerable data across all the remaining disks in a distributed manner. The scheme strips every disk across all other disks, thereby allowing many disks to participate in the recovery by reading/writing a relatively small portion of the data. This procedure can dramatically reduce the recovery time. For example, by utilizing a dense network, Microsoft's new FDS (Flat Datacenter Storage) is capable of replicating nearly 100 GB in about 50 seconds across a 100 disk system [16].

The price of the much faster recovery time is an increased risk of data loss due to multiple disk failures. The risk is greater because every disk is striped across every other disk, which means *any* two simultaneously failing disks would have a nonempty intersection that would be permanently lost. Importantly, the probability to experience double disk failure when each disk is striped across all others is proportional to size of the system. Namely, more disks imply a greater risk. In contrast, more traditional storage systems arrange disks in, e.g., small subclusters of RAID5 configurations, such that data loss can occur only if the double disk failure happens in the same sub-cluster, an event with a much lower probability.

For these reasons, contemporary replicating storage systems typically choose to sacrifice another sizable portion of their storage capacity and opt to employ three, rather than two, replicas. Such systems include the Google File System [10], Windows Azure Storage [1], Microsoft's FDS [16], HDFS [2], and OpenStack's [17] Swift [24].

2 But How Fast is the Recovery *Really*?

Let us carefully consider the duration of the procedure to recover from a single disk failure in replicating systems that stripe each disk in small chunks across many other disks. This duration is largely determined by two factors:

1. *Network Bandwidth* – Recovery is made fast because many pairs of disks simultaneously exchange relatively small chunks of data in parallel. But simultaneous exchanges are possible only if the network is powerful enough to avoid congestion in the face of the overwhelming surge of communication.

2. *Recovery Bandwidth* – System designers aspire to reduce the interference of background recovery activity with regular activity to avoid degrading the performance of foreground workloads. The recovery procedure can go unnoticed if it is throttled to consume only a fraction of the available disk/network bandwidth.

Notably, the aforementioned quick recovery of the experimental FDS (nearly 100 GB in 50 seconds) was made possible only: (1) because the recovery procedure was allowed to consume *all* available bandwidth, while *no* regular application activity was present in the system; and (2) because FDS employs a very dense network comprised of many switches that provides full bisection bandwidth, allowing *all* disks in the system to *simultaneously* send or receive their *entire* bandwidth [16].

Not all systems enjoy such a powerful network, in which case recovery might be slower, possibly significantly. But even as they do enjoy such a network, production systems (rather than experimental ones) would likely allow for a much smaller recovery bandwidth.

Consider, for example, the IBM XIV storage system, which maintains *two* replicas for each data unit and stripes each disk across all other disks. (XIV utilizes 180 x 3TB SATA disks and enough Infiniband switches to allow for a full bisection bandwidth, similarly to FDS.) The XIV book specifies that recovery from one disk failure can take up to 50 minutes [6], even though the task could be accomplished in about 11 minutes if utilizing all the bandwidth.[1]

Interestingly, this vulnerability window of 50 minutes (during which the system is exposed to a second disk failure and hence to permanent data loss) is the main argument XIV's industrial competitors use against it [3].

[1]Because each surviving disk keeps at most 3TB/179 $\approx$ 16GB of data from the failing disk. Conservatively assuming a 50 MB/s disk bandwidth, it takes 2 $\times$ 16GB / 50 MB/s $\approx$ 11 minutes to send and then receive 16GB between a pair of disks. Note that most I/O operations are serial because XIV keeps data in chunks of 1MB.

3 Layout of SIMFAIL

We have explained why replicating systems that stripe each disk across many other disks suffer from a greater risk of experiencing data loss. We have further explained why the faster recovery times such systems may theoretically enjoy can in fact be quite longer. Equipped with these understandings, our goal is to come up with a system design that would require only two replicas (as in XIV) but would eliminate the risk to experience data loss due to simultaneous disk failures.

Like the systems it models, SIMFAIL strips each disk on many other disks. But unlike existing systems, SIMFAIL does it in a coarse-grained, deterministic manner (namely, data units are of the size of GBs, and no randomization is involved in their placement on disk). Specifically, if the system is comprised of N disks of size S, then we divide each disk to $N-1$ contiguous data units of the size $S/(N-1)$. We call these units *superchunks*. For example, if the system has 101 x 1TB disks, then each superchunk is of the size 1TB / 100 = 10GB. We then associate each superchunk on one disk with a superchunk on another disk, such that every superchunk is associated with a different disk. This association determines how data is replicated on the disks in SIMFAIL.

Superchunk association changes are rare. Associations can change only when: (1) disks fail (or are removed from the system for some other reason); when (2) disks are added to the system (in place of failing disks, or if we want to increase capacity); or when (3) we choose to proactively balance load across disks in resolution of superchunks (in case some disks store significantly more data than others). Regardless of any association changes, throughout its lifetime, SIMFAIL consistently maintains two important invariants:

- *1-mirroring*, whereby every superchunk is mirrored by another superchunk on another disk, and
- *1-sharing*, whereby no two disks share more than one superchunk.

Table 1 gives an example of how superchunks could be laid out in a 7-disk system, such that the above two invariants are satisfied. The superchunks association function in this case is:

$$f(i,j) = \begin{cases} (\ \ i+1\ \ ,\ \ (j+\frac{i}{2}+1)\%N\ \ \ ) & \text{, even } i \\ (\ \ i-1\ \ ,\ \ (j-\frac{i-1}{2}-1)\%N\ \ ) & \text{, odd } i \end{cases}$$

where j is the index of the disk (column), i is the index of the superchunk on that disk (row), and $N = 7$.

Note that SIMFAIL does not impose any constraints regarding how storage systems should manage data within the superchunks. In particular, replicating systems like GFS, HDFS, FDS, Azure, and Swift set the granularity of

	D_1	D_2	D_3	D_4	D_5	D_6	D_7
S_1	1	2	3	4	5	6	**7**
S_2	7	**1**	2	3	4	5	6
S_3	8	9	10	11	12	**13**	14
S_4	13	14	**8**	9	10	11	12
S_5	15	16	17	18	**19**	20	21
S_6	19	20	21	**15**	16	17	18

Table 1: *Example for a superchunk layout that satisfies 1-sharing and 1-mirroring. Columns are disks. Rows are superchunks within disks. Numbers are IDs of superchunks. IDs in bold correspond to superchunks that mirror disk D_1.*

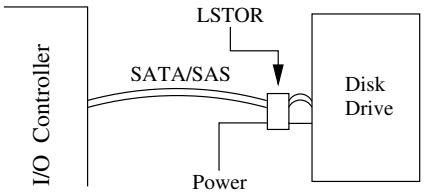

Figure 1: LSTOR *interposes between a drive and its controller.*

their data units to typically be a few MBs [1,2,10,16,24]. They still can and should do that on their own. Combined with such systems, the SIMFAIL superchunk layout is merely a meta-structure that determines the placement of copies. For example, assume the storage system utilizes 1 MB data units (as is the case in XIV). Given such a unit on some disk, SIMFAIL simply determines the location of its replica (and nothing else). The superchunk of the replica is decided by the aforementioned association, and the offset of the replica (within its superchunk) is set to be identical to the offset of the original unit.

4 Fault Tolerance in SIMFAIL

As noted, the goal of SIMFAIL is to mitigate the risk of experiencing permanent data loss due to temporally adjacent disk failures, while still employing only two replicas. SIMFAIL achieves this goal by augmenting the storage system with disk "add-ons", which we denote as LSTORs (that stand for "local storage").

An LSTOR is a small, simple device that has just enough computational power to understand the I/O traffic that flows to/from the disk to which it is attached, and just enough memory to allow it to store one superchunk. LSTORs interpose the I/O between their disk and its controller, as illustrated in Figure 1. We envision the size of an LSTOR to be similar to, e.g, a pinkie-sized SATA-to-USB converter that is sold by Amazon under $10. We further envision that LSTORs could be made part of disk drive enclosures. A critical property we require is that disk failures occur separately from LSTORs failures, such that LSTORs remain accessible after their disks fail.

In the simpler version of SIMFAIL, denoted SIMFAIL^s, LSTORs typically do nothing but observe the traffic that flows from the I/O controller to the disk. When SIMFAIL^s identifies that some disk D has failed, it labels the replicas of D's superchunks as *risky superchunks*. It then sends a "message" to all the LSTORs, informing them to copy the risky superchunk that resides on their disk onto their LSTOR memory; recall that 1-sharing ensures us that each surviving disk will hold one risky superchunk at most. (A "message" is a write of information that allows the LSTOR to identify that the data was sent to it rather than to its disk.)

Assuming a disk size of 1TB, a superchunk size of 10GB, and a disk bandwidth of 50 MB/s, SIMFAIL^s can replicate all the content of D in under 3.5 minutes, if the recovery bandwidth is not constrained. Once D resides on the LSTORs, the risk for losing data is eliminated.

The appealing property of SIMFAIL^s is that it replicates without accessing the network, thereby providing cost-effective fast recovery capabilities to storage systems that do not enjoy an expensive, all-powerful network with full bisection bandwidth.

Of course, SIMFAIL^s is not a full solution, because the risk still exists for some short period of time, which can be much longer if recovery bandwidth is constrained. SIMFAIL^s further does not provide any defense against occasional bad sectors that the system might encounter while recovering using the surviving replicas.

In order to provide a risk-free solution that is immune to occasional bad sectors, we configure SIMFAIL such that each LSTOR continuously maintains the parity (xor) of the superchunks on its corresponding disk (illustrated in Figure 2). Let us now assume that a double disk failure has occurred. Seemingly, such a failure means that we have lost the data residing in the "intersection" of the two failing disks, as there is no other replica in the system that holds it. But 1-sharing assures us that the lost intersection is comprised of only one superchunk. In addition, 1-mirroring assures us that all the other superchunks of the failing disks are still available elsewhere. Lastly, the LSTORs of the two failing disks are still accessible to us, so we also have the superchunk parity of the failing disks at our disposal. Consequently, by utilizing the surviving superchunks and the parity, we can easily reconstruct the lost superchunk and recover.

For example, suppose D_2 and D_3 in Figure 2 failed, then block e is "lost" because both of its replicas are gone. But e can be recomputed in either of two ways: (1) $L_2 \oplus b_1$ and (2) $L_3 \oplus d_0$. This flexibility gives SIMFAIL the ability to avoid hotspots during a reconstruction.

Notice that the full SIMFAIL solution helps in coping with occasional bad sectors as well. Assume that one disk D fails and the system is now busy in replicating D's risky superchunks so as to restore 1-mirroring. Further

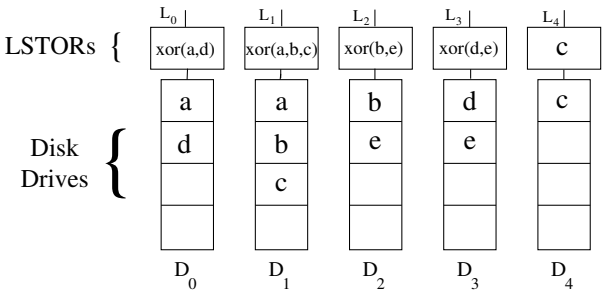

Figure 2: SIMFAIL *full solution. Each superchunk appears twice, no two disks share more than one superchunk, and every disk's superchunks are xor-ed in the* LSTOR.

assume that, during the replication, SIMFAIL encounters a bad sector in one of those risky superchunks. SIMFAIL can then utilize the parity information in the LSTOR to reconstruct the bad sector.

Moreover, assume that two disks fail and SIMFAIL encounters a bad sector while reconstructing their lost intersection superchunk. Since there are two failing disks, there are two different LSTORs and two different sets of risky superchunks to use in order to reconstruct the lost intersection. So if one set does not work due to a bad sector, SIMFAIL can utilize the other set.

5 Recovery and Load Balance in SIMFAIL

A consequence of all failures is that many chunks are left unmirrored. It is critical that these risky superchunks quickly be duplicated onto other disks, because the next disk failure may require a costly chunk reconstrunction or result in data loss. We refer to the set of disks tasked with transferring risky superchunks as *senders*.

We have two goals when duplicating superchunks in a recovery: maintain 1-sharing and minimize load imbalance between disks. 1-sharing must be maintained throughout a recovery because it ensures that any lost data in a double disk failure is recoverable using the LSTOR. Keeping disks load balanced prevents a situation where some disks may become hotspots after being on the receiving end of many superchunk transfers. Load balancing goes hand in hand with ensuring that all transfers happen in parallel in order to quicken the recovery process. Thus, no disk should be on the receiving end of more than one superchunk transfer per failure recovery.

Optimally, a recovery will match each sender with a receiving disk according to the above criteria. We initially frame the recovery process as a maximum matching problem between sender disks and receiver disks, for which *all* senders must be matched with a receiver disk. There are readily available algorithms that efficiently provide a solution for maximum matchings [9, 11].

Figure 3 depicts the failure of disk D_1, and the graph in Figure 4 depicts each sender disk (left) with an edge

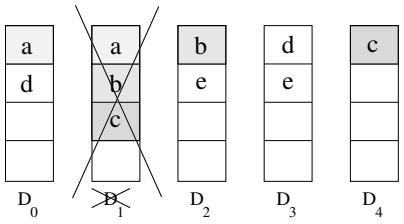

Figure 3: *Disk D_1 fails in a 5 disk array. To recover, disks D_0, D_2, and D_4 must duplicate the now risky superchunks that they shared with D_1.*

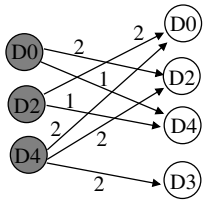

Figure 4: *Senders (left) have edges to the disks which do not share a block with the sender (right). Each receiving disk on the right side may only receive one risky superchunk per recovery.*

to each receiving disk that it can be matched with (right), for now disregarding the numerical value on each edge.

This formulation provides that all senders will be matched, and no receiver will be matched with more than one sender. Unfortunately, a basic matching algorithm might provide a matching such that D_0 sends a chunk to D_2, and D_2 sends a chunk to D_0. Such a matching is unacceptable because D_0 and D_2 would violate 1-sharing. This example is intended to illustrate that the assignment is a nontrivial task due to 1-sharing. Another shortcoming of the current formulation is that it does not take the load on a disk into account, which means lightly loaded disks may be neglected in favor of heavily loaded ones in a matching. Such a recovery is sub-optimal.

We amend the formulation to the one pictured in Figure 4, where we assign costs to edges according to the amount of load on disk, and apply a "minimum-cost" matching algorithm that finds the smallest total cost for the assignment [14]. In addition, we use a dynamic algorithm which allows us to remove edges (and update costs) after each assignment. Mills-Tettey et al. provide a dynamic version of the Hungarian Algorithm that achieves an optimal matching [15]. Due to space constraints, we avoid elaborating on this algorithm further.

5.1 Proactive Load Balance

When there is a load difference greater than two superchunks between the heaviest and lightest loaded disks, the system can be load balanced. Superchunks can be transferred from heavier disks to lighter disks, provided

that 1-sharing and 1-mirroring are maintained. result of the transfer.

1-sharing is at risk of being violated when a heavily loaded disk has a superchunk which is mirrored on a disk that the lighter disk already shares with. Transferring such a superchunk to the lighter disk results in the lighter disk sharing *two* superchunks with another disk in the system. Fortunately, such a transfer is avoidable because the heavier disk has *at least* 2 more superchunks than the lighter disk; thus there are at least 2 superchunks which do not violate 1-sharing upon transfer. The final restriction is that if the heavier disk mirrors one of the lighter disk's superchunks, the heavier disk must not send the shared superchunk. Otherwise, both copies will reside on the same disk, violating 1-mirroring. These principles are directly applied when adding a disk to the system.

5.2 Internal Fragmentation

As data gets erased, unused data regions form within superchunks. If this space is neglected, it impacts the performance of the system in terms of computing XORs and chunk transferrs, and also reduces the usable storage.

Fragmentation can be reduced by periodic data scrubbing and realignment in affected chunks, or maintaining a table of fragmented data segments that can be written to during subsequent writes to the system.

6 Trade-offs and Overheads

Performance vs. Safety – Maintaining parity information on each LSTOR is costly. Each write to a disk requires reading the old data prior to writing the new data to keep the LSTOR parity up to date.

We set out to investigate the performance overheads of adding an additional read before each write, and ran several tests on a 5400 RPM 1 TB Samsung HD103S1 disk. We also ran a *Sysbench* test on random reads (9 ms) and writes (22 ms) to sanity-check our test results.

We performed sets of 2000 requests on blocks of 512 bytes to simulate reading a 1 MB file. We found that on average, each write of a 512 byte block took 18 ms, and preceding it with a read to the same block added another 6 ms (33% overhead). We made sure to disable any hardware or software optimizations that could minimize the read overhead by disabling disk caching, read lookahead, and performing an *fdatasync* after every write.

The I/O overheads associated with keeping parity up to date are well-documented as the "small write problem" [4]. Encouragingly, there is considerable prior work on solving the small write problem and minimizing performance degradation due to parity [5, 13, 21, 23]. This work can be applied to how parity data is written to LSTORs. Two such examples are delaying parity updates until there is a lull in other contending workloads [23] and logging small parity updates until they can be batched into more efficient large accesses [21].

Replication vs. Erasure Coding – Thus far, this research has only considered replicated storage systems when evaluating SIMFAIL. However, erasure coding can provide resilience to more simultaneous failures with a much smaller storage overhead. For example, the Reed-Solomon (6,3) erasure code splits a block of data into 6 uniformly-sized blocks and computes 3 additional blocks for parity. The original 6 data blocks can be reconstructed using *any* 6 of the 9 total blocks. As a result, Reed-Solomon (6,3) tolerates up to 3 failures while only incurring a 50% storage overhead [12, 19].

Despite the advantages in storage efficiency and failure tolerance, erasure codes are not the primary method of storing newly written data in systems such as Windows Azure and GFS. Both systems replicate data initially. Windows Azure encodes data only after it is in a read-only state [1, 12], and GFS specifically mentions read-only data as a candidate for erasure coding [10].

Avoiding mutable data for erasure codes can mitigate costly overheads such as network bandwidth and computation to maintain updated parity blocks. Fan et al. discuss the overheads surrounding parity updates for small writes in DiskReduce, another system which replicates data prior to encoding it [7]. Also, reconstructing lost blocks may require relocating the requisite data and parity blocks to the same node. This requirement can adversely affect the network and reduce storage efficiency [20].

Replicated systems have the added advantage of requiring fewer failure domains when arranging the data in a manner that prevents data loss from correlated failures. For example, storing 3 replicas of an object in different racks or datacenters requires fewer failure domains than distributing the 9 blocks associated with a Reed-Solomon (6,3) erasure code across the same level of storage granularity [8]. Indeed, the erasure codes available in Windows Azure are limited by the number of failure domains available in the clusters [12].

Additionally, replicating data allows for immediate hotspot avoidance by directing traffic to a storage node experiencing less load. In contrast, avoiding a hotspot in an erasure coded system may first require a resource-intensive reconstruction [12].

SIMFAIL encompasses advantages from both replicated and erasure coded storage. Hotspot avoidance is achieved by storing two replicas in different failure domains. The network overhead required to maintain parity data is eliminated by each LSTOR's proximity to the disk for which it is storing parity data. Finally, the LSTOR enables improved storage efficency versus 3-replication because it allows SIMFAIL to recover from two simultaneous disk failures.

Recovery vs. Application I/O – Optimally, a superchunk can be written to an LSTOR in the time it takes to read it from the disk at peak bandwith. However, reconstruction throttling is a common practice during reconstruction because monopolizing disk bandwidth negatively impacts other workloads [26].

Conversely, throttling bandwidth when there are no competing workloads needlessly extends the vulnerability window. Thus, one of our challenges is throttling the recovery bandwidth no more beyond what is needed to avoid adversely impacting other disk requests.

Capacity vs. 1-sharing –SIMFAIL's current design allows each disk to mirror one superchunk from the other $N-1$ disks in the system. However, after a disk failure each disk will hold an extra superchunk due to still having $N-1$ allocated superchunks, despite only $N-2$ other remaining disks in the system.

Disks can instead be split up into $N-k$ chunks, where $k>1$. Such a layout results in the same size disk being split up into fewer superchunks of greater size. Consequently, for each disk there are at least $k-1$ disks that the disk cannot share a block with. While this is a better use of disk space, it has the trade-off of requiring more storage on the LSTOR and larger block transfers during recovery.

Failure Domains – When an N-disk SIMFAIL array is split up into $N-1$ data blocks, every disk can share with every other disk in the system. This increases the chance that multiple failed disks will have overlapping replicas.

Alternatively, the $N-k$ layout discussed above lowers the risk that simultaneous failures will result in data loss, because each disk has no overlap with $k-1$ other disks.

We have assumed that LSTORs are accessible after a disk failure. In reality, LSTORs and their disks are in the same failure domain, and this is indeed a drawback of SIMFAIL. Still, the severity of this issue is lessened by there being two LSTORs, potentially in different failure domains, which can be used to recover.

Constructing LSTORs– Prototyping the LSTOR remains a major challenge. Each LSTOR will act as a proxy to and from the disk it is attached to, which means it requires a I/O protocol-aware controller (eg. SATA).

There are trade-offs between the memory options for the LSTOR. DRAM, though fast, is expensive and is sensitive to fluctuations in power due to its volatility. Flash memory is cheaper, but deteriorates over time [25].

One alternative to constructing an LSTOR is to repurpose a data block on each disk to store parity for a different disk, eliminating the need for new hardware. However, using a second disk adversely affects throughput because it imposes a burden on the network and requires an additional disk's participation for each write request.

7 Conclusions

SIMFAIL is a performant storage solution that offers fast recovery, tolerates simultaneous failures, and provides cost savings versus other systems. The introduction of the LSTOR allows for significantly reduced data transfer times and communication overheads. We are optimistic that our design can bring about a paradigm shift in how failure tolerance is built into data centers. Our continued research will show the system's benefits and attempt to resolve the open issues outlined throughout this paper.

References

[1] B. CALDER ET. AL. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *ACM Symp. on Operating Syst. Principles (SOSP)* (2011), pp. 143–157.

[2] BORTHAKUR, D. *HDFS architecture guide*. The Apache Software Foundation, 2008.

[3] BURKE, B. A. Something you should know (about XIV). `http://thestorageanarchist.typepad.com/weblog/2008/09/1025-something.html`, Sep 2008. Blog post by Chief Strategy Officer for the Symmetrix & Virtualization Product Group at EMC Information Infrastructure Products.

[4] CHEN, P. M., LEE, E. K., GIBSON, G. A., KATZ, R. H., AND PATTERSON, D. A. Raid: high-performance, reliable secondary storage. *ACM Comput. Surv. 26*, 2 (June 1994), 145–185.

[5] DHOLAKIA, A., ELEFTHERIOU, E., HU, X.-Y., ILIADIS, I., MENON, J., AND RAO, K. A new intra-disk redundancy scheme for high-reliability raid storage systems in the presence of unrecoverable errors. *Trans. Storage 4*, 1 (May 2008), 1:1–1:42.

[6] DUFRASNE, B., PARK, I. K., PERILLO, F., SAUTTER, H., SOLEWIN, S., AND VATTATHIL, A. *IBM XIV Storage System Gen3 Architecture, Implementation, and Usage*, 5 ed. IBM Redbooks, Jul 2012. `http://www.redbooks.ibm.com/abstracts/sg247659.html` (Accessed: Jan 2013).

[7] FAN, B., TANTISIRIROJ, W., XIAO, L., AND GIBSON, G. Diskreduce: Raid for data-intensive scalable computing. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage* (2009), ACM, pp. 6–10.

[8] FORD, D., LABELLE, F., POPOVICI, F. I., STOKELY, M., TRUONG, V.-A., BARROSO, L., GRIMES, C., AND QUINLAN, S. Availability in globally distributed storage systems. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation* (2010), USENIX Association, pp. 1–7.

[9] FORD JR, L., AND FULKERSON, D. Maximal flow through a network. *Canadian Journal of Mathematics 8* (1956), 399–404.

[10] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. In *ACM Symp. on Operating Syst. Principles (SOSP)* (2003), pp. 29–43.

[11] HOPCROFT, J., AND KARP, R. An nˆ5/2 algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing 2*, 4 (1973), 225–231.

[12] HUANG, C., SIMITCI, H., XU, Y., OGUS, A., CALDER, B., GOPALAN, P., LI, J., YEKHANIN, S., ET AL. Erasure coding in windows azure storage. In *USENIX conference on Annual Technical Conference, USENIX ATC* (2012).

[13] HWANG, K., JIN, H., AND HO, R. Raid-x: A new distributed disk array for i/o-centric cluster computing. In *Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing* (Washington, DC, USA, 2000), HPDC '00, IEEE Computer Society, pp. 279–286.

[14] KUHN, H. The hungarian method for the assignment problem. *Naval research logistics quarterly 2*, 1-2 (2006), 83–97.

[15] MILLS-TETTY, G. A., STENTZ, A. T., AND DIAS, M. B. The dynamic hungarian algorithm for the assignment problem with changing costs. Tech. Rep. CMU-RI-TR-07-27, Robotics Institute, Pittsburgh, PA, July 2007.

[16] NIGHTINGALE, E. B., ELSON, J., FAN, J., HOFMANN, O., HOWELL, J., AND SUZUE, Y. Flat datacenter storage. In *USENIX Symp. on Operating Syst. Design & Implementation (OSDI)* (2012), pp. 1–15.

[17] OpenStack open cloud computing software. `http://www.openstack.org`. (Accessed: Jan 2013).

[18] PINHEIRO, E., WEBER, W.-D., AND BARROSO, L. A. Failure trends in a large disk drive population. In *USENIX Conf. on File & Storage Technologies (FAST)* (2007), pp. 17–28.

[19] REED, I. S., AND SOLOMON, G. Polynomial codes over certain finite fields. *Journal of the Society for Industrial & Applied Mathematics 8*, 2 (1960), 300–304.

[20] RODRIGUES, R., AND LISKOV, B. High availability in dhts: Erasure coding vs. replication. In *Peer-to-Peer Systems IV*. Springer, 2005, pp. 226–239.

[21] SAVAGE, S., AND WILKES, J. Afraid: a frequently redundant array of independent disks. In *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 1996), ATEC '96, USENIX Association.

[22] SCHROEDER, B., AND GIBSON, G. A. Disk failures in the real world: what does an MTTF of 1,000,000 hours mean to you? In *USENIX Conf. on File & Storage Technologies (FAST)* (2007).

[23] STODOLSKY, D., GIBSON, G., AND HOLLAND, M. Parity logging overcoming the small write problem in redundant disk arrays. In *Proceedings, the 20th annual International Symposium on Computer Architecture: May 16-19, 1993, San Diego, California* (1993), IEEE, p. 64.

[24] Swift 1.7.6-dev documentation – Swift architectural overview. OpenStack, LLC `http://docs.openstack.org/developer/swift/overview_architecture.html`. (Accessed: Jan 2013).

[25] THATCHER, J., COUGHLIN, T., HANDY, J., AND EKKER, N. Nand flash solid state storage for the enterprise: An in-depth look at reliability. *Solid State Storage Initiative (SNIA)* (2009).

[26] WU, S., JIANG, H., FENG, D., TIAN, L., AND MAO, B. Workout: I/o workload outsourcing for boosting raid reconstruction performance. In *Proccedings of the 7th conference on File and storage technologies* (Berkeley, CA, USA, 2009), FAST '09, USENIX Association, pp. 239–252.