RAIDP: ReplicAtion with Intra-Disk Parity

Eitan Rosenfeld
Technion – Israel Institute of Technology

Aviad Zuck
Technion – Israel Institute of Technology

Nadav Amit
VMware Research

Michael Factor
IBM Research

Dan Tsafrir
Technion – Israel Institute of Technology & VMware Research

Abstract

Distributed storage systems often triplicate data to reduce the risk of permanent data loss, thereby tolerating at least two simultaneous disk failures at the price of 2/3 of the capacity. To reduce this price, some systems utilize erasure coding. But this optimization is usually only applied to cold data, because erasure coding might hinder performance for warm data.

We propose RAIDP—a new point in the distributed storage design space between replication and erasure coding. RAIDP maintains only two replicas, rather than three or more. It increases durability by utilizing small disk "add-ons" for storing intra-disk erasure codes that are local to the server but fail independently from the disk. By carefully laying out the data, the add-ons allow RAIDP to recover from simultaneous disk failures (add-ons can be stacked to withstand an arbitrary number of failures). RAIDP retains much of the benefits of replication, trading off some performance and availability for substantially reduced storage requirements, networking overheads, and their related costs. We implement RAIDP in HDFS, which triplicates by default. We show that baseline RAIDP achieves performance close to that of HDFS with only two replicas, and performs within 21% of the default triplicating HDFS with an update-oriented variant, while halving the storage and networking overheads and providing similar durability.

CCS Concepts. • **Hardware → External storage**; • **Software and its engineering → Operating systems**.

1 Introduction

Due to the scale of modern datacenters and the non-negligible likelihood of failures, datacenters must employ safeguards to prevent data loss and preserve responsive data availability. Preventing loss necessitates that data can be *eventually* recovered after a failure. Responsive data availability, on the other hand, implies that data can quickly be accessed at will, suggesting that no reconstruction is required after a failure or that such reconstruction is transparent and unnoticeable in terms of performance to a user [17, 25, 45].

In today's distributed storage systems, data is typically stored in a *declustered* fashion using replicas or erasure coded data with parity information. Each logical unit of data is spread across many servers, typically chosen at random [19]. Upon a failure, modern systems reconstruct the data using the declustered chunks, in parallel, for a fast recovery [9, 16, 30, 37, 56].

Distributed storage systems overwhelmingly favor the use of replication over erasure coding when storing warm data [11, 14, 23, 28, 56, 70] for the following reasons. Replication is preferable for *reads* because: (**1**) it allows for load balancing, such that if one node is heavily loaded, the desired data can be retrieved from another node; (**2**) it can accelerate reads by utilizing several replicas in parallel; and (**3**) it avoids the problem of "degraded reads" [23, 37, 41] whereby the desired data needs to be reconstructed due to residing on an unavailable (e.g., rebooting) node, inducing many more I/O operations.

Replication is preferable for synchronous *writes*, because (**4**) it may reduce latency by not needing to wait for a full stripe to accumulate before computing parities.

Replication is preferable to erasure coding for both *reads and writes* because: (**5**) it avoids the CPU processing of encoding the data and decoding it upon a reconstruction; (**6**) it may generate fewer random seeks as it can sequentially write (and later read) a small stripe rather than partition it to even smaller fragments designated to different drives; and, importantly, (**7**) it induces substantially less networking traffic overhead for recovery, because replication recovery involves only one node that holds a replica, whereas erasure recovery involves many nodes associated with the encoding which dramatically interferes with and reduces the available bandwidth for foreground jobs [55, 64].

Alongside the advantages of replication, it has one serious deficiency: high storage overhead (a replication factor of k results in $\frac{k-1}{k}$ overall "wasted" capacity). This drawback has

EuroSys '20, April 27–30, 2020, Heraklion, Greece
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-6882-7/20/04…$15.00
https://doi.org/10.1145/3342195.3387546

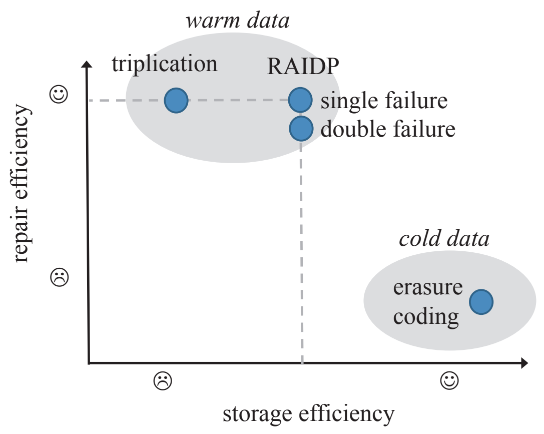


Figure 1. *Triplication has low repair traffic, but it wastes capacity. Erasure coding is more storage efficient but has much higher repair traffic. For this and additional reasons, triplication is suitable for warm data and erasure coding is not. RAIDP is a middle point enjoying most of the benefits of triplication while being more storage efficient.*

brought about the use of replication for warm data in tandem with erasure coding for cold, read-only data [11, 15, 23, 37, 55, 64, 82]. For example, in Ceph [80], "erasure-coded pools [are used for] cold storage with high latency and slow access time" [15]. Such designs allow erasure coding to occur in the background, which increases sequentiality because blocks that comprise an erasure stripe can be set to a large size. As cold data is accessed infrequently, it can tolerate the reduced responsiveness that can occur with erasure coding, e.g., due to degraded reads or a slow device in the stripe.

The state of the art is to provision a datacenter using both redundancy schemes, leaving data in one of two extremes: either the system is performant, induces little repair traffic, but is wasteful in terms of storage or it is subject to large response delays, induces significant repair traffic, but is efficient in terms of storage.

We introduce a new distributed redundancy scheme called RAIDP, which stands for ReplicAtion with Intra-Disk Parity. RAIDP is a new point in the design space of distributed storage systems. It is a hybrid system combining replication with "local" erasure coding, deemphasizing the weaknesses of the two redundancy schemes at the expense of some of their strengths. RAIDP is a middle ground design that is applicable to warm data—largely enjoying the aforementioned seven advantages of replication, providing comparable resiliency, while substantially reducing its overheads. Fig. 1 illustrates how RAIDP relates to both replication and erasure coding.

RAIDP is characterized by three key traits. First, it uses two replicas instead of the typical three (or more) [11, 14, 28, 41, 56]. This potentially reduces much of the storage and networking overheads of triplication-based systems. Consequently, RAIDP can also save upto 33% in hardware (disks, servers enclosing the disks, switches, etc.), power, and facility costs of data centers. RAIDP is intended as a solution for large distributed systems with thousands of disks. Notably, at hyper-scale, even saving a small 1% of these costs can have significant financial implications [31, 34].

Second, RAIDP subdivides each disk into logically contiguous "superchunks", which are uniformly-sized in the order of a few GBs. A superchunk holds the shared content between two disks and RAIDP lays out data such that every two disks share at most one superchunk.

The third RAIDP trait is that it relies on disk "add-ons", which are small auxiliary storage devices attached to each disk. We denote these devices as "local storage devices" or Lstors. An Lstor fails separately from its disk, is much faster/smaller, and is persistent. An Lstor can be implemented using a form of NVRAM, whose independence is commonly utilized [3, 21, 51, 81].

RAIDP uses Lstors to save the need for an additional third replica. In its simplest configuration, RAIDP associates each disk with its own Lstor, which allows the system to tolerate dual disk failures without losing data. Generally, RAIDP may associate $k \geq 1$ Lstors per disk, allowing the system to survive $k + 1$ simultaneous failures.

An Lstor stores a non-rotated erasure code [41] of its disk's superchunks. Since this information is completely local to the node containing the disk, Lstors do not burden the network to be kept up to date. When two disks D_1 and D_2 fail simultaneously, the properties of the RAIDP layout dictate that: (1) the only data that is lost is their shared superchunk, and (2) this data is easily recoverable with the help of either of the two disk's Lstor and surviving superchunks, which are replicated elsewhere. Fig. 2 illustrates a simple example for a legal RAIDP configuration using a single Lstor for every disk. We elaborate on this example later in the paper. We compare RAIDP to other storage system configurations in §2 and describe its design in §3. We analyze the economical feasibility of RAIDP in a datacenter environment in §4.

We implement RAIDP in the Hadoop Distributed File System (HDFS) by superimposing on it the superchunk layout, supplementing each disk with a simulated Lstor, and incorporating a journal for crash consistency. We describe our implementation in §5. Our experimental evaluation in §6 confirms that the network and storage overheads are halved as compared to the default (triplicating) HDFS. The cost is up to 21% runtime degradation on update-intensive workloads, largely due to the additional disk operations required to maintain the parity on the Lstors. We additionally demonstrate that RAIDP successfully recovers from dual disk failures. We survey related work in §7, and discuss future work and conclude in §8.

2 Why Replication is Hard to Avoid

Table 1 lists the advantages and disadvantages of storage system configurations that can withstand two simultaneous failures, including RAIDP. In this section, we compare

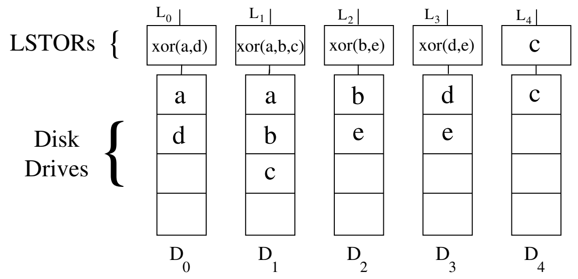


Figure 2. *Illustrating RAIDP. Each superchunk appears twice, no two disks share more than one superchunk, and every disk's superchunks are XORed in the Lstor.*

properties	*3 replicas*	*erasure n+2*	*raidp*
storage capacity	-	+	±
read	+	-	±
parallelism			
load balancing			
degraded read (temp. unavailable data)			
cpu consumption (sync latency)	+	-	±
disk sequentiality	+	-	+
write—network			
sub-stripe (small write)	-	-	+
full stripe (large write)	-	+	±
write—disk			
sub-sector (small write)	-	-	±
sub-block (medium write)	+	-	±
multi-block (large write)	±	+	-
repair traffic (network and disk)			
single failure	+	-	+
dual failure	+	-	±
failure domain tolerance	+	+	±

Table 1. *Comparing RAIDP to triplication and erasure coding with two parities—all tolerate double disk failures. (Notation: "+", "-", and "±" mean "best", "worst", and "in between".) RAIDP improves upon the worst offending system in all but the two* **bolded** *cases. In the first case (disk multi-block write), RAIDP becomes superior to replication with higher levels of failure tolerance, leaving the second (failure domain tolerance) as the only remaining disadvantage.*

these systems, noting that despite its high storage overhead, replication has key advantages that make it the redundancy scheme of choice for warm data in modern distributed systems [6, 11, 14, 23, 28, 41, 56, 70].

Capacity. With distributed replication, k copies of each datum are made and stored across servers.

With distributed erasure coding, a piece of data is divided into n uniformly-sized blocks plus an additional k parity blocks, together comprising an $n + k$ "stripe." Each of the original n blocks can be recovered so long as at least n of the $n + k$ stripe blocks are available [37, 66]. Typically $n > k$, so erasure coding provides failure tolerance with a smaller storage overhead as compared to replication. Many different erasure codes have been suggested to safely store data on a distributed system with various trade-offs [37, 48, 49, 63, 70]. Therefore, without loss of generality, we focus on systematic codes such as the popular Reed-Solomon codes [66].

RAIDP stores two replicas plus a local erasure code in order to achieve the required failure tolerance. Thus, RAIDP is more storage-efficient than triplication, but worse than erasure coding.

In our discussion, we assume that the erasure codes are "systematic", which means that a stripe is comprised of data blocks that can be directly read and of parity blocks that are used to recover lost data [5]. The prevalent Reed-Solomon is an example of such a code [66]. In "non-systematic" codes, all data accesses require decoding, and are hence slower [58]. The payoff is that in many cases reconstruction induces less disk/network recovery traffic than systematic codes. Although we limit the discussion to former codes, much of the discussions also apply to the latter (barring additional write and CPU overheads). For erasure coding systems, we also assume maximum distance separable (MDS) codes, such as Reed-Solomon, in the discussion below. With MDS codes, any n devices can be used to recover the lost data [48]. While the discussion below also applies to non-MDS codes, the exact parameters we use apply to MDS codes.

Ability to Load Balance & Parallelize. Replication is well-suited for reads because traffic can be redirected away from highly trafficked or temporarily unavailable nodes [11, 46]. Reads in a replicated system can also leverage the multiple copies to read several parts of a file in parallel, potentially providing a significant boost in read bandwidth [38]. When data is encoded with a typical erasure code, by contrast, only one copy of each datum is stored, plus the means to reconstruct the data. While the blocks from an erasure coded stripe in theory can be read in parallel, the system's response time will be constrained by the slowest node. To avoid stressed or unavailable nodes, the system can perform a much slower "degraded read", burdening n nodes in the stripe with reading and transmitting n blocks [23, 37, 41].

Improved Sync Latency & Sequentiality. Distributed storage systems partition streams of incoming data to blocks of m bytes [9, 11, 56, 70]. Relatively large m values improve performance, because they promote sequentiality and reduce the number of random disk seeks. However, they might adversely affect the latency of synchronous writes for $n + k$ erasure coding systems; when the k parity values are computed over the incoming data in an online manner the system must wait for all the $n \cdot m$ bytes to arrive before finalizing the computation of the associated k parities and acknowledging the client's request. With replication, in contrast, every m-sized block can be immediately replicated and ack-ed [9, 10]. On the other hand, reducing the block size m reduces the sequentiality on disk for impending writes and future reads, and generates more metadata. RAIDP has the sequentiality of a replicated system, but incurs a small latency penalty due to the encoding process required to maintain local erasure codes.

Reduced CPU Consumption. Erasure coding additionally induces CPU overhead that is absent from replicating systems, because write operations consume cycles when encoding the parities associated with the stripe.

The CPU is further consumed when decoding missing data upon reconstruction via erasure codes, which may or may not occur when fulfilling a (degraded) read request [55]. Hence, systems reserve erasure coding for cold data and relegate the encoding to the background, whereby the aforementioned disadvantages are less of a concern [11, 23, 55, 70].

RAIDP requires CPU processing to update its dual local erasure codes with every write, incurring the same CPU overhead as fully-erasure coded systems with two parity blocks per stripe. However, parity updates are offloaded to dedicated hardware logic on the Lstor, and there is no additional CPU overhead for reconstruction when only one superchunk copy is unavailable.

Reduced Repair Traffic. The final—and arguably most important—advantage of replication over erasure coding concerns "repair traffic" – the disk and network I/O used to overcome disk failures. Consider the case of a single missing block, which according to Rashmi, et al., accounts for 98% of the observed failure scenarios in distributed storage systems [64]. Upon such a failure in a replicating system, the amount of data that needs to be read from the disks across the system, as well as transmitted over the network, is equivalent to the amount of data that was lost. In contrast, the repair traffic induced by an $n + k$ erasure coding system is n times greater [37, 70]. The importance of this drawback was highlighted by a significant body of recent research directed towards reducing the repair traffic [20, 37, 41, 53, 64, 70, 77, 78, 84].

Even in the event of a double failure, RAIDP improves over erasure coding for subsequent failures since only a fraction of the data on the failed drive – the shared superchunk – is recovered in an erasure coded fashion, as described in §3.

Writing. For brevity, we focus on large writes, which is one of only two problematic areas for RAIDP. As noted above, RAIDP uses local erasure codes to increase its failure tolerance. These erasure codes must remain in sync with the data for every incoming write. For deleted data parity calculations may be deferred to idle times (see §5). However, for update-intensive workloads the system must immediately update the erasure code for old data before it is overwritten, resulting in a read-modify-write sequence on each replicating node. In such cases, if a client wishes to write n blocks of data, then on each node n blocks of data are read and subsequently written to disk, for a total of $4n$ blocks of disk traffic in RAIDP.

When writing a full stripe, erasure coded systems can calculate the parity solely based upon the newly written data; with two parity blocks, this entails a total of $n + 2$ blocks of disk and network traffic. Triplication requires n blocks for every replica, resulting in $3n$.

Failure Domains. The other problematic area for RAIDP is failure domains. As discussed below in §3.2, RAIDP uses the erasure codes stored locally alongside disks in order to achieve a higher failure tolerance. Disk failures are the most probable cause of failures in modern datacenter servers (e.g., [76]). We assume that disks fail separately from the devices storing the erasure codes. However, in practice this assumption may not always hold. For example, a failure of an entire rack would render both a disk and its local erasure codes inaccessible. Thus, RAIDP is inferior to triplication and erasure coding systems in terms of *availability* because those systems can spread different parts of a (replicated or erasure coded) stripe over more than just two failure domains (e.g., multiple racks). However, RAIDP is on par with other systems in terms of *durability* since data and local erasure codes remain intact in RAIDP even when an entire rack fails. We discuss ways to increase RAIDP availability in §8.

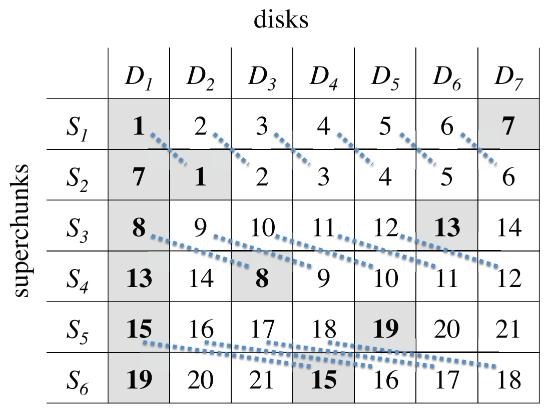
3 Design

We contend that distributed storage systems can benefit from a redundancy scheme that (1) remains applicable to warm data, (2) largely retains the advantages and failure tolerance of triplication, and (3) carries less of a storage footprint. We further contend that RAIDP meets these criteria. Next we describe the layout of RAIDP (§3.1) and its disk "add-ons" (§3.2), which, when combined, allow the system to recover from simultaneous failures. We then explain how RAIDP recovers after such failures (§3.3), and how it maintains crash consistency (§3.4).

3.1 Layout

RAIDP enforces a distributed data layout, such that when two disks fail simultaneously, the shared lost content is contiguous on disk. This layout is realized by first partitioning the disk into "superchunks", which are uniformly-sized data regions, on the order of several GBs. Each superchunk is bitwise mirrored on a different disk so that when the system writes into one superchunk, it also writes to the other in identical offsets in both disks. We call this property "1-mirroring". RAIDP also ensures that no two disks share more than one superchunk, so at most one superchunk is lost in the event of a dual failure. This latter property is called "1-sharing".

Construction. An illustration of one possible RAIDP data layout in a seven-disk system is depicted in Fig. 3. Observe for example how the superchunks on disk D_1 are replicated on the rest of the disks (highlighted by a shaded background). D_1 adheres to 1-mirroring and 1-sharing, as no two copies


disks
superchunks

	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

Figure 3. *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.*

of its superchunks exist on the same disk and all are replicated twice. In our example, a failure of any two disks would result in one lost superchunk. In general, a failure of two disks in RAIDP results in *at most* one lost superchunk, because the layout may include pairs of disks that do not share. Also, disks do not have to be identical as long as the layout maintains 1-sharing and 1-mirroring.

Arranging data across the system in a manner that preserves 1-sharing and 1-mirroring is easy and could be done in many ways. Fig. 3 illustrates one example. The dotted lines identify pairs of superchunk replicas. To preserve 1-mirroring every i-th row's superchunks are replicated in the $i + 1$ row for every odd i. To preserve 1-mirroring the replication shift between rows gradually increases: the replica shift between the first and second rows is one column, the shift between the third and fourth row is two columns, and, generally, the shift between the pair of rows $2i - 1$ and $2i$ is always i columns.

As in other distributed storage systems [10, 11, 14, 28, 56] replicas should be placed not just on different devices but also in different failure domains (e.g., servers, racks or rows). Location metadata for each relevant superchunk may be kept in dedicated master nodes (similarly to HDFS's NameNode).

Implications. In the seven-disk example in Fig. 3, no disk has more than six superchunks. This limit is due to 1-sharing, which dictates that two disks share at most one superchunk. Thus, with N disks, each disk will be comprised of at most $N - 1$ superchunks, limiting the number of superchunks in the system to at most $N \cdot (N - 1)$.

This limit also dictates the *minimal* size for a superchunk – for a disk of size S in a system with N disks the *minimal* size for a superchunk will be $\frac{S}{N-1}$. In a system with 1000 4TB disks this translates to $\approx$ 4GB per superchunk [1]. This limit also dictates that Lstors are a better fit in large systems, where they can be reasonably-sized (and economical).

Recovery following a disk failure involves replicating non-redundant superchunks to different disks in the system. Therefore, keeping the number of superchunks per disk less than the maximum is critical to recovery in RAIDP. Generally speaking, superchunks can be arranged to maintain 1-sharing and 1-mirroring after f failures, so long as there are at most $(N - f) \cdot (N - f - 1)$ superchunks to arrange. In large systems with thousands of disks finding enough disks that do not violate these constraints to allow recovery should not be difficult. The superchunks layout should leave enough free space in every node to allow the recovery process to take load balancing into consideration when re-arranging superchunks. Additional movement of superchunks may also be required during a recovery process, to construct a layout that enables recovery from future failures. We discuss one possible way to improve recovery time in §3.3.

More shared superchunks are lost as clusters inevitably experience multiple failures. To increase the likelihood of tolerating these failures, it is important that disks be replicated across racks so that if a single rack goes down, the availability of the other replica is preserved. Keeping the number of superchunks per disk less than the maximum is also beneficial for failure tolerance. If fewer superchunks reside on each disk, then fewer shared superchunks are lost during failures.

3.2 Disk Add-Ons (Lstors)

To be able to recover a superchunk following a dual disk failure, RAIDP associates each disk with a small disk "add-on", which stores parity information that is exclusively dependent on the local disk's content. We call these add-ons "local storage devices", or Lstors for short. The storage capacity of an Lstor is similar to that of one superchunk, with the addition of a small journal (see below). An Lstor stores an erasure code computed over all local superchunks in the associated disk. When necessary, this parity information is used to recover the lost data with the help of the mirroring nodes. We defer specifying the actual recovery procedure to §3.3. Here, we discuss the properties that Lstors should possess.

Lstor Properties. An Lstor is a small, simple device that has just enough computational power to process the I/O traffic that flows to/from the disk with which it is associated, and just enough memory to allow it to store one superchunk and a small journal to buffer data when processing incoming writes (discussed further in §3.4). Lstors interpose the I/O between their disk and its controller, as illustrated in Fig. 4 (though we acknowledge that it may be better situated in other locations in the storage architecture). Interposition can be logical or physical. We also assume the following properties:

[1]The actual size of a superchunk will also be influenced by the need to place the replicas in different failure domains

1. Disk failures that lead to loss of data occur separately from Lstor failures.
2. Parity information can be stored on the Lstor at least as quickly as data can be stored on the associated disk.
3. The Lstor parity and journal content are persistent.

Building Lstors. For Lstors to be practical, they need to be cheap. Their storage capacity (that is tightly correlated to their cost) can be reduced by making the superchunks smaller as explained above.

The Lstor's journal must be fast enough to incur negligible overhead on the corresponding disk I/O. An immediate solution would be to implement Lstors using battery-backed DRAM. The realistic example mentioned earlier requires 4GB of disk space per superchunk. Therefore, implementing the matching 4GB Lstor requires 4GB of flash, 4GB of DRAM, and a battery. Using an independent power source, such as a battery, may also assist in recovery following a power failure. Using this setup we can store updates in DRAM and persist the DRAM contents on the flash storage either periodically or following a power failure, effectively making the DRAM non-volatile. Another possible alternative is to employ some form of low-latency non-volatile memory [4]. However, such memories are typically slower than DRAM and are either not commercially available or only sold as part of a large device (e.g., 3D Xpoint). In this study we assume that it is possible to build Lstors (see §4), and we simulate them in DRAM as discussed in §5

Overhead of Using Lstors. The Lstors on each machine store parity data for the local disk's superchunks. For every incoming write to a superchunk, the parity must be updated. This update requires RAIDP to read the old data from the disk, compute the delta between the old and the new, and update the existing parity information accordingly, as dictated by the erasure code being used. This may result in a read-modify-write sequence which hurts performance in RAIDP due to the extra disk read. Lstor accesses, however, do not affect performance because (1) parity I/O is done in parallel with the disk accesses, and because (2) writes to the journal are performed at a high bandwidth relative to the disk.

Let us compare the total disk I/O operations conducted by RAIDP and a triplicating system, as both tolerate dual disk failures. Where triplication performs three writes, RAIDP performs two reads and two writes for a total of four I/Os. RAIDP could equalize the overhead by reading the old data once, and transmitting it to the mirroring node (thereby doubling network traffic). We decided, however, not to utilize this optimization so as to keep all parity calculations local and thus avoid synchronizing between replicas on the critical path, as well as to avoid doubling the network requirements when writing.

A write to a RAIDP superchunk may result in two slow seek operations on disk, making RAIDP prohibitively slow for latency-sensitive workloads. However, this predicament is relaxed if the disk write I/O is scheduled by the host immediately after its related read, so it only incurs reduced rotational delays. For throughput-sensitive workloads RAIDP results in 2 seeks and 4 I/Os, versus 3 seeks and 3 I/Os in a triplicating system. However, RAIDP must perform parity modifications for every read-modify-write sequence. Consequently, intervening I/O scheduling during lengthy parity modifications can introduce increased latency overheads. To mitigate, RAIDP can streamline parity calculations for already-read data to minimize the time period spent in the modify phase so that the following write can be more quickly served. Notably, during an update in RAIDP every replica disk performs two I/Os consecutively. In a triplicating system every replica disk performs only a single I/O. However, since seeks incur significantly higher delays than I/Os, overall RAIDP might actually improve performance in some cases versus triplication.

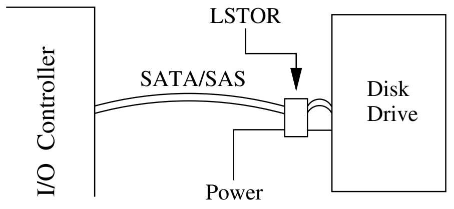


Figure 4. *Lstor interposes between a drive and its controller. The interposition can be physical (as shown in the figure) or logical, as long as the Lstor and disk fail separately.*

3.3 Recovery

To provide a recovery solution that withstands simultaneous disk failures, the Lstors local to a disk continuously maintain the up-to-date superchunk erasure codes of that disk. To explain how a RAIDP system survives double disk failures we now return to the illustrative example in Fig. 2. In this example, a single-Lstor layout is shown, where the erasure code on the Lstor is a simple XOR. Let us now assume that a double disk failure has occurred. Seemingly, such a failure means that we have lost the shared content of the two failing disks, as there is no other replica in the system. But 1-sharing ensures that this shared content is comprised of only one superchunk. In addition, 1-mirroring ensures 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 because Lstors and disks fail separately, so we also have the superchunk parity of the failing disks at our disposal. Consequently, we can reconstruct the lost superchunk and recover.

For example, suppose D_2 and D_3 in Fig. 2 fail, then block e is lost because both of its replicas are gone. But e can be recovered in two ways, either using Lstor L_2: $L_2 \oplus b_1$, or using Lstor L_3: $L_3 \oplus d_0$. This flexibility allows RAIDP to use a different Lstor if one is unavailable or if there are hotspots that are ideally avoided for the reconstruction. Or, the two

Lstors and sets of mirroring superchunks can be used to rebuild the "lost" superchunk in parallel, with each set used to rebuild half.

As mentioned in Section §3.1, in a RAIDP-based system each disk can be mirrored on a smaller subset of the total available disks. As a more concrete example, consider a RAIDP-based system with 10K 1TB disks, 12 disks per server and a superchunk size of 4GB. Each disk is split into 256 superchunks, and each superchunk is mirrored on another disk. Following a disk failure, this still leaves over 9K disks as potential candidates for the 256 non-redundant superchunks previously stored on the failed disk. Even if an entire server fails (meaning that access to all 12 of its disks is permanently lost) RAIDP needs to re-replicate only 3K superchunks to recover. Again, this leaves us with enough empty slots in disks that do no violate any of RAIDP's constraints. We note that with enough disks RAIDP can even preserve node-independence up to a degree. To wit, with 30K disks (and 2.5K servers) in the above example RAIDP can lay out recovered superchunks so that after recovery most servers will store only one superchunk from the failed server.

Improving recovery time. In most replication schemes blocks are allocated randomly or pseudo randomly [27, 56]. When blocks are small, random allocation does not severely impact the failure tolerance of a system, as disks still only share a small portion of their data with other disks and data is typically triplicated. In RAIDP, superchunks are much bigger and the system must maintain 1-sharing, severely restricting how data is organized. A naive allocation of superchunks and unwise assignment of superchunk mirrors after a failure can both burden the system and potentially prolong failure recovery. Additional failures during recovery will further exacerbate reconstruction overheads and may even cause data loss.

One of the challenges of RAIDP is how to optimize the recovery process after a failure. The system would strive to accomplish two goals when duplicating superchunks in a recovery: maintain 1-sharing and minimize load imbalance between disks. 1-sharing would have to be maintained throughout recovery because it ensures that any lost data in a double disk failure is recoverable using Lstor. Keeping disks load balanced would prevent a situation where some disks 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.

For the simple case of a single disk failure, disks storing non-redundant data after the failure would be tasked with transferring risky superchunks. We refer to these disks as *senders*. Optimally, a recovery would match each sender with a receiving disk according to the above criteria, namely

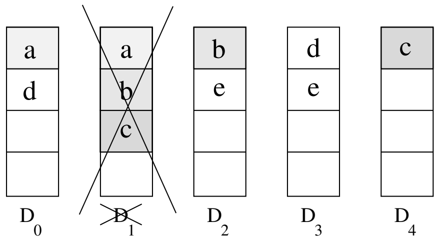


Figure 5. *Disk D_1 fails in a 5 disk array. To recover, disks D_0, D_2, and D_4 must duplicate the now risky superchunks that they shared with D_1.*

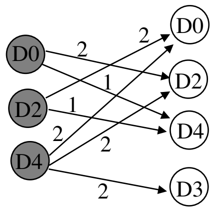


Figure 6. *Senders (left) have edges to the disks that do not share a block with the receiver (right). Each receiving disk on the right side may only receive one risky superchunk per recovery. Edges quantify the load on disks.*

where 1-sharing is maintained and no disk receives two superchunks, the latter criterion ensuring parallelism. We could 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 [26, 36].

Fig. 5 depicts the failure of disk D_1 in the RAIDP setup from Fig. 2. We provide an initial formulation for recovery as a matching problem under this setup in Fig. 6. Each *sender* disk (left) has an edge to each receiving disk that it can be matched with (right), for now disregarding the numerical value on each edge.

Finding a matching solution based on this formulation further requires that all senders could be matched, and no receiver will be matched with more than one sender. A naive 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 could amend the formulation by assigning 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 [44]. We could use a dynamic algorithm that allows us to remove edges and update costs after each assignment. Edge removal could be used to prevent a situation like the previous example where one sharing is violated, and cost updates could be used to weigh the assignments in favor of disks that contain fewer superchunks. Mills-Tettey et al. provide a dynamic version of the Hungarian Algorithm that could be used for such a formulation [52]. We plan to explore and compare optimal recovery-friendly allocation schemes in future versions of RAIDP.

Stacking Lstors. RAIDP can straightforwardly survive additional failures by stacking multiple Lstors per disk, such that every disk is associated with $k > 1$ Lstors, rather than just one (each with its own, separate, power source). Additional Lstors on each disk enable the storage of more local parities, which can be used to recover from more simultaneous failures. The required capacity per Lstor remains as that of a single superchunk, regardless of the number of Lstors per disk. Alternatively, by using ideas or constructions similar to Reed-Solomon codes [49, 61] RAIDP can recover from $k + 2$ simultaneous failures using only k Lstors on each disk (e.g. use a single Lstor on each disk and recover from three simultaneous failures). We leave the implementation of a RAIDP recovery scheme using these codes as future work.

3.4 Crash Consistency

Lstors provide us with the ability to recover a lost superchunk after a simultaneous failure. Consequently, the parity data on a node with a failed disk is largely only called upon during a superchunk reconstruction. The challenge that we need to address is that the Lstor parity reflects the erasure code of its own *local* superchunks, whereas it is used for recovery in conjunction with corresponding surviving *remote* superchunks that may have been updated prior to detecting the disks' failures.

Therefore, from the instance a simultaneous disk failure is detected until the recovery process completes, we divert writes away from the superchunks on the failed disks. Reading is handled similar to erasure coded systems, but the scope of impact is substantially smaller due to the fact we only reconstruct a single superchunk rather than an entire disk.

Next, the Lstor used for recovery must be synchronized with its remote superchunks. These can fall out of sync in the event of transient failures, power outages, and disk failures. To synchronize the Lstor and superchunks, RAIDP utilizes a simple, append-only journal, which resides on the Lstor and is used to roll the Lstor forward. The roll-forward procedure is inspired by the canonical crash consistency protocols [54, 59, 62], where the journal re-establishes consistency between parities and remote supechunks.

The journal is stored in high-bandwidth memory on the Lstor. Local journal entries, created on each node performing the replicated write, mainly contain the new disk data, the old disk data, and newly computed parity. To ensure consistency, RAIDP writes a journal record to the Lstor for every incoming write. Once the journal record is synced, RAIDP can commit the incoming write to disk. After the journal record and write are synced, an acknowledgment is sent to the corresponding remote Lstor. Upon receipt of acknowledgment from the remote Lstor, the entire write is deemed successful and the local journal record is cleared.

If journal records are not cleared, then a component is failing on either the local node or the remote node. In such cases, to re-establish cluster wide consistency between remotely located superchunks and/or parities, the system transfers unresolved journal records and replays the write requests that they represent. For a double disk failure as soon as the failure is detected, the system stops sending new writes to the superchunks in the recovery series. In our experiments (see §6) we observed that journal acknowledgments arrive very quickly with at most one or two outstanding journal records residing in the journal at a time. This behavior allows us to keep the journal small (e.g., 128MB) under normal operating conditions. We describe our full solution in [68].

4 Feasibility and Cost

RAIDP's fundamental trade-off is the ability to trade a third replica, typically stored on disk, for two small disk add-ons on the remaining two replicas. In this section we explain why this trade-off is feasible and economical, especially for a large datacenter setup.

An Lstor is based on flash and enough RAM to maintain parity and a journal of writes to the device. The combined cost of 4GB of flash and 4GB of DRAM is \$9 [1] (all costs as of December 2019). Lstors also require a micro-controller to maintain independence from the local machine and disk. Several relatively powerful micro-controllers are currently available for as little as \$5 (e.g., [2]). During a disk failure we also require several hundred amperes for 2–3 minutes, to read data from the Lstor, easily obtainable from a small supercapacitor. Physically, Lstor components should fit into a small SATA-to-USB converter [7]. In comparison even a modestly-priced commodity 2.5" 4TB disk for storing an additional replica costs \$100 [8], 66% more than a conservative estimate of the cost of two Lstors.

However, the total cost of ownership (TCO) is much more than the direct purchase costs of the disks. We envision RAIDP as a solution for storing warm data in a datacenter environment, comprised of thousands of servers that manage dozens of disks each. Therefore, the associated costs of datacenter storage capacity needs to include the costs of servers as well as their operational costs. These costs scale more or less linearly with the number of disks, meaning that up to a 1/3 of the TCO may be saved by decreasing the replication factor from three to two.

To exemplify the server costs attached to disks using real world examples, we use two possible server configurations.

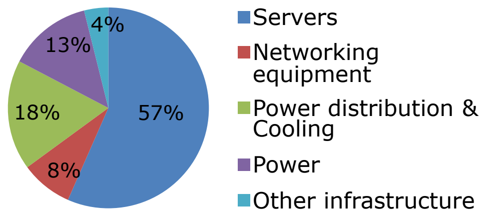


Figure 7. An example datacenter cost analysis by Amazon [33]. Additional costs, other than the acquisition of servers, comprise almost half of overall costs over time.

The first is a high-end Hyper Converged storage server recently purchased by a large academic institution (details omitted for commercial reasons). The total cost of the server is 20K$ and its primary storage devices are six 2TB hard drives. To estimate the hard drives net contribution to the overall cost we use the Dell website [79] as a publicly available source for server component pricing, and chose a representative 2TB 7.2K RPM SATA 3.5" hard drive (priced at $302). Following talks with purchasing administrators, we also assume that large-scale clients receive a significant discount over list prices, e.g. 50%, setting the effective disk price at ≈150$. We conclude that attached server costs can increase a drive's derived costs to $\frac{20K\$-6\cdot150\$}{6} \approx 3K\$$, significantly more than the drive's net cost. The second use case is a simpler SuperMicro 6048R-E1CR72L storage server configuration [74], with 72 7.2K RPM 2TB disks, and 500GB of RAM, priced at 23K$ out of which 9K$ are attributed to disks. Therefore, even in this relatively low-end setup the derived disk cost almost triples the disk's direct cost.

Other significant indirect costs, beyond the acquisition of servers to house the disks, include power consumption, labour, physical space, and networking. Fig. 7 shows one typical estimate from Amazon [33], which illustrates that over time (e.g., three years) these expenses constitute 43% of the datacenter TCO, a number repeated in other estimates (e.g., [13]). More servers take up more physical space, consume more power, need more manual labor for maintenance, etc. Therefore, it is likely that these costs also scale proportionally with the number of disks in the datacenter, nearly doubling the derived costs of using more disks.

In this work we simulate Lstors in RAM. However, as previously explained, a real Lstor would occupy significantly less physical space than a typical drive. Therefore, maintaining an Lstor in addition to every drive in a server would not change the physical space requirements dramatically. An Lstor also performs its parity calculations independently, and does not consume additional computational resources from the host. Finally, an Lstor contains several GBs of DRAM. Therefore, attaching an Lstor to every disk significantly increases the amount of DRAM in the system. However, the added power consumption is minor since most of the power is utilized by other server components and datacenter systems [39, 85].

In conclusion, trading Lstors for a drive is a significant improvement both in terms of direct purchase costs, as well as derived costs. As a result, replacing a third replica for two Lstors brings us close to the upper limit of 33% savings in datacenter TCO.

5 Implementation

We implement RAIDP within HDFS with two replicas. We selected HDFS because it is open source, packaged with standard benchmarks, and well-documented in past systems research.

The RAIDP implementation uses a single Lstor per disk simulated in DRAM. We extended HDFS version 1.0.4 with ≈ 3K lines of code for our RAIDP patch. Our simulated Lstor interposes HDFS accesses to disk, similarly to how a real Lstor employs DRAM and interposes the disk controller.

Superimposing Superchunks on HDFS . Like other distributed storage systems, HDFS stores data in "blocks" whose size typically ranges between single to hundreds of megabytes [9, 11, 56, 70]. In HDFS, the default block size is 64MB, and every block is stored as an ordinary data file (plus an associated checksum file). The "ID" of the block is the name of the file. RAIDP supports only 64MB blocks for simplicity.

HDFS is a virtual file system that maps files to blocks. This mapping is maintained in a central "Namenode", whose role, among other things, is to assign names to blocks and decide which set of "Datanodes" mirror each newly allocated block. To create a block, the local client HDFS library connects to the Namenode and receives a pair of Datanodes that will mirror the new block. RAIDP limits this assignment only to pairs of Datanodes that have a common superchunk. Once such a pair is allocated for a block, the identity of the two mirroring nodes is readily available for both the client and RAIDP, and is used by RAIDP to identify the destination superchunk.

In HDFS, blocks are stored in each Datanode's local file system as files. To straightforwardly and easily add support for RAIDP superchunks we introduce another layer of indirection between the stock HDFS Datanode code and the local file system. This layer pre-allocates a dedicated directory for every superchunk stored in the Datanode. HDFS blocks are then sequentially assigned to matching files in the relevant pre-allocated superchunk directory. The mapping between an HDFS block and its matching superchunk directory/file pair need only be maintained locally on each Datanode.

Notably, HDFS only appends data to existing files and does not support rewrites of data in place. Instead, old data must first be deleted before writing newly updated data. HDFS is not designed for such update-intensive workloads, and assumes that deleted data is generally not immediately purged. This setup is favorable to RAIDP, since parity updates of deleted data can be done in idle times, leaving the remaining block null. This removes much of the overhead of expensive read-modify-write operations once the block is reallocated.

To demonstrate the full functionality of RAIDP under non-favorable terms, we implemented two versions of RAIDP. The base version assumes parity calculations of deleted blocks are done in idle times. An additional update-oriented version assumes that data rewrites, and the resulting read-before-write and parity updates, occur online.

To evaluate the update-oriented version of RAIDP, we preallocate every superchunk's files to induce the overhead of read-modify-writes. Preallocation ensures that reading causes disk accesses instead of returning logical zeros. The implications of this option are evaluated in §6.

Maintaining Local Parity. RAIDP stores parity and journaling information in an Lstor, which is simulated in DRAM. The available DRAM in each machine in our cluster is 16GB, half of which is used to store the parity and journal information.

We modified the HDFS Datanode to interface with the local superchunks layout and Lstor. All HDFS data accesses are conducted via a unified file system interface (called FSDatasetInterface), whose implementation is transparent to the rest of HDFS. The central method of the interface (with respect to updating superchunks) is "writeToBlock". This method returns a pair of FileOutputStreams for writing a block and its associated checksum data. Our implementation returns two instances of a FileOutputStream subclass that write the same block and checksum data along with the added functionality required for RAIDP (we also maintain a parity for the checksum data across the superchunks). When writing a block to a superchunk at a given offset, RAIDP also updates the parity at this same offset.

Optimizations. HDFS transmits data packets over the network at a resolution of 64KB packets. A single 64MB block is thus comprised of 1024 such packets. When all the packets comprising a block arrive, HDFS synchronizes[2] the data to the disk. However, as explained in §3.4 RAIDP also employs a journal, which accumulates records for all writes. Acknowledging a write to a remote node in RAIDP requires that every journal update is followed by a synced write to disk. Therefore, every 64KB write is synced to disk, which cripples performance.

Our solution to this problem was to couple the syncing protocol with a mechanism for accumulating the write operations for a *full* HDFS block in memory before propagating it to the disk and its Lstor. Thus, instead of journaling (and syncing) after each packet, RAIDP syncs at the granularity of entire blocks.

[2]We note that in our baseline version of HDFS, there is no sync performed when concluding the write of a block. We unsurprisingly observe that the disk may perform I/O up to 30 seconds after the application reports completion—leaving the system vulnerable to data loss. To remedy this, we add a sync statement to both RAIDP and the baseline version of HDFS for our evaluation. In more recent versions of HDFS, a sync statement has been added [43].

By itself, however, accumulating is not enough. A Datanode may write multiple blocks (64MB files) concurrently. When an empty ext4 filesystem is instructed by HDFS to write several 64MB block files in parallel, the local file system interleaves the blocks of the different files on the disk, writing sequentially and avoiding seeks. Later, when block files are deleted from HDFS, the sectors they occupy in the local file system will be deallocated, and can be used anew upon subsequent reallocations but will incur seeks to use. In RAIDP blocks always retain their offset within their associated superchunk (for consistency with the parity in Lstor). Since in our evaluation we preallocate each superchunk's files, HDFS block files that were created at the same time in RAIDP will not be interleaved on the disk but will each be physically contiguous on the disk. As a result, a series of concurrent writes to two HDFS block files creates a "ping pong" effect, whereby the read/write head of the drive moves back and forth between the locations assigned to these files on disk.

To remedy such useless seeking and enforce sequential I/O, we extend the aforementioned accumulation optimization with a locking mechanism: When a block file has fully accumulated, the HDFS thread designated for writing the block to disk acquires a lock, thereby preventing other threads from writing to disk concurrently. These optimizations largely eliminate the performance degradation due to syncing and seeking, as shown in §6.1.

6 Evaluation

Hardware setup. We evaluated RAIDP on a 16-node cluster. Each node is a Dell PowerEdge R210 II and is equipped with a 3.10GHz Intel Xeon CPU E3-1220 V2, 16GB of memory, and a 7200 RPM 2TB disk. Each node has two ethernet NICs: a 10Gbps Broadcom NetXtreme II BCM57810 and a 1Gbps Broadcom NetXtreme II BCM5716. All of the nodes connect to a switch in a star topology via both NICs. Nodes run Ubuntu 14.04 with the 3.13.0 kernel and use the ext4 filesystem.

Methodology. We focus our evaluation exclusively on *warm* data, so that the valid comparison is to replicating systems. Erasure coding variants focus on colder data and are therefore irrelevant in this context (see §2). We implemented RAIDP in Hadoop 1.0.4 using a 6GB superchunk size so that all nodes replicate each other (i.e., 768GB effective RAIDP capacity). We compare our baseline RAIDP implementation with two-way replicating and triplicating HDFS, referred to as HDFS-2 and HDFS-3 respectively. In experiments involving the update-oriented version of RAIDP it is referred to as "raidp (re-write)". We perform the evaluation with Hadoop's default configuration. Most notably, this includes the HDFS block size of 64MB.

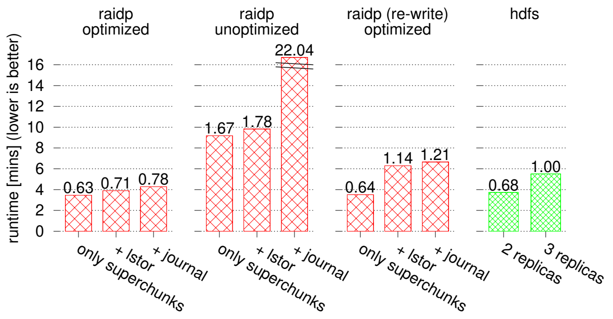


Figure 8. *RAIDP write performance as compared to HDFS-2 and HDFS-3. The basic configuration tested the performance without parity updates and journaling enabled (only superchunks). We then enable parity updates in Lstors, and the journal. Finally, we measure performance in all configurations with and without our suggested optimizations, and repeat the optimized setup with read-before-write. The numbers on top of the bars show the relative performance.*

We use standard benchmarks provided with Hadoop. We measure each result five times and present the average (unless stated otherwise all standard deviations measured were up to 8%). We run TeraSort and Wordcount benchmarks via Intel's HiBench suite [40]. Before running workloads, caches are cleared so that reads reach the disk.

6.1 Writing

Nodes in HDFS process two types of writes: "original" data, and replicated data that originates from other nodes. Because such writes may occur concurrently, data from multiple sources gets interleaved while the node writes to disk. We explained earlier in §5 that in HDFS there is no performance penalty for such interleaving, as the data is serialized sequentially by the underlying local filesystem. But having a superchunk structure means the local file sytem cannot sequentially allocate space for a series of writes to different HDFS blocks. Since the superchunks are pre-existing, writes originating from different nodes must be stored in different superchunks, each of them associated with a different mirroring node, by definition. Consequently, the disk's read/write head movement may span different superchunks between consecutive writes and incur a random I/O performance penalty.

RAIDP largely eliminates this penalty by employing the optimizations outlined in §5. Namely, it buffers the entire incoming block in memory and writes it to disk only after it arrived in its entirety, and it employs a writer's lock to prevent concurrently writing HDFS threads from interfering with each other.

Fig. 8 depicts the performance of both versions of RAIDP compared to baseline HDFS, running the standard HDFS TestDFSIO benchmark configured to write 100GB. A benchmark that only performs writes is the worst-case for RAIDP in terms of performance.

The two left most subfigures pertain to baseline RAIDP that does not perform a read-before-write, employing the superchunk layout with and without the aforementioned optimizations. In the unoptimized setup, RAIDP works with a write resolution size of 64KB (corresponding to HDFS's default "packet" size). In the optimized setup, RAIDP aggregates these packets until the entire 64MB block arrives and prevents concurrent writers to disk.

Each of the RAIDP subfigures shows three bars. The leftmost depicts a RAIDP configuration whereby only the superchunk layout takes effect without parity updates. The middle bar in the leftmost subfigure adds the overhead of updating the parity, though with no journaling.

The third bar adds journaling. To the unoptimized RAIDP variant it demonstrates an off the chart result due to the small default packet resolution. Recall that the journal dictates a disk sync after each transaction.

The optimized results for RAIDP perform much better, with the "only superchunks" setup achieving performance on par with HDFS-2 (right of Fig. 8) and even slightly superior due to a marginally imbalanced data distribution by HDFS. Parity updates and journaling add some overhead, but performance is still better than triplicating HDFS. We conclude that our optimizations eliminate most of the overhead associated with the superchunk layout.

Comparing the optimized update-oriented "re-write" variant (second subfigure from the right in Fig. 8) to triplicating HDFS shows that the former is 21% slower, which is less than the 33% upper-bound caused by having four I/Os rather than three. The journal adds a small overhead.

6.2 Reading

To evaluate the read performance of RAIDP we run a 100GB TestDFSIO benchmark that reads the data written previously by TestDFSIO (in §6.1). Fig. 9 shows the results are similar across the different configurations.

A more interesting result of the read benchmark does not appear in Fig. 9. Rather, it arises from comparing it to Fig. 8, which reveals that the runtime of reading 100GB (roughly 4 minutes) is on par with the runtime for writing 100GB in the optimized superchunks-only and HDFS-2 configurations (left and right of Fig. 8). This result stands out, because the amount of I/O produced by writing is twice that of reading, due to the replication (200GB for writing vs. 100GB for reading). Note that the optimized superchunks-only RAIDP is equivalent to HDFS-2 in that both write two replicas and nothing else, so their similar performance makes sense.

We find that this counterintuitive outcome is due to the concurrency in HDFS. Writing is done across all 16 nodes with the default setting of two tasks per node. Nodes send and receive replicas while performing their tasks, which further increases the interleaving and results in decreased disk sequentiality. Reading is done similarly, however with a 50/50 chance of reading from either replica. Whereas writes

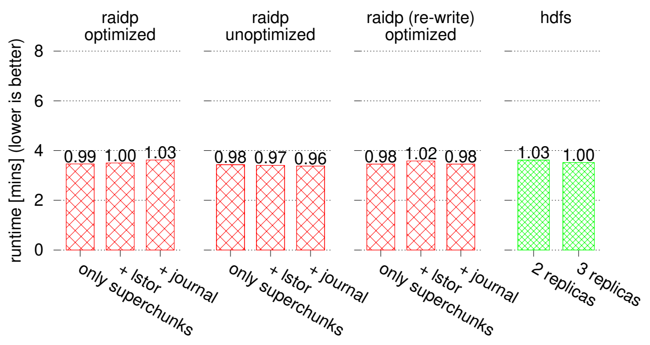


Figure 9. *RAIDP and HDFS read performance under different configurations.*

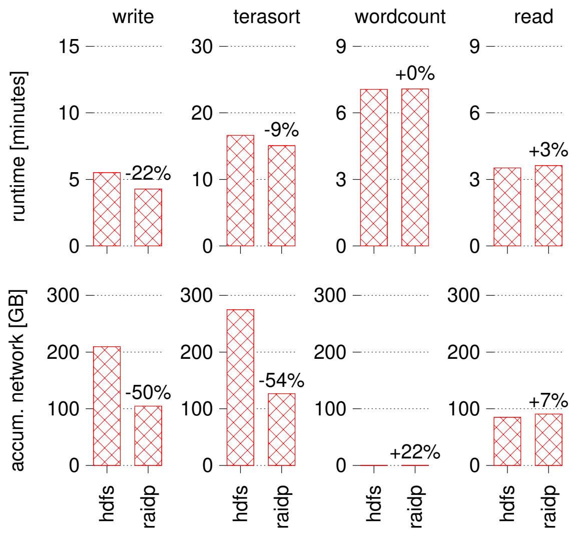


Figure 10. *RAIDP versus HDFS-3 performance*

are serialized by the local filesystem as they arrive, reads must adhere to the arbitrary layout that was generated previously when data was first written, inducing much more disk seek activity.

6.3 Benchmark Performance

We summarize the results of the standard read (§6.2) and write (§6.1) HDFS benchmarks, and contrast them with two additional benchmarks. We now only use the optimized version of baseline RAIDP and HDFS-3, as both tolerate two simultaneous disk failures. (We use the unoptimized versions of RAIDP above to increase understanding).

The first additional benchmark is TeraSort, which sorts 100GB data. The sorted data is generated by TeraGen prior to running TeraSort (generation is not included in the measured runtime). TeraSort only outputs one replica of the sorted data, so we modify it to replicate based on the configured replication factor—three for HDFS-3 and two for RAIDP—to expose the differences in performance and network usage. The results are shown in Fig. 10 (top) positioned near the results of the write benchmark to allow for easy comparison. The results indicate that the performance of HDFS-3 and RAIDP is similar. The reason is that TeraSort requires both read and write I/O as well as processing for sorting. Hence, most of the advantage of writing less replicas in RAIDP is less significant. To verify this we also repeated this test with HDFS-2, which performed only 15% better than HDFS-3 in this benchmark (versus 32% improvement over HDFS-3 in the write benchmark).

With the TeraSort and write benchmarks, data is generated locally and then replicated based on the configured replication. In RAIDP, there is one additional replica and in HDFS-3 there are two. The lower replication factor in RAIDP produces half the network traffic relative to HDFS-3, shown in the bottom left of Fig. 10. The TeraSort network results on RAIDP and HDFS-3 are qualitatively similar to those of writing.

The second benchmark is Wordcount, which computes string frequency over a 100GB input. Again, data generation is not part of the measured workload. Wordcount operates on individual word instances to produce a histogram of the counts of each word. While there are 100GB of word instances to count, there are only 100 unique words to be output along with their counts at the conclusion of the workload. Thus, Wordcount is similar to the read benchmark in that its I/O is overwhelmingly comprised of reads. However, Wordcount also includes a significant CPU component, accounting for the longer runtimes as compared to the read benchmark (top right of Fig. 10). Reads in RAIDP and HDFS-3 are similar in performance. Combined with the significant CPU processing element of Wordcount, the runtimes of RAIDP and HDFS-3 are nearly identical. The quantitative difference in their network volume is negligible and can be attributed to noise (bottom right of Fig. 10). Network traffic for RAIDP in this experiment was the only one with a standard deviation of 23%.

6.4 Superchunk Recovery

The procedure for recovering from a single disk failure in our implementation is similar to the one used by HDFS, and involves replicating non-redundant superchunks to different disks in the system. Recovering from a much rarer double-disk failure requires a similar process for most superchunks in the failed disks that are temporarily non-redundant. However, it also requires reconstructing a single superchunk that is no longer available in the system. During a superchunk reconstruction, a recovery node starts threads that request chunks of superchunk or parity data from the relevant nodes in the cluster (see §3.4). Upon receiving each chunk of parity

System type	chunk size	10Gbps NIC	1Gbps NIC
RAIDP	4MB	125 sec	827 sec
byte range lock	64MB	160 sec	848 sec
RAIDP	64MB	187 sec	850 sec
superchunk lock	4MB	211 sec	852 sec
RAID-6	4MB	1,823 sec	12,300 sec
	64MB	2,227	13,146 sec

Table 2. *16 node cluster, 6GB superchunk recovery runtimes under different configurations (lower is better).*

or superchunk data, the client threads XOR the data in memory, moving each fully-assembled block file to disk until the entire superchunk is recovered.

With 16 nodes available to participate in a 6GB superchunk reconstruction, the recovery node creates 15 different threads: 14 threads to request and XOR superchunk data, and one thread to request and XOR Lstor parity data (with a simulated dead disk). To avoid overwriting of interim data by several threads that are concurrently accessing and XORing the yet-to-be reconstructed block data, each thread in turn acquires a lock on the superchunk data that it is XORing.

Our evaluation of superchunk recovery after a double disk failure spans several different configurations for the recovery of the 6GB superchunk. We vary the amount of superchunk/-parity data requested at a time (4MB versus 64MB), and the network configuration (10Gbps versus 1Gbps interconnect between the nodes). Next, we vary the resolution of the lock over the data being XORed by the threads, and compare locking the entire superchunk versus locking only a portion of it (i.e. byte range). We also simulated the recovery time of a RAID-6 system following a dual disk failure under similar conditions.

First, we refer to the RAIDP results in the column located second-most to the right in Table 2 which use the 10Gbps network. Intuitively, when locking on the entire superchunk, working with larger (64MB) data chunks performs better than working with smaller ones (4MB). With larger chunks, a greater percentage of the recovered file is XORed while other threads wait and we better amortize the overhead of locking. This intuition is confirmed in the middle two rows (10Gbps column) of Table 2.

In contrast, if each thread locks only on the byte range of the recovered file being XORed, then requesting and XORing smaller data chunks allows more nodes to work on the recovery in parallel. The recovery configuration using a byte range lock with a 4MB chunk size performs better than the configuration with the 64MB chunk, and the best overall.

We next changed the network being used to a 1Gbps configuration, shown in the right column of Table 2. This configuration creates a network bottleneck. Changing the chunk size and lock granularity has a minimal effect on runtime, evident in the narrow range of 827 to 852 seconds shown in the table.

Finally, a RAID-6 system (bottom two rows) requires reading, transferring over the network, and performing parity calculations of all data in the remaining valid disks to reconstruct two 96GB pieces of data. As expected, for both stripes with 64MB and 4MB chunks, recovering data this way resulted in significantly longer recovery times than RAIDP.

7 Related Work

Hybrids Combining Erasure Coding & Replication. Google File System (GFS)[28], Windows Azure Storage (WAS) [11], Flat Datacenter Storage (FDS) [56], Haystack [12, 55] and HDFS [14] all triplicate warm data. The storage overheads associated with replication are costly. Each replica needs to be accommodated by not just more storage capacity, but also additional servers, power, and facility space. These costs can be decreased by utilizing only two replicas, but correlated failures [25, 71] and bad sectors on replicas used for recovery [60] encourage a higher level of redundancy.

We refer to WAS [11, 37], Facebook's storage [55, 64], HP AutoRaid [82] (which is not distributed) and DiskReduce [23] as hybrid storage systems because data is erasure coded after first being replicated on the critical path. Ghemawat et al. also discuss the potential for erasure codes for Google's read-only data [28]. Specifically for HDFS there have been several efforts to extend it with erasure codes, including HDFS-RAID [22], DiskReduce [23], HACFS [83], and Xorbas [70].

These hybrid systems exist primarily to minimize the storage costs of replication [11, 55, 64]. RAIDP is unique in that it combines replication with local erasure coding such that it is suitable for warm data, reduces storage overhead, approaches the failure tolerance of triplication, and facilitates an efficient distributed erasure coding recovery on the aforementioned warm data.

Efficient Repair. In replicated systems, re-replication of non-redundant data after a failure can be parallelized and even sped up when more disks are added, as copies of a disk's data chunks can be distributed throughout a cluster. For example, Microsoft's FDS uses a heavily parallelized recovery by utilizing a full bisection bandwidth network [56].

The same recovery using erasure codes, though parallelizable, induces significantly more network traffic, having a greater impact on the foreground jobs [64]. This traffic is due to the amount of data that must be moved to reconstruct a missing block [25, 67, 73], e.g. downloading n blocks on a remote node with a Reed-Solomon $n + k$ code.

Reducing erasure code recovery costs is an important area of research. Proposed solutions include delaying recovery [25, 73], recovering blocks that contain only popular data [75], improved block layouts [35], using SSDs or NVRAM devices to store parity [81], early failure identification [24], pipelining [47], partial parallelization [53], or applying erasure codes that minimize the number of blocks, bandwidth and/or IOPs required to recover lost data [20, 37, 41, 57, 63–65, 70, 77, 78, 84].

RAIDP's Closest Ancestors. RAIDP is in fact a specific point within design spaces described by others. RAIDP could be viewed as a variant of the "two dimensional" erasure codes [29, 32]. The n local superchunks and associated k Lstor (s) comprise one $n+k$ dimension, and the distributed replication comprises a $1 + 1$ erasure code dimension.

A more accurate representation of RAIDP than a two dimensional erasure code could be devised by using the Disaster Recovery Codes (DRC) layout [30], which like RAIDP uses persistent parity devices. DRC divides disks into 1GB "data buckets" that are two-way mirrored. Each bucket contributes to a parity value stored in a "parity group" with the parity values stored on NVRAM. The data buckets contributing to a parity group are stored on different disks. In RAIDP, given a disk, its mirroring superchunks comprise such a parity group with respect to the given disk's Lstor parity. That is, RAIDP adds the constraint that superchunks comprising a parity group reside on one disk with no other superchunks on that disk, and that no two disks share more than one superchunk.

Imposing the 1-sharing property on the above layouts and augmenting the systems with Lstors is what makes RAIDP suitable for warm data, because: (1) parity updates are *local* to nodes and require distributed synchronization only upon reconstructing a lost superchunk due to double disk failure; and (2) the local journal updates are accelerated by Lstors. In contrast, Multi-dimensional / DRC erasure codes that span multiple nodes are, by definition, *distributed* erasure codes and thus are unsuitable for warm data as explained in §2. The theory behind Lstor's design was also previously presented in a non-archival workshop [69].

Other systems that bear resemblance to RAIDP's approach include WAS, which relies on an enhanced erasure coding scheme that includes both locally and globally computed parity fragments [37] in order to minimize repair costs. Tiered Replication [18] uses triplication but assigns one replica as a write-dominated backup that is only read during recovery to reduce costs. RAIDP has elements of both replication and erasure coded systems in terms of repair traffic. As described in §1 and depicted in Fig. 1, RAIDP is on par with replication for single failures. While RAIDP reconstructs block(s) after experiencing additional failures, it only does so for a fraction of the disk, resulting in substantially less traffic than erasure coding.

8 Conclusion and Future Work

RAIDP has room for growth. Our implementation still has yet to be extended for multiple Lstors per node. Additionally, the RAIDP implementation should extend its parent distributed filesystem to support in-place updates. Real-world traces from databases could be used to showcase the I/O savings that such updates provide.

Our Lstor is simulated. The main challenge associated with making Lstors a reality is their cost efficiency. At the extreme, if Lstors turn out to be more costly or bigger than their disks, then of course they will be impractical. Consider for example the journal. The journal must be significantly faster than its disk for the system to perform well. But a large DRAM or an exceptionally powerful battery will make it expensive. One important question is whether we can utilize a small "enough" journal in a manner that does not unacceptably degrade the performance in the face of failures.

We also hope to experiment with different storage media. As the cost per GB of SSDs continues to drop SSDs may soon replace hard drives in geo-distributed data storage systems. For RAIDP the implications are multi-fold. First, upgrading to SSDs will likely reduce the amount of performance impact that random I/O currently has in our workloads. On the other hand, SSD request latency is sensitive to internal scheduling [42] which can make RAIDP Lstor update and recovery times less predictable. Second, SSDs are expected to remain significantly more expensive than hard drives, which increases the importance of reducing the TCO of systems using solutions like RAIDP.

For Lstors, there are other forms of storage worthy of consideration such as network-connected drives [50, 72], which do not require a server to be accessed since they come embedded with their own ethernet. With such an arrangement, the parity data could be further separated relative to the failure domain of the server because the disk is accessible independently. A natural extension to the idea of a networked Lstor would be to equip each Lstor with cheap wireless communication, for use in the event of a failure. Coupled with independent power such as a battery, Lstors would then be able to transmit parity and journal data with less dependence on their surrounding failure domains.

RAIDP is a performant storage solution that offers fast recovery, tolerates simultaneous failures, and provides cost savings versus other systems. RAIDP retains most of the benefits of replicated schemes while trading off some of the storage savings of erasure coding to achieve better performance, in particular for warm data and on the recovery path. In update-intensive setups RAIDP has a 21% write overhead as compared to triplication, but we believe that the potential cost savings in both hardware, power and facility costs and its suitability for warm data are a worthwhile tradeoff. We hope our design can enrich the conversation on how failure tolerance is built into data centers.

9 Acknowledgments

We thank the anonymous reviewers and our shepherd Sam H. Noh for their insightful comments on earlier drafts of the work. This research was supported by a grant from the United States-Israel Binational Science Foundation (BSF), Jerusalem, Israel, grant # 2017702, VMware, and was partially supported by the Technion Hiroshi Fujiwara cyber security research center and the Israel cyber directorate.

References

[1] DRAMeXchnage website. http://www.dramexchange.com/. (Accessed: Nov 2019).

[2] Raspberry Pi Zero. https://www.raspberrypi.org/products/raspberry-pi-zero/. (Accessed: Nov 2019).

[3] Viking Technology. http://www.vikingtechnology.com/.

[4] A review of emerging non-volatile memory (NVM) technologies and applications. *Solid-State Electronics 125* (2016), 25 – 38.

[5] Aguilera, M. K., and Janakiraman, R. Using erasure codes efficiently for storage in a distributed system. In *In Proc. of DSN'05* (2005), IEEE Computer Society, pp. 336–345.

[6] Amazon Web Services (AWS). Amazon S3 product details. http://aws.amazon.com/s3/details.

[7] Amazon.com. USB 2.0 to SATA + IDE cable adapter. http://www.amazon.com/USB-SATA-5-25-Cable-Adapter/dp/B000YJBL78, 2014. (Accessed: Nov 2019).

[8] Amazon.com. Seagate 4TB desktop HDD SATA 6Gb/s 64MB cache 3.5-inch internal bare drive (ST4000DM000). http://www.amazon.com/Seagate-Desktop-3-5-Inch-Internal-ST4000DM000/dp/B00B99JU4S, 2015. (Accessed: Nov 2019).

[9] Apache Software Foundation, T. HDFS architecture guide. http://hadoop.apache.org/docs/r1.0.4/hdfs_design.html. (Accessed: Nov 2019).

[10] Arnold, J. Erasure codes with OpenStack Swift - digging deeper. https://swiftstack.com/blog/2013/07/17/erasure-codes-with-openstack-swift-digging-deeper/, July 2013. (Accessed: Nov 2019).

[11] B. Calder et. al. Windows Azure Storage: A highly available cloud storage service with strong consistency. In *ACM Symp. on Operating Systems Principles (SOSP)* (2011), pp. 143–157.

[12] Beaver, D., Kumar, S., Li, H. C., Sobel, J., Vajgel, P., et al. Finding a needle in Haystack: Facebook's photo storage. In *OSDI* (2010), vol. 10, pp. 1–8.

[13] Bhandarkar, D. Watt matters in energy efficiency. Server design summit keynote 2012, http://www.slideshare.net/dileepb/server-design-summit-keynote-handout, 2012.

[14] Borthakur, D. *HDFS architecture guide.* The Apache Software Foundation, 2008.

[15] Ceph Documentation. Ceph erasure coded pools. https://docs.ceph.com/docs/giant/dev/erasure-coded-pool/. (Accessed: Nov 2019).

[16] Ceph Documentation. Ceph storage cluster. https://docs.ceph.com/docs/master/rados/. (Accessed: Nov 2019).

[17] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst. 26*, 2 (June 2008), 4:1–4:26.

[18] Cidon, A., Escriva, R., Katti, S., Rosenblum, M., and Sirer, E. G. Tiered replication: A cost-effective alternative to full cluster geo-replication. In *USENIX Annual Technical Conference* (2015), USENIX ATC.

[19] Cidon, A., Rumble, S., Stutsman, R., Katti, S., Ousterhout, J., and Rosenblum, M. Copysets: Reducing the frequency of data loss in cloud storage. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)* (San Jose, CA, 2013), USENIX, pp. 37–48.

[20] Dimakis, A. G., Godfrey, P. B., Wu, Y., Wainwright, M. J., and Ramchandran, K. Network coding for distributed storage systems. *IEEE Trans. Inf. Theor. 56*, 9 (Sept. 2010), 4539–4551.

[21] Dragojević, A., Narayanan, D., Nightingale, E. B., Renzelmann, M., Shamis, A., Badam, A., and Castro, M. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 54–70.

[22] Facebook Engineering. HDFS-RAID. https://engineering.fb.com/core-data/saving-capacity-with-hdfs-raid/. (Accessed: Nov 2019).

[23] 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.

[24] Fang, J., Wan, S., and He, X. RAFI: Risk-aware failure identification to improve the RAS in erasure-coded data centers. In *2018 USENIX Annual Technical Conference* (2018), USENIX ATC.

[25] 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.

[26] Ford Jr, L., and Fulkerson, D. Maximal flow through a network. *Canadian Journal of Mathematics 8* (1956), 399–404.

[27] Friedman, R., Kantor, Y., and Kantor, A. Replicated erasure codes for storage and repair-traffic efficiency. In *Peer-to-Peer Computing (P2P), 14-th IEEE International Conference on* (Sept 2014), pp. 1–10.

[28] Ghemawat, S., Gobioff, H., and Leung, S.-T. The Google File System. In *ACM Symp. on Operating Systems Principles (SOSP)* (2003), pp. 29–43.

[29] Gibson, G., Hellerstein, L., Karp, R., Katz, R., and Patterson, D. Coding techniques for handling failures in large disk arrays. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems* (1989), pp. 123–32.

[30] Greenan, K. M., Miller, E. L., Schwarz, T. J. E., and Long, D. D. Disaster recovery codes: Increasing reliability with large-stripe erasure correcting codes. In *Proceedings of the 2007 ACM Workshop on Storage Security and Survivability* (New York, NY, USA, 2007), StorageSS '07, ACM, pp. 31–36.

[31] Greenberg, A., Hamilton, J., Maltz, D. A., and Patel, P. The cost of a cloud: Research problems in data center networks. *SIGCOMM Comput. Commun. Rev. 39*, 1 (Dec. 2008), 68–73.

[32] Hafner, J. HoVer erasure codes for disk arrays. In *Dependable Systems and Networks, 2006. DSN 2006. International Conference on* (June 2006), pp. 217–226.

[33] Hamilton, J. Overall data center costs. Perspectives , http://perspectives.mvdirona.com/2010/09/overall-data-center-costs/, 2010.

[34] Hamilton, J. Why scale matters and how the cloud really is different. Re:invent 2013, https://www.youtube.com/watch?v=WBrNrI2ZsCo, 2013.

[35] Holland, M., and Gibson, G. A. Parity declustering for continuous operation in redundant disk arrays. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 1992), ASPLOS V, ACM, pp. 23–35.

[36] Hopcroft, J., and Karp, R. An nˆ5/2 algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing 2*, 4 (1973), 225–231.

[37] 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).

[38] 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.

[39] Intel. Power management in Intel architecture servers. , http://download.intel.com/support/motherboards/server/sb/power_management_of_intel_architecture_servers.pdf, 2009.

[40] Intel. HiBench - the Hadoop Benchmark Suite. https://github.com/intel-hadoop/HiBench, 2015. Github. Accessed Feb 2015.

[41] Khan, O., Burns, R., Plank, J. S., Pierce, W., and Huang, C. Rethinking erasure codes for cloud file systems: Minimizing I/O for recovery and degraded reads. In *USENIX Conf. on File & Storage Technologies (FAST)* (San Jose, February 2012).

[42] Kim, J., Lee, E., and Noh, S. H. I/o scheduling schemes for better I/O proportionality on flash-based SSDs. In *IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems* (2016), MASCOTS, pp. 221–230.

[43] Kuang, H. Support hsync in HDFS [issue: Hdfs-744]. https://issues.apache.org/jira/browse/HDFS-744, May 2012. Hadoop HDFS JIRA issue tracker (Accessed: November 2019).

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

[45] Lakshman, A., and Malik, P. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev. 44*, 2 (Apr. 2010), 35–40.

[46] Lee, E. K., and Thekkath, C. A. Petal: Distributed virtual disks. *SIGOPS Oper. Syst. Rev. 30*, 5 (Sept. 1996), 84–92.

[47] Li, R., Li, X., Lee, P. P. C., and Huang, Q. Repair pipelining for erasure-coded storage. In *USENIX Annual Technical Conference* (2017), USENIX ATC.

[48] MacWilliams, F. J., and Sloane, N. J. A. *The theory of error-correcting codes*, vol. 16. Elsevier, 1977.

[49] Manasse, M. S., Thekkath, C. A., and Silverberg, A. A Reed-Solomon code for disk storage, and efficient recovery computations for erasure-coded disk storage. *Proceedings in Informatics* (July 2005).

[50] Marvell. Marvell 88SS5000 NVMeoF SSD controller shown with Toshiba BiCS. https://www.servethehome.com/marvell-88ss5000-nvmeof-ssd-controller-shown-with-toshiba-bics. (Accessed: Feb. 2020).

[51] Microsemi. Flashtec NVMe controllers. https://www.microsemi.com/product-directory/storage/3687-flashtec-nvme-controllers. (Accessed: Nov 2019).

[52] 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.

[53] Mitra, S., Panta, R., Ra, M.-R., and Bagchi, S. Partial-parallel-repair (PPR): A distributed technique for repairing erasure coded storage. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), EuroSys.

[54] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS) 17*, 1 (1992), 94–162.

[55] Muralidhar, S., Lloyd, W., Roy, S., Hill, C., Lin, E., Liu, W., Pan, S., Shankar, S., Sivakumar, V., Tang, L., et al. f4: Facebook's warm BLOB storage system. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation* (2014), USENIX Association, pp. 383–398.

[56] Nightingale, E. B., Elson, J., Fan, J., Hofmann, O., Howell, J., and Suzue, Y. Flat datacenter storage. In *USENIX Symp. on Operating Systems Design & Implementation (OSDI)* (2012), pp. 1–15.

[57] Pamies-Juarez, L., Blagojević, F., Mateescu, R., Gyuot, C., Gad, E. E., and Bandić, Z. Opening the chrysalis: On the real repair performance of msr codes. In *14th USENIX Conference on File and Storage Technologies* (2016), FAST.

[58] Pamies-Juarez, L., Oggier, F., and Datta, A. Decentralized erasure coding for efficient data archival in distributed storage systems. In *Distributed Computing and Networking*. Springer, 2013, pp. 42–56.

[59] Pillai, T. S., Chidambaram, V., Alagappan, R., Al-Kiswany, S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, Oct. 2014), USENIX Association, pp. 433–448.

[60] 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.

[61] Plank, J. S., and Huang, C. Tutorial: Erasure coding for storage applications. Slides presented at FAST-2013: 11th Usenix Conference on File and Storage Technologies, February 2013.

[62] Prabhakaran, V., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Analysis and evolution of journaling file systems. In *USENIX Annual Technical Conference, General Track* (2005), pp. 105–120.

[63] Rashmi, K., Nakkiran, P., Wang, J., Shah, N. B., and Ramchandran, K. Having your cake and eating it too: Jointly optimal erasure codes for I/O, storage, and network-bandwidth. In *13th USENIX Conference on File and Storage Technologies (FAST 15)* (Santa Clara, CA, Feb. 2015), USENIX Association, pp. 81–94.

[64] Rashmi, K. V., Shah, N. B., Gu, D., Kuang, H., Borthakur, D., and Ramchandran, K. A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the Facebook warehouse cluster. In *Proceedings of the 5th USENIX Conference on Hot Topics in Storage and File Systems* (Berkeley, CA, USA, 2013), HotStorage'13, USENIX Association, pp. 8–8.

[65] Rashmi, K. V., Shah, N. B., Kumar, P. V., and Ramchandran, K. Explicit construction of optimal exact regenerating codes for distributed storage. In *47th Annual Allerton Conference on Communication, Control, and Computing* (2009).

[66] 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.

[67] Rodrigues, R., and Liskov, B. High availability in DHTs: Erasure coding vs. replication. In *Peer-to-Peer Systems IV*. Springer, 2005, pp. 226–239.

[68] Rosenfeld, E. RAIDP: ReplicAtion with Intra-Disk Parity. Master's thesis, Technion—Israel Institute of Technology, Israel, 2015.

[69] Rosenfeld, E., Amit, N., and Tsafrir, D. Using disk add-ons to withstand simultaneous disk failures with fewer replicas. Workshop on the Interaction amongst Virtualization, Operating Systems and Computer Architecture (WIVOSCA), 2013.

[70] Sathiamoorthy, M., Asteris, M., Papailiopoulos, D., Dimakis, A. G., Vadali, R., Chen, S., and Borthakur, D. XORing elephants: Novel erasure codes for big data. *Proceedings of the VLDB Endowment* (2013).

[71] 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).

[72] Seagate. Seagate Kinetic open storage platform. https://www.seagate.com/em/en/support/enterprise-servers-storage/nearline-storage/kinetic-hdd. (Accessed: Feb. 2020).

[73] Silberstein, M., Wang, Y., Ganesh, L., Alvisi, L., and Dahlin, M. Lazy means smart: Reducing repair bandwidth costs in erasure-coded distributed storage. In *Proceedings of the 7th ACM International Systems and Storage Conference* (2014), SYSTOR'14, USENIX Association.

[74] Thinkmate. Superstorage server 6048r-e1cr72l. , http://www.thinkmate.com/system/superstorage-server-6048r-e1cr72l.

[75] Tian, L., Feng, D., Jiang, H., Zhou, K., Zeng, L., Chen, J., Wang, Z., and Song, Z. PRO: A popularity-based multi-threaded reconstruction optimization for RAID-structured storage systems. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2007), FAST '07, USENIX Association, pp. 32–32.

[76] Wang, G., Zhang, L., and Xu, W. What can we learn from four years of data center hardware failures? In *IEEE/IFIP International Conference on Dependable Systems and Networks* (2017), DSN.

[77] Wang, Z., Dimakis, A. G., and Bruck, J. Rebuilding for array codes in distributed storage systems. *CoRR abs/1009.3291* (2010).

[78] Weatherspoon, H., and Kubiatowicz, J. D. Erasure coding vs. replication: A quantitative comparison. In *Peer-to-Peer Systems*. Springer, 2002, pp. 328–337.

[79] website, D. Poweredge rack servers. http://www.dell.com/us/business/p/poweredge-rack-servers/. (Accessed: July 2019).

[80] Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D. E., and Maltzahn, C. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2006), OSDI '06, USENIX Association, pp. 307–320.

[81] Wildani, A., Schwarz, T., Miller, E., and Long, D. Protecting against rare event failures in archival systems. In *Modeling, Analysis Simulation of Computer and Telecommunication Systems, 2009. MASCOTS '09. IEEE International Symposium on* (Sept 2009), pp. 1–11.

[82] Wilkes, J., Golding, R., Staelin, C., and Sullivan, T. The HP AutoRAID hierarchical storage system. *ACM Trans. Comput. Syst. 14*, 1 (Feb. 1996), 108–136.

[83] Xia, M., Saxena, M., Blaum, M., and Pease, D. A. A tale of two erasure codes in HDFS. In *13th USENIX Conference on File and Storage Technologies (FAST 15)* (Santa Clara, CA, Feb. 2015), USENIX Association, pp. 213–226.

[84] Xiang, L., Xu, Y., Lui, J. C., and Chang, Q. Optimal recovery of single disk failure in RDP code storage systems. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2010), SIGMETRICS '10, ACM, pp. 119–130.

[85] Zhang, D., Ehsan, M., Ferdman, M., and Sion, R. Dimmer: A case for turning off dimms in clouds. In *Proceedings of the ACM Symposium on Cloud Computing* (New York, NY, USA, 2014), SOCC '14, ACM, pp. 11:1–11:8.