RAIDP: ReplicAtion with Intra-Disk Parity

Eitan Rosenfeld
RAIDP: ReplicAtion with Intra-Disk Parity

Research Thesis

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Eitan Rosenfeld

Submitted to the Senate of the Technion — Israel Institute of Technology
Adar 5775 Haifa February 2015
This research was carried out under the supervision of Assistant Professor Dan Tsafrir, in the Faculty of Computer Science, and Dr. Michael Factor, from IBM Research Haifa.

Some results in this thesis have been published as articles by the author and research collaborators in conferences and journals during the course of the author’s research period, the most up-to-date versions of which being:


**ACKNOWLEDGEMENTS**

I would like to thank my advisors, Assistant Professor Dan Tsafrir and Dr. Michael Factor, for their expert guidance, patience, and time that they spent working with me to produce this research.

The Technion’s funding of this research is hereby acknowledged.
## Contents

List of Figures

Abstract ................................. 1

1 Introduction ............................ 3

2 Why Replication is Hard to Avoid ........ 7

3 Design .................................. 13
   3.1 Layout ............................... 13
   3.2 Disk Add-Ons (Lstors) ............... 15
   3.3 Recovery ............................. 17
   3.4 Crash Consistency .................... 19

4 Replication vs. Erasure Codes vs. RAIDP .... 25

5 Implementation .......................... 33

6 Evaluation .............................. 37
   6.1 Methodology ......................... 37
   6.2 Sync Overhead ....................... 38
   6.3 Writing ............................ 39
   6.4 Reading ............................ 41
   6.5 Benchmark Performance ............. 43
   6.6 Superchunk Recovery ............... 44
   6.7 Potential Benefit of Updates-In-Place .. 46

7 Related Work ........................... 47

8 Future Work ............................ 51

9 Conclusions ............................ 55

Hebrew Abstract ......................... i
# List of Figures

1.1 Triplication induces low repair traffic, but it is wasteful in terms of capacity. Erasure coding is more storage efficient but induces much higher repair traffic. For this and additional reasons, triplication is suitable for warm data and erasure coding is not. RAIDP constitutes a middle point. It enjoys most of the benefits of triplication while being more storage efficient. .................................................. 4

2.1 Writing a full stripe in an erasure coding system comprised of $n+k$ blocks, each of the size of $m$ bytes (top) vs. writing one $m$-sized block in a replicating system (bottom). The former takes longer due to having to wait for the entire stripe to arrive before parities can be computed. .................................................. 9

3.1 Example for a superchunk layout that satisfies 1-sharing and 1-mirroring. Columns are disks. Rows are superchunks within disks. Numbers are IDs of superchunks. IDs in bold correspond to superchunks that mirror disk $D_0$. ................................. 14

3.2 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. ................................................................. 17

3.3 Illustrating RAIDP. Each superchunk appears twice, no two disks share more than one superchunk, and every disk’s superchunks are XORed in the Lstor. ........................................................... 18

6.1 Overhead of reading before writing when syncing at different intervals for two 6GB workloads: writing and reading before writing. The x axis shows the I/O operations performed between syncs. The left y axis demonstrates the throughput of each workload illustrated by the write and read-before-write workload lines. The right y axis shows the relative throughput of reading before writing, as compared to (divided by) the write throughput. .......................... 38

6.2 RAIDP write performance as compared to HDFS-2 and HDFS-3. .................. 40

6.3 Disk utilization is more efficient with the additional read before writing. .... 41

6.4 RAIDP seek distance and I/O request size with and without reads before writes. 41

6.5 RAIDP and HDFS read performance under different configurations. ............ 43

6.6 HDFS reads degrade faster than writes due to increased seek distance. ........ 43

6.7 RAIDP versus HDFS-3 performance ......................................................... 45
8.1 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 $D_1$.

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

8.3 Example for a superchunk layout that requires a superchunk rearrangement upon failure. Columns are disks. Rows are superchunks within disks. Numbers are IDs of superchunks. If $D_2$ fails, then $D_1$ cannot replicate the non-redundant superchunk 1 because doing so violates 1-sharing.
Abstract

Distributed storage systems often triplicate data to reduce the risk of permanent data loss, thereby providing failure tolerance for 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 largely retains the benefits of replication, trading off some performance for substantially reduced storage and networking overheads. 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 and 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.) We implement RAIDP in HDFS, which triplicates by default. We show that RAIDP performs within 0%–24% of the baseline while halving the storage and networking overheads and providing similar durability.
Chapter 1

Introduction

Distributed storage systems overwhelmingly favor the use of replication over erasure coding when storing warm data \cite{FTXG09,SAP+13,B11,GGL03,NEF+12,Bor08,SAP+13} for the following reasons. Replication is preferable for \textit{reads} because: (1) it allows for load balancing, such that if one node is currently heavily loaded, then the desired data can be retrieved from another node; (2) it can be used to accelerate reads by utilizing several replicas in parallel; and (3) it avoids the problem of “degraded reads” \cite{FTXG09,HSX+12,KBP+12} 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 \textit{writes}, because (4) it may reduce latency by possibly eliminating the need to wait for an entire stripe to accumulate before the parity can be computed.\footnote{With erasure coding, the only way to survive intermediate failures while submitting data to the distributed file system is to compute fragment replicas or parities for the subset of submitted fragments. However this would cause additional load—on the network and disk if replicating, and additionally, on the CPU if computing parities. Such a design would then forgo many advantages of erasure coding (the reduced network and disk load for full stripe writes) and possibly further aggravate the CPU load. Hence, the sync (where such functionality is desired) is typically deferred to the end of full stripe writes.}

Replication is preferable to erasure coding for both \textit{reads} and \textit{writes} because: (5) it avoids the CPU processing of encoding the data and later 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 repair overhead, because replication recovery involves only one node that holds the replica, whereas erasure recovery typically involves most nodes associated with the encoding—thus, repair traffic of erasure coding distributed storage systems was shown to dramatically interfere with, and reduce the available bandwidth for, foreground jobs \cite{MLR+14,RSG+13}.\footnote{Although these advantages relate to the performance characteristics of magnetic drives, we believe many of these conclusions would apply to SSDs as well. The one possible exception is the sequentiality of writes, since seeks may adversely affect magnetic drives more than random writes affect SSDs.}

Alongside the advantages of replication, it has one serious deficiency of inducing
high storage overhead (a replication factor of $k$ results in $\frac{k-1}{k}$ “wasted” capacity). This drawback has brought about the use of replication for warm data in tandem with erasure coding for cold, read-only data [B. 11, HSX+12, FTXG09, MLR+14, WGSS96, RSG+13, Cepa]. For example, in the Ceph distributed file system [WBM+06], “erasure-coded pools [are used for] cold storage with high latency and slow access time” [Cepa]. Such designs allow for the 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 afford to have the reduced availability associated with erasure coding.\footnote{Throughout this thesis, the term “availability” is defined to mean “availability that is fast enough to be suitable for warm data”, meaning data can be accessed at will, and the runtime overhead of any needed reconstruction of the data to enable access is transparent to clients.}

The state of the art is therefore as follows. Provisioning a datacenter using both redundancy schemes leaves data in one of two extremes. Either the system provides high availability, performs well, induces little repair traffic, but is wasteful in terms of storage. Or it has limited availability, induces significant repair traffic, but is efficient in terms of storage. These tradeoffs help define our goals for this research. We discuss them in §2.

We introduce a new distributed redundancy scheme called RAIDP, which stands for ReplicAtion with Intra-Disk Parity. RAIDP constitutes a new point in the design space of distributed systems. It is a hybrid system that combines replication with “local” erasure coding, with the goal of deemphasizing the weaknesses of the two redundancy schemes at the expense of some of their strengths. RAIDP forms a middle ground design that is applicable to warm data—largely enjoying the aforementioned 7 advantages of replication, providing comparable resiliency, and substantially reducing storage overhead. Figure 1.1 illustrates some of these tradeoffs.

RAIDP is characterized by three key traits. First, it uses exactly two replicas instead of the typical three (or more) [KBP+12, B. 11, GGL03, NEF+12, Bor08]. Second, it
partitions each disk into (logically or physically) contiguous, equisized “superchunks”, on the order of a few GBs. Each superchunk is two-way replicated on different physical disks, and each superchunk stores the shared content between the two disks it appears on (and nothing more). 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 drive. We denote these devices as Lstors, standing for “local storage devices”. An Lstor fails separately from its disk, is much faster/smaller, and is persistent. In its simplest configuration, RAIDP associates each drive 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 (RAID4-like) non-rotated erasure code of its disk’s superchunks. Since this information is completely local to the node containing the disk, Lstors do not burden the network in order 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, e.g., \( D_1 \)’s Lstor and its surviving superchunks, which are replicated elsewhere. Each Lstor additionally stores a journal used for reconciling partial updates after failures. We describe the design of RAIDP in §3 and highlight the tradeoffs that it makes in §4.

We implement RAIDP by superimposing the superchunk layout on top of HDFS, supplementing each disk with a single simulated Lstor, and incorporating the aforementioned journaling scheme. We describe our implementation in §5. We then experimentally evaluate the performance of our implementation. The results confirm that the network and storage overheads are halved as compared to the default (triplicating) HDFS. The cost is 0–24% runtime degradation, which is largely induced by additional disk operations that are required for maintaining the parity information on the Lstors. We additionally demonstrate that RAIDP successfully recovers from dual disk failures. We discuss the performance evaluation in §6, survey the related work in §7, consider future work in §8, and conclude in §9.
Chapter 2

Why Replication is Hard to Avoid

Due to the scale of modern datacenters and the non-negligible likelihood of failures, datacenters must employ safeguards to prevent data loss and preserve data availability. Preventing loss necessitates that data can be eventually recovered after a failure. 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 accessing that data [CDG08, LM10, FLP+10].

In today’s distributed storage systems, data is typically stored in a declustered fashion, where each logical unit of data—including the data itself as well as the corresponding replicas or parity information—is spread across many, typically randomly chosen servers [CRS+13]. Upon a failure, modern systems reconstruct the data using the declustered chunks, in parallel, on many other disks for a fast recovery [Cepb, GMSL07, ASF, HSX+12, NEF+12]. This recovery procedure differs from classic (non-distributed) schemes that use hardware RAID controllers to reassemble entire failed disks on a hot spare, a procedure that can take hours or days [Lev09, Seaa]. Declustered recovery is much quicker than classic recovery because the latter reconstructs an entire disk on a single disk, essentially eliminating the ability to recover in parallel, whereas the former does not even require having a spare, instead recovering in parallel using spare capacity from a large set of disks.

With distributed replication, $k$ additional copies of each data unit 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 a $n + k$ “stripe.” Each of the original $n$ blocks can be recovered so long as at least $n$ of the $n + k$ stripe blocks are available. A $n + k$ erasure code can therefore survive up to $k$ failures with an overhead of $k/n$ [RS60, HSX+12]. Overwhelmingly, $n$ is bigger than $k$, so erasure coding provides failure tolerance with a smaller storage overhead as compared to replication. The latter induces 100% storage overhead (namely, wastes half of the space) when using only two replicas. It induces 200% overhead (two thirds of the space) when using three replicas, and so on. Still, despite its high overhead, replication has
Table 2.1: Advantages of replicating systems for warm data.

<table>
<thead>
<tr>
<th><strong>Operation</strong></th>
<th><strong>Advantage of Replication</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>1. can load balance</td>
</tr>
<tr>
<td></td>
<td>2. can parallelize</td>
</tr>
<tr>
<td></td>
<td>3. no degraded reads</td>
</tr>
<tr>
<td>Write</td>
<td>4. shorter sync latency</td>
</tr>
<tr>
<td>Read/Write</td>
<td>5. improved sequentiality</td>
</tr>
<tr>
<td></td>
<td>6. reduced cpu consumption</td>
</tr>
<tr>
<td></td>
<td>7. reduced repair traffic</td>
</tr>
</tbody>
</table>

Key advantages that make it the redundancy scheme of choice for warm data in modern distributed systems [GGL03, B. 11, NEF+12, Bor08, SAP+13, KBP+12, Ama, FTXG09].

In the remainder of this section, we explain in more detail these key advantages of replication over erasure coding as were enumerated in the beginning of the introduction section and as briefly summarized in Table 2.1. We further survey the existing approaches that attempt to simultaneously leverage the advantages of both replication and erasure coding, while pointing out their shortcomings. 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 [AJ05]. The prevalent Reed-Solomon is an example of such a code [RS60]. In “non-systematic” codes, all data accesses require decoding, and are hence slower [PJOD13]. 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, most of discussion also applies to the latter. 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 [MS77]. More advanced codes such as LRC codes [HSX+12, SAP+13] are not MDS codes, which means recovery is possible using fewer than \( n \) devices, but also means that only a specific subset of those devices will enable a recovery. 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, for example, due to machine reboots and rolling upgrades [B. 11, LT96]. 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 [HJH00]. Erasure coding, in contrast, contains only one copy of each data unit plus the means to reassemble those data units remotely. 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 completely avoid a stressed or unavailable node, the system must perform a much slower “degraded read”, whereby the erasure coded data is reassembled elsewhere, burdening \( n \) nodes with reading and transmitting the \( n \) blocks from the stripe to cope with the
erasure_write_stripe( byteStream s ) {
    char p[k][m] = <0>; // k parities
    for( j = 1...n ) // get n blocks of stripe
        blk[m] = s.get_next_block()
        network_send( j, blk )
    for( i = 1...k )
        p[i] = compute_parity( i, p[i], blk )
    for( i = 1...k )
        network_send( p[i] )
    gather_acks( n+k )
    ack_client()
}

replication_write_block( byteStream s ) {
    blk[m] = s.get_next_block()
    for( i = 1...k )
        network_send( i, blk )
    gather_acks( k )
    ack_client()
}

Figure 2.1: Writing a full stripe in an erasure coding system comprised of \( n + k \) blocks, each of the size of \( m \) bytes (top) vs. writing one \( m \)-sized block in a replicating system (bottom). The former takes longer due to having to wait for the entire stripe to arrive before parities can be computed.

limited availability (or unavailability) of the original node [FTXG09, KBP+12, HSX+12].

**Improved Sync Latency & Sequentiality** Distributed storage systems, in particular those supporting erasure coding, partition streams of incoming data to blocks of \( m \) bytes, where \( m \) can be on the order of single to hundreds of megabytes [B.11, ASF, NEF+12, SAP+13]. These relatively large \( m \) values improve performance, because they promote sequentiality and reduce the number of random disk seeks. Large \( m \) values, however, 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 latency is prolonged because of the need to wait for all the \( n \cdot m \) bytes to arrive before finalizing the computation of the associated \( k \) parities and acknowledging the client, as illustrated at the top of Figure 2.1. Replication, in contrast, provides shorter sync latency, because every \( m \)-sized block can be immediately replicated and ack-ed [Arn13, ASF], as illustrated at the bottom of Figure 2.1. Erasure coding systems may of course opt for alleviating the sync latency problem by reducing the block size \( m \). But then the smaller blocks would reduce the sequentiality on disk for the impending writes and future reads. A smaller size would additionally increase the complexity of the system due to generating more metadata.

**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 being written. (Compare the top and bottom procedures in Figure 2.1.)
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. As demonstrated in the f4 study by Facebook, “CPU decryption is computationally expensive [when reconstructing a stripe]” [MLR+14]; in fact, because of this overhead, block reconstruction at Facebook is done in an altogether different tier of machines, which place emphasis on CPU cores instead of storage [MLR+14]. We encountered this effect in the experimental section §6, where we show that the CPU can become the bottleneck during reconstruction when the network and disk are not bottlenecked. Hence, systems reserve erasure coding for cold data and relegate the encoding to the background, whereby the aforementioned disadvantages are less of a concern [B.11, FTXG09, MLR+14, SAP+13].

**Reduced Repair Traffic** The final—and arguably most important—key advantage of replication over erasure coding concerns the repair traffic used to overcome disk failures, consisting of both disk I/O and network I/O. Consider the case of a single disk failure, which, according to Rashmi et al., accounts for 98% of the observed failure scenarios in distributed file systems [RSG+13]. 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 equal to the amount of data that was lost. In contrast, the repair traffic induced by a $n + k$ erasure coding system is $n$ times greater. Namely, for every reconstructed byte, $n$ bytes will be read from the disks across the system and then transmitted over the network.

The importance of this drawback cannot be overstated, as highlighted by a significant body of recent research directed towards reducing the repair traffic [RSG+13, KBP+12, HSX+12, WK02, WDB10, XXLC10, DGW+10, SAP+13]. For example, Facebook “estimate[s] that if 50% of the cluster was Reed-Solomon encoded, the repair network traffic would completely saturate the cluster network links” [SAP+13], noting that with only 8% of the data being erasure coded, the repair traffic already consumes a disproportionate amount of the available bandwidth, which adversely affects the performance of foreground map-reduce jobs [RSG+13, SAP+13].

**Partial Solutions** Due to the serious drawbacks of erasure coding as outlined above, production distributed storage systems opt for replication when storing their warm, frequently accessed data [Ama,B.11, GGL03, MLR+14, FTXG09, ASF, NEF+12, SAP+13]. But the storage overheads associated with replication are painful and costly. Replicating systems typically store at least three replicas, and each replica needs to be accommodated by not just more storage capacity, but also additional servers, power, and facility space. These costs can in principle be decreased by utilizing only two replicas. But correlated failures [FLP+10, SG07] and bad sectors on replicas used for recovery [PWB07] promote using a higher level of redundancy.

Researchers attempted to alleviate the problem of excessive repair traffic to a certain extent by: delaying recovery, e.g., in case the failure is transient [FLP+10,
recovering blocks that contain only popular data \cite{TFJ07}; and applying more sophisticated erasure codes that use fewer blocks and less traffic for recovery \cite{RSG13, KBP12, HSX12, WK02, WDB10, XXLC10, DGW10, SAP13}. None of these solutions eliminate the problem, and, to our knowledge, none are applied to warm data in production distributed file systems. (Nor can they be applied with satisfactory results, we speculate, due to the aforementioned disadvantages of erasure coding.)

A complimentary thread of research relates to devising hybrid systems that combines replication for warm data and erasure coding for cold data in the background, such that the nature of the workload minimizes the schemes’ disadvantages \cite{GGL03, MLR14, B11, FTXG09, SAP13, Had, WGSS96}. For example, encoding cold data in the background means that larger stripe blocks can be encoded to promote sequentiality and encoding can be done in a throttled manner that does not overburden the system resources. At the same time, replicating warm data means that prioritized recoveries will not be as costly. Thus, when a workload consists of warm and cold data, hybrids are preferable over keeping data in either a replicated or an erasure coded state, as the distributed file system benefits from a redundancy scheme that remains applicable to warm data, largely retaining the advantages of triplication while incurring a smaller storage footprint. Systems still pay the full price of triplication for their warm data.
Chapter 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 this 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 (physically or logically) contiguous. 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 the same offset. We call this property “1-mirroring”, denoting that each disk’s superchunks are two-way replicated. RAIDP also ensures that no two disks share more than one superchunk, such that at most one superchunk is lost in the event of a dual failure. This latter property is called “1-sharing”.

Construction  An example of the RAIDP data layout is depicted in Figure 3.1, whereby columns are disks and rows are superchunks. Observe how the superchunks on disk $D_0$ are replicated on the rest of the disks (highlighted by a shaded background). We can see that $D_0$ adheres to 1-mirroring and 1-sharing, as no two copies of its superchunks exist on the same disk and all are replicated twice. Therefore, if $D_0$ and another disk fail simultaneously, only the one superchunk that they share is lost, whereas the rest of their superchunks merely become non-redundant. Because all of the disks in our example observe 1-mirroring/1-sharing, and, additionally, every two disks share a superchunk with one another, a failure of any two disks would result in one lost superchunk. In general, however, a failure of two disks results in at most one lost superchunk, because
the layout may include pairs of disks that do not share.

Arranging the data across the system in a manner that preserves 1-sharing and 1-mirroring is easy and could be done in many ways. Figure 3.1 shows one example. The dotted lines identify pairs of superchunk replicas. Observe that first-row superchunks are replicated in the second row, that third-row superchunks are replicated in the fourth row, and that, generally, $i$-th row superchunks are replicated in the $i + 1$ row for every odd $i$. Further observe that the replication shift between rows is gradually increasing: the replica shift between the first and second rows is exactly one column, the shift between the third and fourth row is exactly two columns, and, generally, the shift between the pair of rows $2i - 1$ and $2i$ is always $i$ columns. Thus, the initial replication layout of the storage system in this example is given by:

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

where $j$ is the index of the disk (column), $i$ is the index of the superchunk within that disk (row), and $N$ is the number of disks in the system.

**Implications**  
Notice that in the seven-disk configuration example in Figure 3.1, 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 by at most $N - 1$ superchunks, which means the number of superchunks across the system is at most $N \cdot (N - 1)$.

Suppose that a RAID5 configuration has 1000 disks of 4TB each. Then, as noted, each disk may hold a maximum of 999 superchunks. Assuming those superchunks span the entire disk and each disk is replicated across all other disks, then there are 999 superchunks of the size 4TB / 999 ≈ 4GB. The minimal size for a superchunk is therefore $S/(N - 1)$, where $S$ is the size of the disk. The superchunk size will be larger if a disk is replicated on less than $N - 1$ other disks.
After experiencing a single disk failure, it is critical that we preserve the 1-sharing and 1-mirroring properties when we re-replicate the failing disk’s non-redundant superchunks from their mirroring nodes. With one less disk in the system, the total number of superchunks to be laid out must be at most $N' \cdot (N' - 1)$, where $N' = N - 1$ due to the disk failure. Thus, superchunks can be arranged to maintain 1-sharing and 1-mirroring after $f$ failures, as long as there are at most $(N - f) \cdot (N - f - 1)$ superchunks to arrange.

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 in the event of multiple failures.

### 3.2 Disk Add-Ons (Lstors)

For the remainder of the discussion, we assume the storage space is laid out within RAIDP superchunks, which ensures that data loss upon a dual disk failure is encapsulated within a single superchunk. To be able to recover this superchunk, 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. The 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 to 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). As the storage capacity of the Lstor is compatible with that of the superchunk size, it can be reduced by increasing the number of disks in the system as explained above. Lstors interpose the I/O between their disk and its controller, as illustrated in Figure 3.2. Such interposition can be logical or physical. We assume Lstors have the following properties:

1. Disk failures occur separately from Lstor failures.
2. The Lstor capacity is that of a superchunk to keep parity information, plus a small journal.
3. Parity information can be stored on the Lstor at least as quickly as the data can
be stored on the associated disk.

4. The Lstor’s journal is fast. It can be read from and written to at a high enough speed such that the overhead is negligible as compared to the corresponding disk I/O. It can be implemented, for example, using DRAM equipped with a battery.

5. The Lstor parity and journal content are persistent.

We envision the physical size of an Lstor to be similar to, say, a pinkie-sized SATA-to-USB converter that costs only a few U.S. dollars [Ama14]. We speculate that Lstors could be made part of disk drive enclosures, or disk drives themselves.

**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. Such a reduction can be achieved by increasing the number of disks in the system, as explained earlier. A realistic system of about 1000 disks of 4TB can yield an Lstor capacity as small as $4\text{TB}/999 \approx 4\text{GB}$.

Suppose we implement the 4GB Lstor. We could do so, for example, with 4GB of flash, 4GB of DRAM, and a battery. The flash storage provides persistence for the parity information. However, the parity must be updated upon each write operation directed at the disk, so the flash might experience wear. This problem can potentially be resolved by storing most updates in DRAM and only periodically persisting the DRAM contents on the flash storage in a sequential manner. The battery will then be used only in the event that power is lost while DRAM contents are not yet persisted, powering the DRAM until it is copied to the non-volatile flash. Alternatively, if the flash component is big enough such that wear is less of a problem, the DRAM size could be shrunk to only accommodate the Lstor journal. An intriguing possible direction to explore could be to enhance RAIDP by equipping Lstors with cheap wireless communication, thereby allowing them to talk sideways with other disks in the cluster when they are otherwise unreachable.

Even today, Solid State Hybrid Drives (SSHD) such as Seagate’s employ two types of storage—SSD and spinning platters—in a single device [Seac]. The SSD portion in the Seagate drive is 8GB, larger than what we envision the capacity of the Lstor to be. While today’s SSHDs are designed to boost performance via caching, we postulate that the interposing properties of such hybrid devices could be repurposed for reliability.

In this study, we do not build Lstors. Rather, we assume that it is possible to build them, and we simulate them in DRAM as discussed in §5. We leave the question of whether Lstors are feasible and economical open.

**Stacking Lstors** Another possible feature we consider below is that Lstors can potentially be “stacked”, such that every disk is associated with $k > 1$ Lstors, rather than just one. We outline how such a feature could be used in order to increase the failure
Figure 3.2: 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.

tolerance of the system. Placing additional Lstors on each disk enables the storage of additional local parities, which can be used to recover from additional simultaneous failures. The required capacity per Lstor is the size of a single superchunk, regardless of the number of Lstors per disk.

Cost of Using Lstors The Lstor on each machine stores 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 read-modify-write sequence negatively affects 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 (recall that according to Property 3 above, the Lstor is at least as fast as the disk), 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 double disk failures. Where triplication performs three writes, RAIDP performs two reads and two writes for a total of four I/Os. In principle, RAIDP could equalize the overhead by reading the old data only 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.

3.3 Recovery

To provide a recovery solution that withstands simultaneous disk failures and occasional bad sectors, the Lstors local to a disk continuously maintain the up-to-date superchunk erasure codes of that disk. We describe two recovery scenarios: (1) surviving two disk failures with a single Lstor on each disk, and a generalized solution for (2) surviving \( k + 1 \) disk failures with \( k \) Lstors.
Surviving a Double Disk Failure  A single-Lstor configuration is illustrated in Figure 3.3, 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 Figure 3.3 failed, then block $e$ is lost because both of its replicas are gone. But $e$ can be recomputed 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 gives RAIDP the ability to use a different Lstor if one is unavailable or if there are hotspots that are ideally avoided for the reconstruction. Alternatively, the two Lstors and sets of mirroring superchunks can be used to each reconstruct the “lost” superchunk in parallel, with each set used for reconstructing half.

Surviving Additional Failures  Such a recovery also works for additional simultaneous failures when multiple Lstors are “stacked” on each disk. For example, if $k$ Lstors are attached to each disk with $n$ superchunks per disk, then the Lstors can store non-rotated $n + k$ erasure codes, where each of the $k$ parities is stored on an Lstor. RAIDP tolerates a simultaneous failure of up to $k + 1$ disks because of the replication combined with the 1-sharing property. Suppose $k + 1$ disks fail in such a system. Each of the $k + 1$ failed disks shared superchunks with at most $k$ other failed disks, guaranteeing that the at most $k$ superchunks from each single disk are lost from the system. The $n + k$ erasure codes stored on each set of Lstors can be used to recover the lost superchunks by accessing the other superchunks in the stripe, which are guaranteed to have a replica left in the system due to 1-sharing.

It is important to note that for all failure scenarios, recovery in RAIDP is not
complete until 1-mirroring is restored across the cluster, meaning that the superchunk reconstruction is only the first step. Upon completing the reconstruction, the system performs a standard replicated recovery like any other replicating system: the non-redundant superchunk data is copied to other disks. The system has recovered when 1-mirroring is established in a manner that also maintains 1-sharing.

**Coping with Bad Sectors** Another attractive property of RAIDP is that it copes with occasional bad sectors. Assume that a system with a single Lstor on each disk suffers a dual disk failure. The system is now busy reconstructing the lost superchunk and replicating non-redundant superchunks to restore 1-mirroring. Further assume that RAIDP encounters a bad sector in a non-redundant superchunk used in the recovery. RAIDP can then utilize the parity information on the local Lstor to reconstruct the bad sector with locally available disk data. This property is similar to the intra-disk parity scheme proposed in [DEH+08].

Given the offset within the superchunk of the bad sectors, the data at the same offset from the parity and other superchunks is used to compute the lost data. Suppose, for example, that a device has an 8kB region of bad sectors in one superchunk and there are 100 superchunks on the device. Given a <10ms seek time, the 100 seeks required to read from the other superchunks would take less than one second.

### 3.4 Crash Consistency

The main purpose of Lstors is to 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 about to be used in conjunction with the corresponding surviving _remote_ superchunks, which might have been updated prior to detecting the disks’ failures. Therefore, upon a simultaneous failure, the Lstor used for recovery must be synchronized with its remote superchunks. To achieve synchronicity, RAIDP utilizes a simple, append-only journal, which resides on the Lstor and is used for rolling the Lstor forward. The roll-forward procedure is similar to the canonical crash consistency protocol [PCA+14, MHL+92, PADAD05], apart from the addition of the old disk data in the journal record. This old disk data does not change the essence of the protocol. Instead, it is used only by remote machines without access to their local disks, as discussed below.

**Writing** In RAIDP, a write request consists of the first section in the JournalRec structure shown in Listing 3.1. Upon receiving a write request, the node allocates a JournalRec and populates its first section using the write request contents: (1) a globally unique, monotonically increasing serial number provided by the underlying system (shared between each pair of replicated writes); (2) the host ID of the mirroring node for
the write; (3) the local destination superchunk; (4) the offset of the write within the
superchunk (and hence within the Lstor); (5) the length of the soon-to-be-written data,
and (6) the new data itself. Additionally, the node populates the second section of the
JournalRec with the help of its locally residing information: (7) the old disk data, which
is read according to the given offset and length inside the specified superchunk; and
(8) the corresponding soon-to-be-written new Lstor parity, which is computed based on
the associated old disk data, new disk data, and old Lstor data. The now-full JournalRec
is checksummed, added to the journal (along with the checksum), and sync-ed. (Since
writes are two-way replicated in RAIDP, matching data for each incoming write reaches
two nodes and thus a pair of corresponding journal records are eventually created for
each write.1)

With the new journal record synced, the Lstor and disk can now be safely updated
because if a crash happens midway during the updates, the system would be able to
later recover by replaying the journal. RAIDP therefore writes and syncs the new data
and parity. After, it sends an acknowledgment (ack) message to the mirroring node
that consists of the serial of the write request. If the matching ack from the peer node
has already been received earlier, RAIDP clears the record from the journal. Otherwise,
clearing will be done later on, upon receiving the ack. RAIDP resends acks periodically,
in case they “get lost” in the network. We will explain below why this procedure is
ensured to terminate. We note that, by our measurements, journal records are cleared a
few tens of milliseconds after they arrive. The pseudo code of the write procedure is
outlined in Listing 3.2.

The write procedure includes locks on the disk and Lstor byte ranges so that they are
read and written in an atomic fashion, rather than being prone to interleaving/reordering
with other requests. Locking on a byte range is the most concurrency-friendly way of
ensuring this atomicity. In practice, a global lock for the Lstor and disk might be just

---

1 Note that, for correctness, the underlying distributed replicating system guarantees that replicated
writes to the same block are consistent. Meaning, if there are two writes to the same place, they arrive
to the two mirroring nodes in the same order.
write(Boolean replaying) {
    1. Lock range <offset,len> in superchunk & Lstor
    2. if(not replaying)
    3. Read range from superchunk & Lstor
    4. Populate/write/sync new JournalRec
    5. Update/sync superchunk & Lstor
    6. Unlock range
    7. Upon receiving matching ack, clear JournalRec
}

Listing 3.2: Write procedure. Boolean is set only upon recovery.

as, or even more, efficient because the disks perform requests serially. The lock prevents
the disk from reordering of outstanding requests. However, if blocks are on the order
of megabytes or tens of megabytes as is typically the case with HDFS, such potential
reordering has little value or even a negative effect. By design, the request is big enough
to amortize the cost of the associated seek.

**Transient Failure** Some nodes can resume execution after being temporarily unavailable, e.g., after an intermittent power failure, a node reboot, or completion of a rolling upgrade. Upon resuming execution, the node must renegotiate with a “master” node (e.g., the Namenode in HDFS), regarding which recovery procedure to employ: it is possible that the master will determine that the node should behave as it would for a cluster-wide power outage—namely swapping journal entries with other nodes to restore cluster-wide consistency, described below—or, if the cluster has had time to duplicate the unavailable node’s superchunk mirrors elsewhere, then the node can resume execution as if it were a new node added to the cluster. During a node’s downtime, the master informs its peers to stop sending periodic acks and replicates its superchunks as described further below.

**Power Outage** After a power outage, consistency must be restored across the Lstors and their remote superchunks, since some updates may have failed to propagate between nodes. Importantly, if a journal record appears in one journal, then RAIDP must ensure that the record will eventually appear in its peer’s journal as well. Thus, upon power restoration, each node forwards its journal records to the mirroring node. It thus concurrently receives journal records as well. Of course, for efficiency nodes can first exchange serial numbers of the maximum journal record for a given superchunk and only if needed exchange journal records. Journal records are forwarded without the newParity field because it corresponds to the local Lstor only.

After each node completes the transfer of its records, it broadcasts a message to signify the completion of its transfers. After all nodes have completed sending their journal records, each node has the records it needs to become consistent with the other nodes in the cluster. The node sorts its local and received journal records according to their serial, discarding any duplicates in favor of its local record. The sorting and deduplication leaves the sequence of journal records in the order that the requests would
resumeAfterPowerOutage() {
    1. Send each JournalRec to JournalRec.mirrorNode (omitting JournalRec.newParity)
    2. Broadcast completion of JournalRec distribution
    3. Wait until cluster-wide completion
    3. Sort all JournalRecs, discarding duplicates
    4. Replay sorted sequence of journal records, calculating new parities as necessary
}

Listing 3.3: Resuming execution after power outage.

have been executed had the power outage not occurred.

The node replays the remaining sorted sequence of records as follows. If a record has been received from a remote node and is not available locally, then the newParity field within the journal record is empty (because it would have been meaningless). The newParity is therefore calculated with newData and oldData taken from the record, and the parity taken from the local Lstor. With these fields, the journal record can be replayed. If the record was created locally, then all of the information needed to replay the record is already available. Thus, in both cases the disk and Lstor are updated via the record by executing as outlined in Listing 3.2, with the boolean “replaying” set accordingly. The pseudo code describing this power outage recovery is outlined in Listing 3.3.

Single Disk Failure When a single disk fails, we do not need its Lstor. But the Lstor might be needed in case another disk fails shortly after. Thus, we need to have the Lstor protect new data written to superchunks on the failed drive until the single failure recovery is completed. The problem is that the Lstor can no longer update itself after its corresponding disk fails: it lacks a local disk from which to read oldData to contrast with oldParity and newData. It therefore cannot compute a new XOR, let alone populate a full journal record. For this reason, upon a disk failure, the node cannot reach Step 6 in Listing 3.2 and therefore the node stops sending corresponding acks, instead only accumulating partial journal records (that lack oldData). Because the peers of the node with the failed disk do not receive acks for those write requests, they also accumulate journal records. To limit the accumulation of journal records, the system, upon failure detection, diverts write traffic away from any non-redundant superchunks in a best effort manner. (It can only do so for new, rather than existing, incoming writes.)

Meanwhile, the distributed storage system replicates the surviving non-redundant superchunks as usual. Let S be such a superchunk. Let \( N_{\text{dead}} \), \( N_{\text{old}} \), and \( N_{\text{new}} \) be the node with the failed disk, the node holding surviving superchunk, and the node with the new mirror of the superchunk, respectively. When \( N_{\text{new}} \) completes the replication of S, S is no longer at risk of being lost due to a second disk failure. Additionally, subsequent writes are now being journaled on \( N_{\text{old}} \) and \( N_{\text{new}} \). Thus, the Lstor on \( N_{\text{dead}} \) can be “decommissioned” for that superchunk. At this point, \( N_{\text{new}} \) and \( N_{\text{old}} \) drop the “zombie” journal records associated with \( N_{\text{dead}} \). The pseudo code describing how to accommodate
singleDiskFailure(Lstor jfail) {
    // jfail represents the Lstor with the failed disk
    1. jfail stops sending acks
    2. jfail accumulates partial JournalRecs (no oldData)
    3. Peers write data, accumulate complete JournalRecs
    4. System replicates surviving each superchunk S
    5. N_old and N_new drop zombie JournalRecs of N_dead
    6. N_dead drops JournalRecs relating to S
}

Listing 3.4: Recovering from a single disk failure.

Second (Simultaneous) Disk Failure  
A superchunk is lost in the event of a second simultaneous failure. So if the disk on N_old dies, the system must recover S using parity information. There are two Lstor parities that the system can use in order to recover: that of N_dead and N_old. The system chooses one. Without loss of generality, let us assume it is the Lstor of N_dead.

We define the “recovery series” as consisting of the following information: (1) the parity residing on the Lstor of N_dead; (2) the journal records of N_dead; (3) all of the surviving superchunks that resided on N_dead and the journal records associated with these superchunks. As soon as the failure is detected, the system stops sending new writes to the superchunks in the recovery series. Incoming writes that were triggered before the simultaneous failure was detected become part of the recovery series, affecting both journal records and superchunks.

After all earlier writes are added to the series, the series is “sealed” and is sent to a recovery node, which will be one of two new mirroring nodes for the recovered S. The recovery node sorts the journal records of N_dead for all of the remotely-received superchunks by the serial number. The recovery node replays those records on the parity of N_dead. If a matching journal record is available from N_dead with a populated newParity, then newParity is written to Lstor parity of N_dead that is being rolled forward. This action is required because the processing of the corresponding journal record may have been interrupted by a crash and rewriting the newParity in sorted order is idempotent. Otherwise the new parity is calculated and written based upon the JournalRec.oldData, JournalRec.newData, and the current parity on Lstor of N_dead.

At this point, the parity of N_dead is consistent with the surviving superchunks, all of which are available locally on the recovery node. The S that was stored on N_dead is reconstructed in memory, and written into a new physical location in the superchunk according to Listing 3.2 as if it were a new incoming write. Some journal records may relate to S. Such records correspond to the journals of N_dead and N_old. The recovery only uses the former. Therefore, the recovery can be completed even if the Lstor of N_old is unavailable. If there are journal records for S, those records are also executed in serial number order according to Listing 3.2, using the newly recovered S for the oldData required by the write sequence. This process rolls S forward according to any writes
that were interrupted locally or occurred on \( N_{old} \) but not on \( N_{dead} \).

**Avoiding Accumulation of Journal Updates** Under normal operating conditions, replicating nodes exchange acknowledgments so that journal records can be cleared. However, if one node has a disk failure then any journal records for non-redundant superchunks will not be acked and thus accumulate.

To avoid this problem, peer Lstors send the still-journaled old disk data to nodes with disk failures, such that those nodes can fully populate their journal records. Those nodes will not be able to update their disks (as they have failed), however, they will be able to update their parity data and send acknowledgments, thereby allowing both the local and remote journals to be cleared.

**Garbage Collection and Journal Size** Log-structured file systems offer sequentiality and are well suited for the append-only writes desired by the RAIDP journal [SBMS93]. In principle, the Lstor does not require a full fledged log file system. A simpler alternative will do, namely structuring the journal as a cyclic ring buffer with a head and tail pointer. Journal records are appended to the head of the buffer, and under ideal conditions the tail of the buffer continues to advance as journal entries are garbage collected. The buffer can be appended to so long as the head does not advance past the tail. In our experiments, we observed that 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 under normal operating conditions.
Chapter 4

Replication vs. Erasure Codes vs. RAIDP

RAIDP encompasses tradeoffs from both replicated and erasure coded storage. RAIDP has properties of replicated systems that make it suitable for warm data, and it uses erasure codes on the Lstors to increase failure tolerance. In this section, we explore these tradeoffs in detail. We focus the discussion on configurations that can tolerate two simultaneous failures, which is the typical failure tolerance employed for warm data [GGL03, B. 11, NEF+12, Bor08, SAP+13, KBP+12, FTXG09]. Accordingly, Table 4.1 summarizes the tradeoffs between triple replication, \( n + 2 \) erasure coding with two parity blocks, and RAIDP configured with a single Lstor associated with each disk. For example, the first line in the table indicates that all configurations tolerate two simultaneous disk failures. Although we focus on these configurations, the table remains the same for setups that tolerate additional failures, with one exception highlighted later. We now address the items listed in Table 4.1 in order.

**Capacity**  Erasure coding systems provide the best storage capacity, requiring less than a 100% storage overhead whenever \( n > k \) in an \( n + k \) stripe. For example, a Reed-Solomon 10 + 2 erasure code can survive two simultaneous failures with an overhead of 2/10, or 20%. Triplication stores three copies of every data unit, so it has a 200% overhead. RAIDP stores only two copies plus the storage on the Lstors. Namely, its overhead is 100% due to replication plus \( S/N \) per disk for a system of \( N \) disks of a size \( S \), due to the Lstors.

**Repair Traffic**  A single disk failure is by far the most common failure scenario [RSG+13, KBP+12], and RAIDP recovers from it using the same repair procedure and disk/network traffic as that of a triplicated system. Namely, both redundancy schemes make direct copies of the under-replicated blocks using a surviving replica. In particular, assuming a disk size of \( d \) bytes, the recovery procedure for a single failed disk in both systems requires reading \( d \) bytes to get the content of the failing disk, transmitting these \( d \) bytes to other disks in a declustered manner, and then writing the \( d \) bytes on the
<table>
<thead>
<tr>
<th>properties</th>
<th>3 replicas</th>
<th>raidp</th>
</tr>
</thead>
<tbody>
<tr>
<td>survives dual disk failures</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>storage capacity</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>repair traffic (network and disk)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>single failure</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>dual failure</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>support for updates in place</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>cpu consumption (sync latency)</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>disk sequentiality</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>write—network</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub-stripe (small write)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>full stripe (large write)</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>write—disk</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub-sector (small write)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>sub-block (medium write)</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>multi-block (large write)</td>
<td>±</td>
<td>+</td>
</tr>
<tr>
<td>read</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>parallelism</td>
<td></td>
<td></td>
</tr>
<tr>
<td>load balancing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>degraded read (temp. unavailable data)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 4.1:** 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.

<table>
<thead>
<tr>
<th>failure</th>
<th>I/O</th>
<th>3 replicas</th>
<th>raidp</th>
<th>erasure n + 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>single</td>
<td>disk</td>
<td>2d</td>
<td>2d</td>
<td>(n + 1) · d</td>
</tr>
<tr>
<td>network</td>
<td></td>
<td>1d</td>
<td>1d</td>
<td>(n + 1) · d</td>
</tr>
<tr>
<td>sum</td>
<td></td>
<td>3d</td>
<td>3d</td>
<td>(2n + 2) · d</td>
</tr>
<tr>
<td>double</td>
<td>disk</td>
<td>4d</td>
<td>4d</td>
<td>(n + 2) · d</td>
</tr>
<tr>
<td>network</td>
<td></td>
<td>2d</td>
<td>3d</td>
<td>(n + 2) · d</td>
</tr>
<tr>
<td>sum</td>
<td></td>
<td>6d</td>
<td>7d</td>
<td>(2n + 4) · d</td>
</tr>
</tbody>
</table>

**Table 4.2:** Repair traffic upon a single/double disk failure. RAIDP traffic is nearly identical to triplication, because superchunks are significantly smaller than disks. Erasure coding, on the other hand, is $O(n)$ times worse.

The latter. The total is thus $d + d = 2d$ bytes of disk I/O and $d$ bytes of network traffic, as indicated in the top half of Table 4.2.

With triplication, the actions taken to recover from a double disk failure are similar, but the I/O traffic is doubled (bottom half of Table 4.2). The latter statement does not apply to RAIDP, which recovers differently from a dual failure of disks $D_1$ and $D_2$. 
Specifically, the surviving superchunks of, say, $D_1$ are used in conjunction with $D_1$’s Lstor to reconstruct the one lost superchunk $S_{\text{lost}}$. RAIDP induces the same repair I/O and network traffic as triplication to re-replicate the surviving superchunks from $D_1$ and $D_2$. But it additionally induces nearly $d$ bytes of network traffic to reassemble $D_1$’s mirrored superchunks on the recovery node when reconstructing $S_{\text{lost}}$. Therefore, in total, RAIDP induces $2d + 2d = 4d$ bytes of disk I/O for reading and then writing elsewhere the replicas of $D_1$ and $D_2$. RAIDP induces $2d$ bytes of network traffic for the associated transmission, plus another $d$ for reassembling $D_1$’s Lstor and surviving superchunks, yielding a total of $3d$ network traffic.

Notably, RAIDP reconstruction requires far fewer network/disk resources as compared to erasure coding. Assume an $n + 2$ encoding (the likes of Reed-Solomon) with a disk size of $n$ bytes. Erasure coding requires reading and transmitting the content of $n$ disks in their entirety in order to reconstruct $j$ failed disks ($j = 1, 2$), totaling in $n \cdot d$ bytes of disk I/O and network traffic. The reconstructed $j \cdot d$ bytes of the $j$ failed disks must then be transmitted and written to different disks in a declustered fashion. Thus, overall, erasure coding requires a total of $(n + j) \cdot d$ bytes of disk I/O and $(n + j) \cdot d$ bytes of network traffic to recover from $j$ failed disks.

Table 4.2 summarizes the repair traffic of the three redundancy schemes. We can see that RAIDP is nearly as efficient as triplication, whereas erasure coding is $O(n)$ times worse. Recall once more that the repair traffic associated with erasure coding can dramatically impact the performance runtime of foreground workloads and needs to be invoked on accesses during even transient failures. RAIDP does not suffer from this problem, a key factor in making it suitable for warm data.

Support for Updates in Place In any distributed storage system, updating data in place mandates that stripe consistency be recoverable in case a node experiences a failure during an update of a stripe. Such consistency can be maintained through logging or versioning mechanisms [GWGR04, AJ05, Bel13, CDLC14]. Replicating systems have the inherent advantage that once an update in place resides in the journal of a node, the update of the local replica can be initiated without depending on, or synchronizing with, other nodes. In the event of a failure resulting in a partial update, all of the replicas can be rolled forward based on the entry of a single journal.

Conversely, a record of a pending change residing on a journal is not enough for a change to commence in an erasure coding system. Rather, the initiation of the update must be coordinated across the $k$ parity nodes and all nodes in which updates are taking place. Consider the update of a single block. Only when the pending change resides in all the $k + 1$ journals can the initiation of the update take place. Comparably, RAIDP also requires $k$ (or rather, $k - 1$ for the same level of fault tolerance) synchronized Lstor journal updates. But the journals all reside on one node, ridding the system from the need to synchronize between $k + 1$ lock-stepped nodes.

Another advantage of a system that utilizes Lstors (RAIDP in our case) is that the
time it takes to perform an I/O on the journal is negligible in comparison to the time it takes to perform disk I/O. Thus, regardless of the number of journals associated with each disk, a RAIDP write operation requires a total of 4 disk I/O operations. In contrast, in a (Lstor-less) distributed erasure coding system, the equivalent update requires \(3\times(k+1)\) I/O operations, such that the “3” coefficient corresponds to (1) reading the old data to compute parity, (2) writing the new data to the journal, and then (3) writing the new data in place.

To conclude, we see the RAIDP journal as a potentially powerful optimization tool for supporting updates in place in distributed storage systems. Updates in place are easier to implement in replicated / RAIDP systems in comparison to distributed erasure coding systems. In all redundancy schemes, Lstors could be used to accelerate the process, as they constitute a modular, high-performance journal. In Table 4.1, in the “support” line corresponding to the current discussion, we assign “+” to RAIDP because it is already supported in systems such as Ceph [vS14] and GlusterFS [Bel13]; we assigned “-” to erasure coding because of its aforementioned complexity and since, to our knowledge, no production distributed erasure coding system supports it; and we assigned “±” to RAIDP because we could not evaluate the functionality of the feature, as we were limited by the distributed storage system used for our implementation (HDFS), whose interface does not permit updates in place.

**CPU and Sync Latency** Erasure coding is the worst offender in terms of CPU and sync latency. As noted in §2, if the parity is not known a priori, then the parity must be calculated with all blocks accumulated in the stripe—a CPU-intensive task—and only then can the parity data be distributed and the entire stripe fully synced. For example, in a 10+2 stripe in erasure coding, the system must wait for all 10 blocks before finalizing the calculation of the parity blocks and distributing them. RAIDP also calculates parities, but does not have to wait for a requisite number of blocks before sending an ack to a client since blocks are independent. Each block (or bit for that matter) is replicated as it arrives, and the parity on the Lstor can be updated independent of the mirroring node or any other blocks. We consider triplication as the best in terms of sync latency and CPU utilization because, like RAIDP, data is replicated as it arrives, and unlike RAIDP, triplication requires no parity calculations.

**Sequentiality** Both triplicated systems and RAIDP promote sequentiality on disk because data is replicated and written to disk in its entirety. In contrast, erasure coding splits a piece of data up into \(n\) blocks that have \(1/n\)-th the size as part of a stripe. If this piece of data is small—a possible outcome of trying to reduce sync latency—then performance when writing and subsequently reading the individual \(1/n\)-th smaller blocks may suffer due to the increased disk seeking caused by the reduced sequentiality of the data on disk.

**Network Traffic for Writes** In this and the following paragraphs, we discuss disk and network write traffic. When we refer to operations that write less than a full stripe
(i.e., sub-stripes and sub-sectors), we assume that the system supports updates in place.

When writing less than a stripe, erasure coding encounters the small write problem [FTXG09]. On the network, these types of writes involve the incoming deltas being sent to the nodes maintaining the parities so that the parities can incorporate the newly updated data. In an \( n + 2 \) erasure coded stripe, assume \( b \) bytes are received from a client (where \( b \) is less than a full stripe). Then, \( 2b \) bytes of additional traffic are created to update the parities, for a total of \( 3b \) bytes. Replication creates \( 3b \) bytes of traffic, because the \( b \) bytes are replicated on three nodes. RAIDP performs the best, inducing \( 2b \) bytes of network traffic since only two replicas are created and all parity changes are local.

Conversely, when writing a full stripe, erasure coding performs the best since the network demands reduce to spreading the blocks of an \( n + k \) stripe, where the \( k \) parity blocks are already computed and just need to be stored. If the client writes \( b \) bytes for the data portion of a full stripe, then the total network bandwidth would be, in the case of a \( 10 + 2 \) erasure code, \( 1.2b \). Or more generally, \( b + 2 \frac{k}{n} \) in the case of a \( n + 2 \) stripe. Triplication and RAIDP create the same network traffic for a full stripe as they do for a sub-stripe: \( 3b \) and \( 2b \), respectively. Table 4.3 summarizes the network traffic for all three redundancy schemes.

**Disk Traffic for Writes**

In the following discussion, when comparing RAIDP to other systems we do not include the I/O related to reading and writing the parity on the Lstor or writing to the journal, as they should not affect performance. As described in §3, we assume that reads and writes to the parity are done in parallel with disk accesses and performed at least as fast. When writing to the journal, we assume that it is done at a high enough bandwidth so as to not affect performance. We now describe the traffic induced by different types of writes.

Writing less than a sector (referred to as “sub-sector” in Table 4.1) requires writing the entire sector anew, even if the change is as small as modifying one byte. Thus, the entire sector must first be read before it is rewritten with the sub-sector change. Triplication and \( n + 2 \) erasure coding each require applying read-modify-writes on three sectors, yielding six operations altogether: for triplication, the read-modify-write is applied to three replicas; and for erasure coding, the read-modify-write is applied to one data sector and two parity sectors. Conversely, RAIDP has only two replicas, so it only performs two sector read-modify-writes for a total of four sector operations.

Let us next consider the disk traffic associated with writing a “sub-block”, which means in the current context that an integral number \( s \) of sectors is being written, such that \( s \) is less than the number of sectors in a block. For erasure coding systems, this type of write induces \( s \) reads to get the old data (so as to be able to compute the new parity) and \( s \) writes of the new data. In addition, to update the parity in an \( n + 2 \) erasure code, there are \( 2s \) reads of old parity data and \( 2s \) writes of newly calculated parity data. The total is \( 6s \) sectors of traffic.
In a triplicated system, the data is replicated as-is, thus writing a sub-stripe of \( s \) sectors entails writing a total of \( 3s \) sectors. RAIDP performs in between erasure coding and triplication—only two replicas of the sub-stripe are created, but before writing each replica the old data is read first in order to update the Lstor parities. Thus, the read-before-write in RAIDP induces \( 4s \) sectors of disk traffic.

Writing multiple blocks of data (\( t \) blocks) in triplication still induces \( 3t \) blocks of disk traffic throughout the system, one for each replica. Erasure codes in most cases induce the least disk traffic. For sub-stripe writes, the traffic involves reading and writing the original data blocks plus two additional parity blocks for a total of \( 2t + 4 \). If the integral number of blocks modified is greater than two, both triplication and RAIDP require a greater number of I/Os. In RAIDP, we always need to perform two read-modify-writes, regardless of whether the write spans multiple blocks. Thus, RAIDP results in \( 4t \) blocks of traffic.

For full stripe writes, erasure coding can save the reads prior to writing, only writing the original \( n \) data blocks in a full stripe, plus two additional parities. RAIDP, by contrast, needs to execute a read-modify-write cycle for each of the two replicas, resulting in a disk traffic of \( 4n \) blocks. Triplication requires \( n \) blocks for every replica, resulting in \( 3n \). This holds for multi-block writes that are both sub-stripes and full stripes.

With respect to disk I/O, RAIDP is inferior to triplication for everything other than sub-sector writes. Namely, the four disk I/O operations (two reads and two writes) constitute a 33% increase as compared to the three write operations required for the same workloads in a triplicated system. Importantly, however, this disadvantage only applies under the current single-Lstor/triplicated configurations, because the four disk I/O operations for a RAIDP write operation remain constant regardless of the number of Lstors for each disk. That is, if a system requires a higher level of failure tolerance, the number of Lstors increases without inducing additional disk I/O. Thus, RAIDP with two Lstors would have the same number of disk I/O operations as compared to a four-way replicated system (four I/O ops in RAIDP vs. four I/O ops in replicated). And RAIDP will induce fewer I/O ops relative to replication with three Lstors as compared to a five-way replicated system (four in RAIDP vs. five replicated), and so on.

Further, as mentioned in §3, RAIDP can reduce its disk I/O traffic by reading the old disk data only once and transmitting it to the mirroring node. The transmission of the old data reduces the number of I/Os in RAIDP from four to three, therefore on par with triplication in terms of disk I/O, but at the cost of additional network traffic.

Table 4.3 summarizes the write traffic of the three redundancy schemes. On the network, RAIDP is the best for sub-stripes and in between for full stripes. On the disk, RAIDP is the best for sub-sectors, in the middle for sub-blocks, and creates the most disk traffic for multiples of blocks up to a full stripe writes. Recall once more, however, that the RAIDP disk traffic is on par with triplication when transmitting the old superchunk data between nodes, and can be reduced relative to replication when increasing the failure tolerance of each system.
I/O 3 raidp erasure

<table>
<thead>
<tr>
<th></th>
<th>replicas</th>
<th>n+2</th>
</tr>
</thead>
<tbody>
<tr>
<td>write—network</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub-stripe</td>
<td>3b</td>
<td>3b</td>
</tr>
<tr>
<td>full stripe</td>
<td>3b 2b b+2b</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3b 2b b+2b</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>3s 4s 6s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3t 4t 2t+4</td>
<td></td>
</tr>
<tr>
<td>full-stripe</td>
<td>3n 4n n+2</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.3: Write traffic on the network and disks. b is number of bytes, s is number of sectors, t is number of blocks, and n is the number of data blocks in a stripe.

**Read Performance**  RAIDP and triplication can use alternate replicas to improve read performance, either by reading from multiple nodes in parallel or directing traffic away from a replica on a burdened node to a different replica. Triplication can do better than RAIDP in this regard, because triplication has an additional replica to read from. Both RAIDP and triplication can avoid the degraded reads of erasure coded systems by using an alternate replica—an advantage lacking in erasure coded systems. The advantages associated with replication for RAIDP contribute again to its suitability for warm data.

**Failure Domains**  As mentioned in §3.2, we assume that disks fail separately from their Lstors. But in practice, Lstors and their disks might not always fail separately. For example, a failure of an entire rack would render both a disk and its Lstor inaccessible. Thus, RAIDP is inferior to triplication and erasure coding systems in this area because those systems can spread different parts of a (replicated or erasure coded) stripe over more than just two failure domains (where the failure domain here refers to the rack).

Recall, however, that there are two Lstors which may be used to recover from a dual disk failure. Thus, if two disks located on different racks have failed and only one rack is inaccessible, then the Lstor on the other rack can be used to recover the lost superchunk. This alternate means of recovering from a correlated failure of a disk and its associated Lstor is why we assign “±” to RAIDP in the “failure domains” row in Table 4.1, signifying that RAIDP is better than a replicating system that employs two replicas only.

We note in passing that RAIDP and default HDFS provide a similar level of failure tolerance with regards to rack failures, because the default replica placement in three-way replicated HDFS also only spans two racks. The first replica is placed on a node in one rack, and the second and third replicas are placed on two nodes in the same rack [ASF]. Like HDFS, RAIDP can recover from a double disk failure so long as one rack, and with it the Lstor for the failed disk, is available.
Chapter 5

Implementation

We implement RAIDP within HDFS (Hadoop Distributed File System) with two replicas. Our reasons for selecting HDFS are that it is open source, packaged with standard benchmarks for use in our evaluation, and well-documented in past systems research.

The RAIDP implementation uses a single Lstor per disk simulated in DRAM. HDFS is roughly \( \approx 60\text{K} \) lines of code, and the RAIDP patch is another \( \approx 3\text{K} \) lines.

Superimposing Superchunks on HDFS  

Like other distributed storage systems, HDFS stores data in “blocks”. As noted, the size of these ranges between single to hundreds of megabytes [B. 11, ASF, NEF+12, SAP+13]. In HDFS, the default block size is 64MB, and every block is stored as an ordinary data file (plus an additional associated checksum file). The “ID” of the block is the name of the file. HDFS supports smaller size files in case clients wish to store smaller data items, but RAIDP supports only 64MB to simplify our implementation.

The HDFS architecture has a “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. The client HDFS library then begins transferring data to the first Datanode (which we call the “primary”), which forwards the data to the mirroring Datanode (which we call the “secondary”). Thus, for each block, the identity of the two mirroring nodes is readily available for both and is used by RAIDP to identify the destination superchunk.

Let H be the absolute path of the directory on each node’s disk under which all HDFS blocks reside. (For example, H could be “/hdfs/”). To avoid having many files under a single directory, HDFS saves block files into random subdirectories under H. Introducing superchunks to HDFS is therefore a matter of simply replacing a random subdirectory under H with a directory named for the superchunk, and ensuring that blocks shared by a pair of Datanodes are saved to that directory on both nodes.

Given a block that is mirrored on Datanodes \( A_i \) and \( A_j \) (where \( i < j \)), RAIDP replaces the corresponding random subdirectory with a the superchunk directory \( H/A_{i,j} / \), thus ensuring that block files reside within their designated superchunk directories.
Our assumption that all blocks are uniformly sized allows us to manage the content of the superchunk using a bitmap whereby the setting of a bit represents allocating the corresponding block and unsetting the bit represents deleting it. Upon startup, bitmaps are initiated to zero, which for XORing purposes, implies that the corresponding block bits are zero.

Mirrored pairs of superchunks need to be logically bitwise identical, requiring a consistent and well-defined block order. We thus let names of blocks dictate their ordering within the superchunk, such that the $k$-th block in the superchunk corresponds to the file $H/A_{i,j}/k$. The “offset” field of the superchunk of Listing 3.1 can be easily deduced from $k$.

**Maintaining Local Parity**  Now that we have a superchunks layout on a Datanode, our goal is to extend HDFS such that the the parity of the superchunks is stored on the Lstor. Going over the entire massive codebase of HDFS to incorporate superchunks and Lstor parity logic might have been a painful, excruciating task. Thankfully, all HDFS data accesses are conducted via a unified file system interface (called FSDatasetInterface). The manner by which this interface is implemented is transparent to the rest of HDFS, making the required RAIDP modifications relatively straightforward.

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. In our implementation we use tmpfs in DRAM to store the parity and journal information. The kernel constrains the size of tmpfs to be half the DRAM size.

The heart of the parity maintenance logic resides in our reimplementation of the aforementioned interface, utilizing polymorphism to maintain the interface contract. When writing a block to a superchunk at a given offset, RAIDP also updates the parity at this same offset.

In more detail, the central method of the interface (also 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 read-modify-write functionality required for RAIDP. (We also maintain a parity for the checksum data across the superchunks.)

**Optimizations**  RAIDP synchronizes data to the disk and its Lstor according to the protocols specified in §3. We discovered that doing so in a naive manner induces performance degradation, as explained next. 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 the block. We note that in the baseline 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 despite what a client is told. To remedy this, we add a sync statement to both RAIDP and the baseline version of HDFS for our evaluation. We are encouraged to see that in subsequent versions, a sync statement has been added [Kua12].
the data to the disk. RAIDP, however, cannot afford to wait that long. It is more constrained because every write is synced to the disk and journal (Listing 3.2). But synchronizing every 64kB cripples performance, as shown in §6.2.

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. By itself, however, accumulating is not enough due to the following reason.

A Datanode may write multiple blocks (64MB files) concurrently. When the local filesystem is instructed to write several 64MB files in parallel, baseline HDFS writes in an append-only manner, interleaving the files in order to write sequentially. Later, when these files are deleted, the sectors they occupy will be deallocated by the underlying local filesystem and be used anew upon subsequent reallocations. This is not the case for RAIDP in our implementation. Once sectors have been associated with a file (64MB block), this association persists throughout (de)allocations, because a superchunk is an XORable entity and block \( k \) always retains its offset within its encapsulating superchunk. Therefore, for blocks that were previously allocated non-concurrently in a contiguous manner, subsequent concurrent writes might create a “ping pong” effect, whereby the read/write head of the spinning drive moves back and forth between the associated files.

To combat such useless seeking and enforce sequential I/O, we introduce a locking mechanism in tandem with the aforementioned accumulation, as follows. When a block file has fully accumulated, the thread designated for writing the block to disk acquires a lock, thereby preventing other threads from interrupting by writing to disk concurrently. These tandem optimizations largely eliminate this performance degradation due to syncing and seeking, as demonstrated in §6.3.

**Issues With A More Realistic Implementation**

Our assumption that all blocks are 64MB greatly simplifies our prototype, but it might not be appropriate for all setups. In a real system, to eliminate this assumption, we will need to manage the storage within superchunks by handling arbitrary allocations and deletions of block files of different sizes—which in essence is what local filesystems do. That is, we will need to deploy a filesystem on each superchunk. This can be done, for example, by structuring each superchunk as a partition and formatting it as a standalone standard filesystem, such as ext4, however doing so brings about many other challenges.

Importantly, a pair of mirroring superchunks must still be bitwise identical. We hypothesized that such synchronicity will be achieved by performing the same operations on a pair superchunks in the same order. But experimentation proved us wrong.

Upon investigation, we found the file metadata can be different across the replicas, i.e., the atime, mtime, and ctime, might be different due to time transpiring between block replication. We eliminated the atime problem by disabling it upon mounting the superchunk partitions. We eliminated the mtime problem by setting it explicitly to be
the same value on the mirrored superchunks via the utime system call [Lin08]. We were unable to resolve ctime differences in a similar manner, because ctime is not a settable attribute. We therefore resorted to patching the Linux kernel, such that the ctime is set to be the mtime whenever the latter is specified [Ros13]. (We applied these modifications to ext2 [Lin14] because it is simpler than the alternatives.)

Dealing with timestamp differences, however, was not enough. We discovered that, by design, there are various sources of randomness within the filesystem. This randomness manifests in the same files being allocated in different locations across the pair of mirroring superchunks and in a different internal filesystem representation. We looked into the factors providing randomness, and identified three: the pid, and two internal fields in the ext2 filesystem: inode group and s_next_generation. After replacing those non-seeded random values with seeded ones, the filesystem behaved identically to its mirrored counterpart. We were able to mount the superchunk partitions to the aforementioned HDFS data directory on each node, and see that two mirroring superchunks were bitwise identical upon executing the same file operations in the same order.

The remaining requirement for RAIDP was that we update the local parity for every write to a superchunk. We attempted to use BUSE (Block Device in Userspace) [Coz13] with each superchunk partition, which would allow us to proxy operations on the underlying block device in userspace. Before forwarding writes to the block device, our implementation would intercept the incoming data, length, and offset within the filesystem for each write, and make the corresponding update to the parity. While we achieved a functional setup, we were unable to use BUSE because it does not follow sync semantics, precluding us from using this entire software stack.
Chapter 6

Evaluation

6.1 Methodology

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 Corporation NetXtreme II BCM57810 and a 1Gbps Broadcom Corporation NetXtreme II BCM5716. All of the nodes connect to a switch in a star topology via both NICs. Each node runs Ubuntu 14.04 with the 3.13.0 Linux kernel, using the ext4 filesystem.

We implemented RAIDP in Hadoop 1.0.4 using a 6GB superchunk size. We compare our RAIDP implementation with two-way replicating and triplicating HDFS, referred to as HDFS-2 and HDFS-3 respectively. We perform the evaluation with Hadoop’s default configuration. Most notably, this configuration includes the HDFS block size of 64MB and a maximum of two simultaneous map tasks and two simultaneous reduce tasks for each data node.

In our evaluation of RAIDP, we preallocate the block files in every superchunk to induce the overhead of read-modify-writes. Otherwise, superchunk updates will return logical zeros without actually accessing the disk, thus hiding the cost of the read.

We use standard benchmarks provided with the Hadoop distribution, measuring each result five times and presenting the average. The TeraSort and Wordcount benchmarks are executed via Intel’s HiBench suite [Int15]. Prior to running each RAIDP workload, all of the caches were cleared on each node to ensure that the read I/Os would reach the disk.

Measuring CPU and I/O Statistics

`iostat` [Lin13a] reports CPU statistics and I/O statistics for devices. We use iostat to measure disk and CPU utilization, average I/O request size, and read and write bandwidth on the disk.

`/sys/block/<dev>/stat` [Lin] is a file which reports a variety of block layer statistics for the device specified in place of “<dev>”. We query this file before and after running benchmarks to determine the I/O that each node performed during the benchmark. These totals are useful in determining how Hadoop is able to load balance the work done
throughout the cluster.

**Measuring Seek Distances**  
blktrace [Lin07b], blkparse [Lin07a], and btt [Lin07c] are the trio of utilities that we use to measure disk seeks. Prior to running a benchmark, we initiate blktrace on each node to record traces of the I/O and event traffic from the kernel for each CPU. (In our cluster, there are 4 CPUs per node.) After terminating blktrace on each node, blkparse was performed on the traces to combine the traces into one. Finally, btt was used to process the blkparse output into a concise summary of the total seeks, seek distance, and I/Os that were performed during the benchmark [Bru07].

**Measuring Network Bandwidth**  
We measure network bandwidth using the data from /proc/net/dev. This file contains the up-to-date number of bytes sent and received for each Ethernet interface. We query this file before and after running benchmarks to determine the amount of network bandwidth used during the workload [Lin13b].

### 6.2 Sync Overhead

In 3.4, we perform journal accounting in resolution of the incoming writes, where each write must be synced to both the journal (fast) to the disk (potentially slow). In practice, we have the flexibility to adjust the granularity of the journal accounting anywhere between the size of incoming writes in HDFS, which is 64kB, to the size of the HDFS block, which is 64MB.

We would expect that smaller writes would potentially propagate to disks on the same rotational track, thus only incurring rotational latency with the tradeoff of causing more syncs due to the write procedure in Listing 3.2. In contrast, journaling at a coarser granularity induces fewer syncs, but may cause additional movement of the read/write head in order to reach on a different track on the disk.
Given these considerations, we now consider the question of how much data should be journaled at a time: should the chunks be smaller or bigger? We answer this question by conducting the following experiment. We run two different workloads: writing 6GB and reading before writing 6GB. We vary the resolution at which the disk I/O operations and syncs are performed in powers of 2, ranging from 64kB to 64MB. The results are shown in Figure 6.1. The x axis shows the progression of the I/O operation size and sync interval. The left y axis shows the throughput of writing and reading before writing as represented by the two lines. The right y axis applies to the bars, which show the relative throughput of reading before writing as compared to only writing. (The values represent read before write throughput divided by the throughput of only writing).

The overhead of reading before writing is the smallest in the left side of the figure. But when the syncs occur at a fine grain resolution, the disk syncs dominate the workload and therefore minimize the overall throughput. As we move to the right with coarser-grained syncs, the throughput monotonically improves. Despite the increased relative penalty of reading before writing (as indicated by the declining gray bars), the throughput is the highest for both workloads at 64MB with an overhead of 1.8x for reading before writing. Our conclusion is that journaling at a resolution of 64MB, as indicated in §5, is the most advantageous despite having the highest overhead.

Note that this 1.8x overhead does not directly translate to the overhead of using RAIDP. It demonstrates the effects of the extra I/O on a single machine, not accounting for Hadoop’s overheads which amortize the cost of reading before writing, and also not accounting for the comparison of RAIDP versus HDFS-3. As discussed in §4, for every unit of data written, HDFS-3 produces three I/Os in the cluster. RAIDP produces a total of four: two reads and two writes. The transition from three I/Os to four I/Os is an increase of 33% in the amount of total I/O, whereas in the case of the microbenchmark we only quantify the impact of going from one I/O to two (an increase of 100%). Therefore, the 33% increase in I/O informs our worst-case expectations regarding the cost of the extra read-modify-write.

6.3 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 interferes with the ability of the local filesystem to perform such serialization. The local filesystem is unable to perform the serialization because the superchunks are pre-existing, and writes originating from different nodes must be stored in different superchunks since each superchunk is 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 performance penalty of random I/O.

RAIDP largely eliminates this penalty by employing the two optimizations outlined in §5. Namely, it buffers the entire incoming block in memory, writing 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.

Figure 6.2 depicts the performance of employing the superchunk layout with and without the aforementioned optimizations as compared to baseline HDFS, running the standard HDFS TestDFSIO benchmark configured to write 100GB. Notably, a benchmark that only performs writes constitutes the worst-case for RAIDP in terms of performance.

The left and middle subfigures pertain to the RAIDP configurations that forgo and employ these optimizations, respectively. Namely, in the unoptimized setup, RAIDP works with a write resolution size of 64kB (corresponding to the HDFS’s default “packet” size). And 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 superchunk layout takes effect without parity updates nor reading before writing. The middle bar in the leftmost subfigure adds the overhead of reading before writing and updating the parity, though with no journaling. Counterintuitively, reading before writing improves performance. To investigate this result, we took a closer look at the disk, pictured in Figure 6.3. The x axes show a short interval while running the write benchmarks. The left y axis shows the disk throughput and corresponds to the solid curves. The right y axis shows the disk utilization corresponding to the dashed curves. Without reading before writing, the disk utilization occasionally skyrocket while the disk throughput remains low, suggesting that the disk is being used inefficiently. When adding the read before write and parity updates, this phenomenon disappears.

Figure 6.4 sheds further light on this counterintuitive result. On the left, it shows the seek distance of the two RAIDP setups, highlighting that reading before writing actually induces fewer seeks, despite the disk performing twice the I/O. The subfigure to
Figure 6.3: Disk utilization is more efficient with the additional read before writing.

Figure 6.4: RAIDP seek distance and I/O request size with and without reads before writes.

the right explains this surprising result, which is that the extra read I/O results in the I/O scheduler aggregating more I/O requests, yielding 2.2x larger requests on average. Thus, having more requests in the I/O queue that are in close proximity to each other (as is the case with the read-before-write) apparently allows the scheduler to merge I/O requests more effectively. Larger I/O requests and reduced disk seeks result in improved performance.

Returning to Figure 6.2, the third bar in the leftmost subfigure adds journaling to the RAIDP variant that reads before it writes. It demonstrates a result that is off the chart, which is due to the default packet resolution. Recall that, by Listing 3.2, the journal write sequence causes a disk sync after each transaction. This result mirrors the poor throughput exhibited at the left of Figure 6.1.

The optimized results for RAIDP perform much better (middle of Figure 6.2), with the “only superchunks” setup achieving performance on par with HDFS-2 (right of Figure 6.2). We conclude that our optimizations are successful in eliminating any overhead associated with the superchunk layout. Comparing the read before write setup to tripling HDFS shows that the former is 20% slower, which is less than the 33% upper-bound caused by having four I/Os rather than three (§4). Adding the journal worsens performance somewhat.

6.4 Reading

To evaluate the read performance of RAIDP we run a 100GB TestDFSIO benchmark that reads the data written previously by TestDFSIO (in §6.3). Figure 6.5 shows the results
are uniform across the different configurations. RAIDP experiences a small overhead. We find that this is due to the preallocated superchunk layout, which causes a slightly smaller average I/O request size with RAIDP (not shown).

The really interesting result of the read benchmark does not appear in Figure 6.5. Rather, it arises from comparing it to Figure 6.2, which reveals that the runtime of reading 100GB (roughly 4 minutes) is nearly the same as the runtime for writing 100GB in the optimized superchunks-only and HDFS-2 configurations pictured (middle and right of Figure 6.2). This result does not make sense, because the amount of I/O produced by writing is twice that produced by reading, due to the replication (200GB for writing vs. 100GB for reading). We remark that the optimized superchunks-only RAIDP is equivalent to HDFS-2 in that both write two replicas and nothing else, so the fact that they themselves perform comparably 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 sequentially. Reading is done similarly, however with a 50/50 chance of reading from either replica. Whereas writes 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 substantially more disk seek activity.

This pathology is worsened as the size of the workload increases. Figure 6.6 demonstrates this finding with two workloads contrived to exaggerate the negative effect. The first consists of a varying number of 5GB files being written, and the second workload consists of exclusively reading the files written by the previous workload. The x axis denotes the total number files being written or read by the workload.

The bars in Figure 6.6 are associated with the right y axis, denoting the factor by which the write workload was longer than the read. The leftmost bar indicates that writing (5GB×2) takes twice as long as reading (5GB×1), which is the result we would expect. As we increase the number of files being written, this ratio monotonically drops until reading (5GB×128) becomes as slow as writing (5GB×128).

The two curves in the figure are associated with the left y axis that displays seek distance. The results confirm our explanation that reads induce significantly more seeks as compared to writes. In the rightmost workload, notice that reads induce twice the seeks with half the I/O as compared to writes, meaning that the seek distance per I/O is in fact 4 times longer.

We note in passing that there have been a number of optimizations to the read path on top of the HDFS version that we are using for our evaluation. One optimization in particular [McC14], enables a readahead feature that “allows the disk to make larger contiguous reads” for the expressed purpose of avoiding seeks. This patch “helps hide the latency of rotational media and send larger reads down to the device” [McC11].
Figure 6.5: RAIDP and HDFS read performance under different configurations.

Figure 6.6: HDFS reads degrade faster than writes due to increased seek distance.

6.5 Benchmark Performance

We summarize the results of the standard read (§6.4) and write (§6.3) HDFS benchmarks, and contrast them with two additional benchmarks. We now only use the optimized version of RAIDP and HDFS-3, which we contend represent the most fair comparison, as both tolerate two simultaneous disk failures. (We used the less optimized versions of RAIDP above merely to assist our analysis and 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 such that it replicates 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 Figure 6.7 (top) positioned near the results of the write benchmark to allow for easy comparison. Unsurprisingly, the performance penalty induced by RAIDP (13%) is less than that of only writing (24%). The reason is that TeraSort requires both read and write I/O as well as processing for sorting. Hence, the RAIDP read before write penalty is amortized over the time spent performing other work, which consists of reading and sorting, in addition to writing.
With both the TeraSort and write benchmarks, data is generated locally and then replicated based on the configured replication factor. In RAIDP, there is one additional replica and in HDFS-3 there are two. The lower replication factor of RAIDP results in the network traffic relative to HDFS-3 being halved, pictured on the bottom left of Figure 6.7. The total write network traffic is a bit more than 200GB for HDFS-3 because 100GB are written in total, such that each byte is generated on some local node and is replicated to two remote nodes. Similarly, RAIDP produces half the network traffic, namely 100GB, because only one additional replica of each byte is transmitted on the network. The network results when running TeraSort on RAIDP and HDFS-3 are qualitatively similar to those of writing. The input data is generated with TeraGen, which generates random data within an increasing range for each map task. Bounding the input data within ranges allows the sorting and output processes to be executed mostly locally, hence the relatively similar quantitative network results between writing and sorting.

The second benchmark is Wordcount, which computes string frequency over a 100GB input. Like in the case of TeraSort, the 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, resulting in negligible output data (on the order of kB). Thus, Wordcount is similar to the read benchmark in that its I/O is overwhelmingly comprised of reads. Wordcount is different in that it also includes a significant CPU processing component, accounting for the longer runtimes of Wordcount as compared the read benchmark (top right of Figure 6.7). Reads in RAIDP and HDFS-3 have an already small performance difference. With Wordcount, this difference is further masked by the CPU processing, resulting in the nearly identical runtimes of RAIDP and HDFS-3. Since both Wordcount and the read benchmark are comprised of the same amount of read operations, the quantitative (absolute) difference in their network volume is negligible (bottom right of Figure 6.7).

6.6 Superchunk Recovery

During a superchunk reconstruction, a recovery node initiates threads that request small chunks of superchunk or parity data from the relevant nodes in the cluster (members of the recovery series, as described in §3.4). Upon receiving each chunk of parity or superchunk data, the client threads XOR the data in memory and then move each fully-assembled block file to disk until the entire lost 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). Because several threads access and XOR the yet-to-be reconstructed block data
concurrently, each thread must lock on the superchunk data that it is XORing; otherwise multiple threads may overwrite each other’s XORed data as it is written.

Our evaluation of the superchunk recovery after a double disk failure spans several different configurations for the recovery of the 6GB superchunk. For the reconstruction to complete, 90GBs of data are processed: one 6GB Lstor parity, and 84GB of superchunk data (comprised of 14 superchunks of 6GB each).

We vary the amount of superchunk/parity data requested at a time (4MB versus 64MB), the resolution of the lock over the data being XORed (locking over the byte range versus the entire superchunk), and the network configuration (10Gbps versus 1Gbps interconnect between the nodes).

First, we refer to results in the column located second-most to the right in Table 6.1 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), because with larger chunks, a greater percentage of the recovered file is XORed while other threads wait. This intuition is confirmed in the bottom two rows of the left column of Table 6.1.

In contrast, if each thread locks only on the byte range of the recovered file that it is XORing, then requesting and XORing smaller data chunks allows a greater number of 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. Table 6.1 shows these results in the top two rows of the left column. In our cluster of 16 nodes with a 6GB superchunk, the 125 second completion time means that 90GB were transferred over the network and processed (14 superchunks and 1 Lstor parity) for a recovery throughput of about 50 MB/s. We observed that this reconstruction saturates the CPU rather than the disk or network, meaning it can be further optimized.
We next changed the network being used to a 1Gbps interface, shown in the right column of Table 6.1. The 1Gbps network configuration moves the bottleneck to the network, and changing the chunk size and lock granularity has a minimal effect on recovery performance, apparent in the narrow range of 827 seconds to 852 seconds shown in the table. The poor recovery performance of 1Gbps network (or alternatively, the dependence on the 10Gbps network) demonstrates the reliance (and potential burden) of an erasure coded recovery on the network. An erasure coded system recovers from all disk failures in this manner. RAIDP, on the other hand, must recover only one superchunk in this manner and only needs to do so after a second disk failure. (The rest of the under-replicated superchunks are merely copied as opposed to reconstructed.)

6.7 Potential Benefit of Updates-In-Place

As alluded to §4, HDFS does not support in-place updates. This limitation in HDFS prevents us from demonstrating a workload where RAIDP’s in-place updates would have shined. For example, suppose a client wants to change one 512-byte sector in every 64MB of a 5GB file. With the 64MB block size in HDFS, the 5GB file is comprised of (5GB/64MB≈) 78 block files. A distributed storage system that does not support in-place updates would need to completely read-modify-write every updated 64MB file anew, resulting in a total disk I/O of 10GB comprised of reading 5GB and then writing 5GB (despite the tiny amount of modified data). RAIDP, on the other hand, would only need to do a read-modify-write on the 78 updated sectors—one sector in each 64MB block—resulting in a total disk I/O of less than 80kB (78 × 512 × 2 bytes). While this particular workload is contrived, the in-place updates in RAIDP nonetheless demonstrate up to a 125,000-fold difference in I/O for 64MB blocks.

<table>
<thead>
<tr>
<th>lock resolution</th>
<th>chunk size</th>
<th>10Gbps NIC</th>
<th>1Gbps NIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>range lock</td>
<td>4MB</td>
<td>125 sec</td>
<td>827 sec</td>
</tr>
<tr>
<td>range lock</td>
<td>64MB</td>
<td>160 sec</td>
<td>848 sec</td>
</tr>
<tr>
<td>superchunk lock</td>
<td>64MB</td>
<td>187 sec</td>
<td>850 sec</td>
</tr>
<tr>
<td>superchunk lock</td>
<td>4MB</td>
<td>211 sec</td>
<td>852 sec</td>
</tr>
</tbody>
</table>

Table 6.1: 16 node cluster, 6GB superchunk recovery runtimes under different configurations (lower is better).
Chapter 7

Related Work

Hybrids Combining Erasure Coding & Replication  Google File System (GFS) [GGL03], Windows Azure Storage (WAS) [B. 11], Flat Datacenter Storage (FDS) [NEF+12], Haystack [BKL+10,MLR+14] and HDFS [Bor08] all triplicate warm data. We refer to Ceph [Cepa], WAS, Facebook’s storage [MLR+14,RSG+13], HP AutoRaid [WGSS96] (discussed below) and DiskReduce [FTXG09] as hybrid storage systems because data is erasure coded after first being replicated on the critical path. In WAS, data is erasure coded after a given extent reaches a certain size, whereby it is “sealed” and erasure coded in the background [B. 11,HSX+12]. Ghemawat et al. also discuss the the potential use of erasure codes for Google’s increasing read-only storage requirements [GGL03].

Facebook’s f4 introduces new terminology for data “temperature zones”, separating data into three categories: hot, warm, and cold. Facebook stores its “hot” incoming or recently written data in Haystack, which is a replicating system. Hot data is triplicated with two replicas in one datacenter, a third in another datacenter, and further protected with hardware RAID6 on each node. Facebook calls data older than one month “warm” in f4, where the data is erasure coded and distributed within a datacenter and two-way replicated across datacenters. Finally, Facebook considers its oldest data as “cold”, where only erasure codes are used and data can take days or hours to be retrieved [MLR+14,BKL+10]. In this work, we use the term “warm data” equivalently to Facebook’s “hot data”.

HP AutoRaid (which is not distributed) also provides a hybrid storage hierarchy, combining two-way mirroring for warm data and RAID5 for cold, read-only data. Data is moved between the two levels of the storage hierarchy at the granularity of 64kB units, and this migration has little affect on performance because it takes place in the background. HP AutoRaid attempts to mitigate the small write problem by directing RAID5 writes to a log structured filesystem and only writing to empty areas of the disk [WGSS96].

There have been several efforts to extend HDFS with erasure codes. HDFS-RAID [Had] and DiskReduce [FTXG09] add XOR parity and Reed-Solomon codes...
to HDFS. HDFS-RAID does not preclude a cluster from also maintaining replicas, while DiskReduce triple-replicates data blocks first and subsequently encodes them in the background. Xorbas, a work that seeks to minimize the high repair overheads with erasure codes, extends HDFS-RAID with an erasure code that uses a larger storage footprint steady state, while using half as much disk I/O and network bandwidth during a recovery [SAP+13].

The aforementioned hybrid systems exist first and foremost to minimize the storage costs of replication [RSG+13,B. 11,MLR+14]. RAIDP is unique in that it combines replication with local erasure coding in a manner that keeps it suitable for warm data, reduces storage overhead, and approaches the failure tolerance of triplicated systems. RAIDP introduces a locally stored parity and journal that together facilitate an efficient distributed erasure coding recovery on the aforementioned warm data.

**Efficient Repair** Replicated systems have the advantage of an efficient recovery procedure, because the amount of data that needs to be re-replicated after a failure is equivalent to the amount that was lost. This efficiency translates to low recovery overheads for the disks and network in the replicated system. Hence, re-replication of non-redundant data after a failure can be parallelized and even sped up when more disks are added, as replicas of individual partitions of a disk’s data chunks are typically distributed on many other machines and disks. Microsoft’s triplicated Flat Datacenter Storage exemplifies a case of a heavily parallelized recovery, replicating nearly 100 GB in about 50 seconds across a 100-disk system [NEF+12].

The same recovery using erasure codes, though parallelizable, induces significantly more network traffic, having a greater impact on the foreground jobs [RSG+13]. This traffic is a consequence of the amount of data that must be moved to recover a missing block [RL05,FLP+10,SWG+14]. Rashmi et al. report that over 98% of the stripes which are missing blocks in Facebook’s warehouse cluster are actually only missing one block. However, regardless of the number of missing blocks (even in the case of only one block) reconstruction using a Reed-Solomon $n + k$ code requires downloading $n$ blocks on a remote node [RSG+13]. Facebook further estimates that in their 3000-node cluster, erasure coding 50% of their data would result in the network being completely saturated by repair traffic [SAP+13].

Accordingly, minimizing the cost of recovery for erasure coded systems is an important area of research. Proposed solutions include delaying recovery (in case the failure is transient) [FLP+10,SWG+14], recovering blocks that contain only popular data [TFJ+07], or applying erasure codes that use fewer blocks and/or less bandwidth to recover lost data [RSG+13,KBP+12,HSX+12,WK02,WDB10,XXLC10,DGW+10,SAP+13].

One particular erasure code that seeks to minimize repair costs is the Local Reconstruction Code (LRC), used in Windows Azure Storage. LRC reduces the recovery cost of the first failure in a stripe while maintaining low storage overhead and high failure tolerance [HSX+12]. LRC’s canonical $12+2+2$ code has 12 data blocks split into two
"local stripes" of 6 blocks. LRC also includes two types of parities, local and global. In this case, the $12 + 2 + 2$ code contains 2 local parities and 2 global parities, respectively. A single missing data block can be recovered using only members of its local stripe plus a local parity block. Thus, LRC can recover from a single failure using only 6 blocks (instead of 12), e.g., with five data blocks and one parity block from the same local parity stripe. More generally, in an $n + l + g$ stripe (whereby "l" stands for local parities and "g" for global), LRC can recover from a single failure using only $n/l$ blocks.

Replicated Erasure Codes (REC) [FKK14] replicate entire erasure coded stripes in a P2P system. This combination of redundancy schemes allows any (parity or data) block within a stripe to be recovered via replication so long that a replica remains. Because of the high repair costs of erasure codes, block reconstruction is resorted to only when the available data descends below a certain threshold. RAIDP in contrast, deals with all failures immediately and replicates or reconstructs blocks as quickly as possible. RAIDP also uses local parity in proximity to allow for local read-modify-writes. P2P systems present an entirely different set of constraints than traditional datacenters or clusters which RAIDP is geared for, including dynamic and unreliable nodes and a lack of a central storage provider.

RAIDP has elements of both replication and erasure coded systems in terms of repair traffic. As described in §4 and depicted in Figure 1.1, 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.

**Updates In Place** The Ceph Storage Cluster offers both replicated and erasure coded "pools" for redundancy [Cepb] although there are restrictions on the use of erasure coded pools (e.g. no partial writes) [Cepa]. Ceph and GlusterFS both provide replicated storage with updates in place [vS14,Bel13].

ZFS is a local filesystem that contains RAID-Z, a software RAID solution with a variable-size stripe for every write. By creating a new stripe for every write with ZFS’s copy-on-write semantics, RAID-Z trades off the small write problem of updating existing stripes in favor of variable-size stripes (meaning all writes are full stripe writes). ZFS eliminates the risk of inconsistent stripes due to failures during writes by signaling their completion using a copy-on-write with atomic updates to the RAID-Z "uberblock" [BM08].

**Bad Sectors** Silent data corruptions such as corrupted disk sectors can be particularly problematic because they are only detected when the sector is accessed [EP09, BADAD+08]. By storing additional checksums for blocks and checking data integrity during reads, ZFS detects and subsequently recovers from silent data corruptions by reassembling damaged stripes [Bon05].

Dholakia, et al., propose interleaving intra-disk parity data between other data segments in the disk in order to repair bad sectors. Their goal beyond adding resilience is to use the locality of the interleaved parity to minimize performance degradation [DEH+08].
RAIDP’s Closest Ancestors  RAIDP is in fact a specific instance within design spaces described by others. Our contribution is (1) noticing that this specific instance has certain attractive properties that can make it suitable for warm data in the manner that significantly improves storage efficiency in comparison to the state of the art, and (2) leveraging these properties to devise such a system.

RAIDP could be viewed as a variant of the “two dimensional” erasure codes proposed in previous studies [Haf06, GHK+89]. The $n$ local superchunks and associated $k$ Lstor(s) comprise one $n + k$ dimension, and the distributed replication comprises an additional $1 + 1$ 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 [GMSL07], 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 in the event of a 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 detail in §2.
Chapter 8

Future Work

RAIDP has exciting 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. For disks, upgrading to SSDs will likely reduce the amount of performance impact that random I/O currently has in our workloads. For Lstors, there are new forms of storage worthy of consideration such as Seagate’s Kinetic HDDs [Seab], 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.

From the several possible future work challenges, we have selected one to discuss in more detail. The state of the art for replication is that replicated blocks are allocated randomly or pseudo randomly [FKK14, NEF+12]. 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, leaving the system vulnerable an additional failures that require a reconstruction or even cause data loss.

We next explore this problem of designing an optimal 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 a recovery because it ensures that any lost data in a double disk failure is recoverable using the 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 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 [FJF56, HK73].

Figure 8.1 depicts the failure of disk $D_1$, and the graph in Figure 8.2 depicts each sender disk (left) with an edge to each receiving disk that it can be matched with (right), for now disregarding the numerical value on each edge.

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

We could amend the formulation to the one pictured in Figure 8.2, where we assign
Figure 8.2: Senders (left) have edges to the disks which do not share a block with the sender (right). Each receiving disk on the right side may only receive one risky superchunk per recovery.

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 [Kuh06]. We could use a dynamic algorithm which 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 which contain fewer superchunks. Mills-Tettey et al. provide a dynamic version of the Hungarian Algorithm that could be used for such a formulation [MTSD07].

In §3 we noted that an $N$-disk system’s “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.” While this statement remains true, it does not take into account the aforementioned optimality when deciding where to transmit non-redundant superchunks after a failure. Importantly, there are valid superchunk layouts that serve as examples of where an optimal recovery arrangement is not possible.

For example, consider a superchunk layout in a 10 disk system, such that each disk has slots for 9 superchunks. Suppose two disks’ superchunk slots are fully occupied, meaning (1) the two disks share a superchunk with each other and (2) with all of the other 8 disks. Such a layout is exemplified in Fig. 8.3. Notice that there are 34 total superchunks (17 unique superchunks, where each is two-way replicated).

For a 10-disk system, applying the above formula to solve for $f$ shows that the system can still in theory arrange the 34 superchunks in a valid manner after 3 failures. However, suppose disk $D_2$ fails in Fig. 8.3. All of the surviving disks store a non-redundant superchunk that must be replicated, but $D_1$ already shares a superchunk with every other disk in the system, labelled superchunk 1. It cannot copy superchunk 1 to another disk without violating 1-sharing.

Hence, despite the low number of total superchunks, there is no optimal recovery given the current layout. Instead, a less efficient rearrangement of superchunks will be required. For example, $D_1$ can make two copies of superchunk 1 onto two disks $D_x$ and $D_y$ that do not currently share a superchunk with each other, and afterwards
Figure 8.3: Example for a superchunk layout that requires a superchunk rearrangement upon failure. Columns are disks. Rows are superchunks within disks. Numbers are IDs of superchunks. If D₂ fails, then D₁ cannot replicate the non-redundant superchunk 1 because doing so violates 1-sharing.

D₁ can remove its local copy. So long as the number of total superchunks is less than \((N - f) \cdot (N - f - 1)\), then two such disks \(D_x\) and \(D_y\) exist.

More generally, if disk failure leaves a non-redundant superchunk on a disk that shares superchunks with all of the remaining disks, then a rearrangement is required. (Otherwise, any straightforward copy of the non-redundant superchunk will violate 1-sharing.)
Chapter 9

Conclusions

RAIDP is a high-performance 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. RAIDP suffers from some write overhead as compared to triplication, but we believe the potential cost savings and its suitability for warm data are a worthwhile tradeoff. We are optimistic that our design can enrich the conversation regarding how failure tolerance is built into data centers.
Bibliography


O. Khan, R. Burns, J. S. Plank, W. Pierce, and C. Huang. Re-thinking 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.


[Lin] Linux Kernel Documentation. Block layer statistics in


structured storage system. SIGOPS Oper. Syst. Rev., 44(2):35–40,
April 2010.

1996.

[McC11] Colin Patrick McCabe. Make hdfs advisory caching configurable on
a per-file basis [issue: Hdfs-4817]. https://issues.apache.org/
jira/browse/HDFS-4817, May 2011. Hadoop HDFS JIRA issue

[McC14] Colin McCabe. Feeding the elephant: Optimizing the read path of
the hadoop distributed filesystem. ApacheCon North America 2014,
April 2014.

[MHL+92] C Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter
Schwarz. Aries: a transaction recovery method supporting fine-
granularity locking and partial rollbacks using write-ahead logging.
ACM Transactions on Database Systems (TODS), 17(1):94–162,

[MLR+14] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill,
Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath
Sivakumar, Linpeng Tang, et al. f4: Facebooks warm BLOB
storage system. In Proceedings of the 11th USENIX conference


הליקון, באתר ממחוז מחוזים בשכון חסניא דגלא נוצר הכנסים הם אחר מвест

ה❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉❉缨
In the first 3 blocks of the input file, we calculate the
percentage of blocks with at least 60% of their bytes
below a certain threshold. We then group these blocks
using k-means clustering with k=3. Finally, we apply
HDFS-GetMultiple for each of the 3 clusters.

In order to evaluate the performance of our
approach, we use the TeraSort benchmark.

The TeraSort benchmark is a distributed sort
algorithm that reads an input file from HDFS and
writes the sorted output file back to HDFS.

In our experiments, we compare the performance
of our approach with the default TeraSort
implementation. The default implementation
uses the map-reduce framework provided by
Hadoop.

Our approach, on the other hand, uses a
combination of distributed sorting and
parallel processing.

We evaluate our approach on a 100GB
input file consisting of 1 million
randomly generated 13-byte strings.
תקציר

מרעות אוחסן מגוריהזת מעליתית מחונשת משתרעת בשתי מונחים של יונה על יוקרה בפוגג אינדקס
אבק נחלות של יסודות ואבני פורצות יסודות הפיגורית של דרור ונסוית שיתוף עליון. מונח השכלת
שמח בקדキー מעריקת מקרה. ליאבי הפגזים של דרור והון שלו ירחיות תחתית
ופטיאס专心 ל״דה, שב שיבת פוגריה בגדת אוחסן והשוחר📒 תחתית.

委员 אוחסן שיתוף דובסי וϚוקデザイン שיתה שיתוף פוגריה של שיתוף אוחסן ישיבắcיה שיתוף
בכדי לחגון עדיפות בתפילת בשתיות מגוון. שימש במעון לקידר ה传奇游戏 גונור
וכני מנהיג עליון רוח אוחסן. בבר. אזור מתאש מנרי עליון. שיתוף אוחסן
שעתון והשוחたく רוח אוחסן. קוח שקור, מועשרת מתאש תחתית שלש圳ות משיכצל
בנבר מ lié ליחו של. מירוזרא שועג מייג של עיון רוח אוחסן.

בעבר מירוזר שיחת יסודי. (1) יש בו יסודי של יבשות פוגריה וא contaקטים בין
שלוחת ואגרה. (2) יסודי הזкон רביעי צדב מערה תוכרים (א) פוגריה. (3) אケースטר
ועוברים שיתוף יסודי תוכרון מערה פוגריה (א) פוגריה. (4) אケースטר
שלוחת אגרה בין נוגי פוגריה ו ייתוס פוגריה.

שלוחת פוגריה בין נוגי פוגריה ובית פוגריה (במקרה
生态圈 פוגריה-הՊוגריה מתאשפתiance פוגריה עדיפות יש פוגריה.

生态圈 פוגריה-הՊוגריה מתאשפתiance פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-הՊוגריה מתאשפתiance פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-הՊוגריה מתאשפתiance פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-הՊוגריה מתאשפתiance פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-הՊוגריה מתאשפתiance פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-הՊוגריה מתאשפתiance פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-הՊוגריה מתאשפתiance פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-הՊוגריה מתאשפתiance פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-הՊוגריה מתאשפתiance פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-הՊוגריה מתאשפתiance פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-הՊוגריה מתאשפתiance פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-הՊוגריה מתאשפתiance פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-הՊוגריה מתאשפתiance פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-הՊוגריה מתאשפתiance פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-הՊוגריה מתאשפתiance פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-הՊוגריה מתאשפתiance פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-הՊוגריה מתאשפתiance פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-הՊוגריה מתאשפתiance פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-הՊוגריה מתאשפתiance פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-הՊוגריה מתאשפתiance פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-הՊוגריה מתאשפתiance פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-הՊוגריה מתאשפתiance פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-הՊוגריה מתאשפתiance פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-הՊוגריה מתאשפתiance פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-הՊוגריה מתאשפתiance פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-הՊוגריה מתאשפתiance פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-הՊוגריה מתאשפתiance פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-הՊogrיה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
生态圈 פוגריה-ה珊珊 פוגריה עדיפות יש פוגריה.
אחסון מבוצר יעיל במאחורי שיכופל
נתונים רקדים מחיקת מכמי

היבר על מחקר

ל瑁 מפיל תכליך של הדרישות להבנה והאות
מומר על למענה במועד המחבר

איתן רונגלד

רשות לטכנולוגיה – מחוז טכנולוגיה ישראל
ארז הדרשמ"א
חיפה
פברואר 2015
אחת מוברר ימי במציאותشعب
נתרונים רקידת מיתיקה מקומית

איטה רודנקל