A "Hitchhiker's" Guide to Fast and Efficient Data Reconstruction in Erasure-coded Data Centers

K. V. Rashmi[1], Nihar B. Shah[1], Dikang Gu[2], Hairong Kuang[2], Dhruba Borthakur[2], and Kannan Ramchandran[1]

[1]UC Berkeley [2]Facebook

{rashmikv, nihar, kannanr}@eecs.berkeley.edu, {dikang, hairong, dhruba}@fb.com

ABSTRACT

Erasure codes such as Reed-Solomon (RS) codes are being extensively deployed in data centers since they offer significantly higher reliability than data replication methods at much lower storage overheads. These codes however mandate much higher resources with respect to network bandwidth and disk IO during reconstruction of data that is missing or otherwise unavailable. Existing solutions to this problem either demand additional storage space or severely limit the choice of the system parameters.

In this paper, we present *Hitchhiker*, a new erasure-coded storage system that reduces both network traffic and disk IO by around 25% to 45% during reconstruction of missing or otherwise unavailable data, *with no additional storage*, the same fault tolerance, and arbitrary flexibility in the choice of parameters, as compared to RS-based systems. Hitchhiker "rides" on top of RS codes, and is based on novel encoding and decoding techniques that will be presented in this paper. We have implemented Hitchhiker in the Hadoop Distributed File System (HDFS). When evaluating various metrics on the data-warehouse cluster in production at Facebook with real-time traffic and workloads, during reconstruction, we observe a 36% reduction in the computation time and a 32% reduction in the data read time, in addition to the 35% reduction in network traffic and disk IO. Hitchhiker can thus reduce the latency of degraded reads and perform faster recovery from failed or decommissioned machines.

1. INTRODUCTION

Data centers storing multiple petabytes of data have become commonplace today. These data centers are typically built out of individual components that can be unreliable, and as a result, the system has to deal with frequent failures. Various additional systems-related issues such as software glitches, machine reboots and maintenance operations also contribute to machines being rendered unavailable from time to time. In order to ensure that the data remains reliable and available despite frequent machine unavailability, data is replicated across multiple machines, typically across multiple racks as well. For instance, the Google File System [10] and the Hadoop Distributed File System (HDFS) [28] store three copies of all data by default. Although disk storage seems inexpensive for small data sizes, this assumption is no longer true when storing multiple copies at the massive scales of operation of today's data centers. As a result, several large-scale distributed storage systems [4,10] now deploy erasure codes, that provide higher reliability at significantly lower storage overheads, with the most popular choice being the family of Reed-Solomon (RS) codes [23].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGCOMM'14, August 17–22, 2014, Chicago, USA

Copyright is held by the owner/author(s). Publication rights licensed to ACM.
Copyright 2014 ACM 978-1-4503-2836-4/14/08 ...$15.00

http://dx.doi.org/10.1145/2619239.2626325

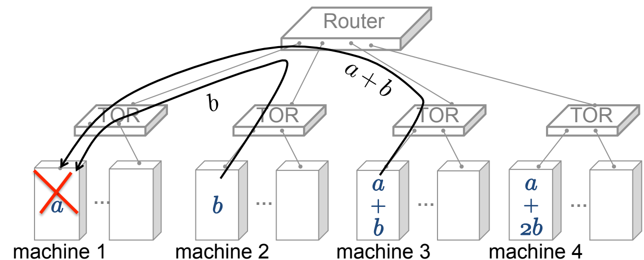


Figure 1: High network usage during reconstruction of erasure-coded data: reconstructing a single missing unit 'a' in this example requires transfer of twice the amount of data through the top-of-rack (TOR) switches.

An RS code is associated with two parameters: k and r. A $(k,\ r)$ RS code encodes k units of data into r *parity* units, in a manner that guarantees the recoverability of all the k data units from *any* k out of these $(k+r)$ units. It thus allows for tolerating unavailability of *any* r of its $(k+r)$ units. This collection of $(k+r)$ units is called a *stripe*. In a system deploying an RS code, the $(k+r)$ units belonging to a stripe are stored on distinct machines in order to maximize diversity and tolerate maximum machine unavailability. In addition, these machines are chosen from distinct racks in order to maximize tolerance to rack failures. An example of such a setting is depicted in Fig. 1, with an RS code having parameters $(k = 2, r = 2)$. Here $\{a,\ b\}$ are the two data units, which are encoded to generate two parity units, $(a+b)$ and $(a+2b)$. The figure depicts these four units stored across four machines on different racks. The data-warehouse cluster at Facebook employs an RS code with parameters $(k = 10, r = 4)$, thus resulting in a 1.4× storage overhead. In contrast, for similar levels of reliability, a replication-based system would require a 3× storage overhead.

We posit that the primary reason that makes RS codes particularly attractive for large-scale distributed storage systems is its two following properties:

P1: Storage optimality. A $(k,\ r)$ RS code entails the minimum storage overhead among all $(k,\ r)$ erasure codes that

tolerate any r failures.[1] This property is particularly relevant for large-scale storage systems since disk space usage is quickly becoming the dominant factor in these systems.
P2: Generic applicability. RS codes can be constructed for arbitrary values of the parameters $(k,\ r)$, thus allowing complete flexibility in the design of the system.

Although RS codes improve storage efficiency in data centers, they cause a *significant increase in the disk and network traffic*. This is due to the heavy download requirement during reconstruction of any missing or otherwise unavailable unit, as elaborated below. In a system that performs replication, a data unit can be reconstructed simply by copying it from another existing replica. However, in an RS-coded system, there is no such replica. To see the reconstruction operation under an RS code, let us first consider the example of a $(k = 2,\ r = 2)$ RS code as in Fig. 1. The figure depicts the reconstruction of the first data unit "a" (machine 1) from machines 2 and 3. Observe that this reconstruction operation requires transfer of two units across the network. In general, under a $(k,\ r)$ RS code, reconstruction of a single unit involves the download of k of the remaining units. An amount equal to the logical size of the data in the stripe is thus read from the disks and downloaded through the network, from which the missing unit is reconstructed. *This is k times the size of the data to be reconstructed.*

We have performed extensive measurements on Facebook's data-warehouse cluster in production, which consists of multiple thousands of nodes, and which stores multiple Petabytes of RS-encoded data. These measurements reveal that a median of more than 50 machine-unavailability events occur per day, and a median of 95,500 blocks of RS-encoded data are recovered each day (the typical size of a block is 256 Megabytes (MB)). The reconstruction operations for RS-encoded data consume a large amount of disk and cross-rack bandwidth: a median of more than 180 Terabytes (TB) of data is transferred through the top-of-rack switches every day for this purpose. For more details, the reader is referred to our earlier work [20].

In addition to the increased toll on network and disk resources, the significant increase in the amount of data to be read and downloaded during reconstruction affects RS-based storage systems in two ways. First, it *drastically hampers the read performance of the system in "degraded mode"*, i.e., when there is a read request for a data unit that is missing or unavailable. Serving such a request is called a 'degraded read'. In a replication based system, degraded reads can be performed very efficiently by serving it from one of the replicas of the requisite data. On the other hand, the high amount of data read and downloaded as well as the computational load for reconstructing any data block in an RS-based system *increases the latency of degraded reads*. Second, it *increases the recovery time of the system*: recovering a failed machine or decommissioning a machine takes significantly longer than in a replication-based system. Based on conversations with teams from multiple enterprises that deploy RS codes in their storage systems, we gathered that *this increased disk and network traffic and its impact on degraded reads and recovery is indeed a major concern*, and is one of the bottlenecks to erasure coding becoming more pervasive in large-scale distributed storage systems.

[1]In the parlance of coding theory, an RS code has the property of being 'Maximum Distance Separable (MDS)' [16].

Storage optimality and generic applicability:

Storage requirement	Same (optimal)
Supported parameters	All
Fault tolerance	Same (optimal)

Data reconstruction:

Data downloaded (i.e., network traffic)	35% less
Data read (i.e., disk traffic)	35% less
Data read time (median)	31.8% less
Data read time (95$^{\text{th}}$ percentile)	30.2% less
Computation time (median)	36.1% less

Encoding:

Encoding time (median)	72.1% more

Table 1: Performance of *Hitchhiker* as compared to Reed-Solomon-based system for default HDFS parameters.

The problem of decreasing the amount of data required to be downloaded for reconstruction in erasure-coded systems has received considerable attention in the recent past both in theory and practice [7,11–14,17,19,21,22,26,27,29,30,32]. However, all existing practical solutions either demand additional storage space [7, 12, 17, 21, 26, 27]), or are applicable in very limited settings [11, 15, 30, 32]. For example, [7,12,17] add at least 25% to 50% more parity units to the code, thereby increasing the storage overheads, [21,27] necessitate a high redundancy of $r \geq (k-1)$ in the system, while [11,15,30,32] operate in a limited setting allowing only two or three parity units.

In this paper, we present *Hitchhiker*, an erasure-coded storage system that fills this void. *Hitchhiker reduces both network and disk traffic during reconstruction by* 25% *to* 45% *without requiring any additional storage and maintaining the same level of fault-tolerance as RS-based systems.*[2] Furthermore, Hitchhiker can be used with any choice of the system parameters k and r, thus retaining both the attractive properties of RS codes described earlier. Hitchhiker accomplishes this with the aid of two novel components proposed in this paper: (i) a new encoding and decoding technique that builds on top of RS codes and reduces the amount of download required during reconstruction of missing or otherwise unavailable data, (ii) a novel disk layout technique that ensures that the savings in network traffic offered by the new code is translated to savings in disk traffic as well.

In proposing the new storage code, we make use of a recently proposed theoretical framework [22] called the 'Piggybacking framework'. In this paper, we employ this theoretical framework of piggybacking to design a novel erasure code that reduces the amount of data required during reconstruction while maintaining the storage optimality and generic applicability of RS codes. Our design offers the choice of either using it with only XOR operations resulting in significantly faster computations or with finite field arithmetic for a greater reduction in the disk and network traffic during reconstruction. Interestingly, we also show that the XOR-only design can match the savings of non-XOR design if the underlying RS code satisfies a certain simple condition.

The proposed storage codes reduce the amount of download required for data reconstruction, and this directly trans-

[2]It takes a free-ride on top of the RS-based system, retaining all its desired properties, and hence the name 'Hitchhiker'.

lates to reduction in network traffic. In the paper, we then propose a novel disk layout which ensures that the savings in network resources are also translated to savings in disk resources. In fact, this technique is applicable to a number of other recently proposed storage codes [11,15,19,21,26,27,29] as well, and hence may be of independent interest. The proposed codes also help reduce the computation time during reconstruction as compared to the existing RS codes.

Hitchhiker optimizes reconstruction of a single unit in a stripe without compromising any of the two properties of RS-based systems. Single unit reconstruction in a stripe is the most common reconstruction scenario in practice, as validated by our measurements from Facebook's data-warehouse cluster which reveal that 98.08% of all recoveries involve recovering a single unit in a stripe (see §6.5). Moreover, at any point in time, Hitchhiker can alternatively perform the (non-optimized) reconstruction of single or multiple units as in RS-based systems by connecting to *any* k of the remaining units. It follows that any optimization or solution proposed outside the erasure coding component of a storage system (e.g., [3, 5, 18]) can be used in conjunction with Hitchhiker by simply treating Hitchhiker as functionally equivalent to an RS code, thereby allowing for the benefits of both solutions.

We have implemented Hitchhiker in the Hadoop Distributed File System (HDFS). HDFS is one of the most popular open-source distributed file systems with widespread adoption in the industry. For example, multiple tens of Petabytes are being stored via RS encoding in HDFS at Facebook, a popular social-networking company.

We evaluated Hitchhiker on two clusters in Facebook's data centers, with the default HDFS parameters of ($k = 10, r = 4$). We first deployed Hitchhiker on a test cluster comprising 60 machines, and verified that the savings in the amount of download during reconstruction is as guaranteed by theory. We then evaluated various metrics of Hitchhiker on the data-warehouse cluster in production consisting of multiple thousands of machines, in the presence of ongoing real-time traffic and workloads. We observed that Hitchhiker reduces the time required for reading data during reconstruction by 32%, and reduces the computation time during reconstruction by 36%. Table 1 details the comparison between Hitchhiker and RS-based systems with respect to various metrics for ($k=10, r=4$).[3] Based on our measurements [20] of the amount of data transfer for reconstruction of RS-encoded data in the data-warehouse cluster at Facebook (discussed above), employing Hitchhiker would save close to 62TB of disk and cross-rack traffic every day while retaining the same storage overhead, reliability, and system parameters.

In summary, we make the following contributions:

- Introduce *Hitchhiker*, a new erasure-coded storage system that reduces both network and disk traffic during reconstruction of missing or otherwise unavailable data by 25% to 45% without requiring any additional storage and maintaining same level of fault-tolerance as RS-based systems. It can be used with any choice of the system parameters k and r. To the best of our knowledge, this is the first practical solution available in literature that reduces the disk and network traffic during reconstruction without increasing the storage overhead or (severely) limiting the system design.
- Propose a new storage code built on top of RS that makes use of the theoretical framework of piggybacking to reduce the amount of data required for reconstruction.
- Propose a novel disk layout technique that ensures that the savings in network traffic are translated to savings in disk traffic as well. This technique is general and can also be used to save disk resources in other storage codes.
- Implement Hitchhiker in HDFS, and test it by deploying it on a test cluster with 60 machines in a data center at Facebook and verify the theoretical guarantees of 35% savings in network and disk traffic during reconstruction.
- Evaluate the read time and compute time metrics of Hitchhiker on *the data-warehouse cluster in production at Facebook consisting of multiple thousands of machines with ongoing real-time traffic and workloads*, showing a 32% reduction in time to read data and a 36% reduction in computation time during reconstruction. This establishes that Hitchhiker can reduce the latency of degraded reads, and also perform faster recovery from machine failures and decommissioning.

[3]More specifically, these numbers are for the 'Hitchhiker-XOR+' version of Hitchhiker.

2. THEORETICAL BACKGROUND

2.1 Reed-Solomon (RS) codes

As discussed briefly in §1, a (k, r) RS code [23] encodes k data bytes into r parity bytes. The code operates on each set of k bytes independently and in an identical fashion. Each such set on which independent and identical operations are performed by the code is called a *stripe*. Fig. 2 depicts ten units of data encoded using a ($k = 10,\ r = 4$) RS code that generates four parity units. Here, $a_1, \ldots, a_{10}$ are one byte each, and so are $b_1, \ldots, b_{10}$. Note that the code is operating independently and identically on the two columns, and hence each of the two columns constitute a stripe of this code.

We use the notation $\boldsymbol{a} = [a_1\ \cdots\ a_{10}]$ and $\boldsymbol{b} = [b_1\ \cdots\ b_{10}]$. Each of the functions f_1, f_2, f_3 and f_4, called *parity functions*, operate on $k = 10$ data bytes to generate the $r = 4$ parities. The output of each of these functions comprises one byte. These functions are such that one can reconstruct $\boldsymbol{a}$ from any 10 of the 14 bytes $\{a_1, \ldots, a_{10}, f_1(\boldsymbol{a}), \ldots, f_4(\boldsymbol{a})\}$. In general, a ($k$, r) RS code has r parity functions generating the r parity bytes such that all the k data bytes are recoverable from any k of the $(k + r)$ bytes in a stripe.

In the above discussion, each unit is considered indivisible: all the bytes in a unit share the same fate, i.e., all the bytes are either available or unavailable. Hence, a reconstruction operation is always performed on one or more entire units. A unit may be the smallest granularity of the data handled by a storage system, or it may be a data chunk that is always written onto the same disk block.

Reconstruction of any unit is performed in the following manner. Both the stripes of any 10 of the remaining 13 units are accessed. The RS code guarantees that any desired data can be obtained from any 10 of the units, allowing for reconstruction of the requisite unit from this accessed data. This reconstruction operation in the RS code requires accessing a total of 20 bytes from the other units.

2.2 Theoretical Framework of Piggybacking

In this section, we review a theoretical framework proposed in [22] for the construction of erasure codes, called the

		1 byte	1 byte
Data	unit 1	a_1	b_1
	⋮	⋮	⋮
	unit 10	a_{10}	b_{10}
Parity	unit 11	$f_1(\boldsymbol{a})$	$f_1(\boldsymbol{b})$
	unit 12	$f_2(\boldsymbol{a})$	$f_2(\boldsymbol{b})$
	unit 13	$f_3(\boldsymbol{a})$	$f_3(\boldsymbol{b})$
	unit 14	$f_4(\boldsymbol{a})$	$f_4(\boldsymbol{b})$
		stripe	stripe

Figure 2: Two stripes of a (k=10, r=4) **Reed-Solomon (RS) code. Ten units of data (first ten rows) are encoded using the RS code to generate four parity units (last four rows).**

1 byte	1 byte
a_1	$b_1+g_1(\boldsymbol{a})$
⋮	⋮
a_{10}	$b_{10}+g_{10}(\boldsymbol{a})$
$f_1(\boldsymbol{a})$	$f_1(\boldsymbol{b})+g_{11}(\boldsymbol{a})$
$f_2(\boldsymbol{a})$	$f_2(\boldsymbol{b})+g_{12}(\boldsymbol{a})$
$f_3(\boldsymbol{a})$	$f_3(\boldsymbol{b})+g_{13}(\boldsymbol{a})$
$f_4(\boldsymbol{a})$	$f_4(\boldsymbol{b})+g_{14}(\boldsymbol{a})$
1st substripe	2nd substripe
stripe	

Figure 3: The theoretical framework of Piggybacking [22] for parameters (k=10, r=4)**. Each row represents one unit of data.**

1 byte	1 byte
a_1	b_1
⋮	⋮
a_{10}	b_{10}
$f_1(\boldsymbol{a})$	$f_1(\boldsymbol{b})$
$f_2(\boldsymbol{a})$	$f_2(\boldsymbol{b})\oplus a_1\oplus a_2\oplus a_3$
$f_3(\boldsymbol{a})$	$f_3(\boldsymbol{b})\oplus a_4\oplus a_5\oplus a_6$
$f_4(\boldsymbol{a})$	$f_4(\boldsymbol{b})\oplus a_7\oplus a_8\oplus a_9\oplus a_{10}$
1st substripe	2nd substripe
stripe	

Figure 4: Hitchhiker-XOR code for (k=10, r=4)**. Each row represents one unit of data.**

Piggybacking framework. The framework operates on pairs of stripes of an RS code (e.g., the pair of columns depicted in Fig. 2). The framework allows for arbitrary functions of the data pertaining to one stripe to be added to the second stripe. This is depicted in Fig. 3 where arbitrary functions $g_1, \ldots, g_{14}$ of the data of the first stripe of the RS code $\boldsymbol{a}$ are added to the second stripe of the RS code. Each of these functions outputs values of size one byte.

The Piggybacking framework performs independent and identical operations on pairs of columns, and hence a stripe consists of two columns. The constituent columns of a stripe will be referred to as *substripes* (see Fig. 3).

Irrespective of the choice of the Piggybacking functions $g_1, \ldots, g_{14}$, the code retains the fault tolerance and the storage efficiency of the underlying RS code. To see the fault tolerance, recall that RS codes allow for tolerating failure of any r units. In our setting of ($k=10$, $r=4$), this amounts to being able to reconstruct the entire data from any 10 of the 14 units in that stripe. Now consider the code of Fig. 3, and consider any 10 units (rows). The first column of Fig. 3 is identical to the first stripe (column) of the RS code of Fig. 2, which allows for reconstruction of $\boldsymbol{a}$ from these 10 units. Access to $\boldsymbol{a}$ now allows us to compute the values of the functions $g_1(\boldsymbol{a}), \ldots, g_{14}(\boldsymbol{a})$, and subtract the respective functions out from the second columns of the 10 units under consideration. What remains are some 10 bytes out of $\{b_1, \ldots, b_{10}, f_1(\boldsymbol{b}), \ldots, f_4(\boldsymbol{b})\}$. This is identical to the second stripe (column) of the RS code in Fig. 2, which allows for the reconstruction of $\boldsymbol{b}$. It follows that the code of Fig. 3 can also tolerate the failure of any $r = 4$ units. We will now argue storage efficiency. Each function g_i outputs one byte of data. Moreover, the operation of adding this function to the RS code is performed via "finite field arithmetic", and hence the result also comprises precisely one byte.[4] Thus the amount of storage is not increased upon performing these operations. It is easy to see that each of these arguments extend to any generic values of the parameters k and r.

The theoretical framework of Piggybacking thus provides a high degree of flexibility in the design of the erasure code by allowing for an arbitrary choice of the functions g_i's. When using this framework for constructing codes, this choice must be made in a manner that imparts desired features to the erasure code. In this paper we design these functions to increase the efficiency of reconstruction.

[4]We do not employ any special properties of finite field arithmetic in the paper, and do not assume the reader to be conversant of the same.

3. HITCHHIKER'S ERASURE CODE

One of the main components of Hitchhiker is the new erasure code proposed in this paper. The proposed code reduces the amount of data required during reconstruction, without adding any additional storage overhead. Furthermore, the code can be used for any values of the system parameters k and r, thus maintaining both (P1) storage optimality and (P2) generic applicability properties of RS codes. This code is based on the recently proposed theoretical framework of piggybacking ([22];§2).

The proposed code has three versions, two of which require only XOR operations in addition to encoding of the underlying RS code. The XOR-only feature of these erasure codes significantly reduces the computational complexity of decoding, making degraded reads and failure recovery faster (§6.3). Hitchhiker's erasure code optimizes only the reconstruction of data units; reconstruction of parity units is performed as in RS codes.

The three versions of Hitchhiker's erasure code are described below. Each version is first illustrated with an example for the parameters ($k = 10$, $r = 4$), followed by the generalization to arbitrary values of the parameters. Without loss of generality, the description of the codes' operations considers only a single stripe (comprising two substripes). Identical operations are performed on each stripe of the data.

3.1 Hitchhiker-XOR

As compared to a ($k = 10$, $r = 4$) RS code, Hitchhiker-XOR saves 35% in the amount of data required during the reconstruction of the first six data units and 30% during the reconstruction of the remaining four data units.

3.1.1 Encoding

The code for ($k = 10$, $r = 4$) is shown in Fig. 4. The figure depicts a single stripe of this code, comprising two sub-

stripes. The encoding operation in this code requires only XOR operations in addition to the underlying RS encoding.

3.1.2 Reconstruction

First consider reconstructing the first unit. This requires reconstruction of $\{a_1, b_1\}$ from the remaining units. Hitchhiker-XOR accomplishes this using only 13 bytes from the other units: the bytes belonging to both the substripes of units $\{2, 3\}$ and the bytes belonging to only the second substripe of units $\{4, \ldots, 12\}$. These 13 bytes are $\{a_2, a_3, b_2, b_3 \ldots, b_{10}, f_1(\boldsymbol{b}), f_2(\boldsymbol{b}) \oplus a_1 \oplus a_2 \oplus a_3\}$. The decoding procedure comprises three steps. Step 1: observe that the 10 bytes $\{b_2, \ldots, b_{10}, f_1(\boldsymbol{b})\}$ are identical to the corresponding 10 bytes in the RS encoding of $\boldsymbol{b}$ (Fig. 2). RS decoding of these 10 bytes gives $\boldsymbol{b}$ (and this includes one of the desired bytes b_1). Step 2: XOR $f_2(\boldsymbol{b})$ with the second byte $(f_2(\boldsymbol{b}) \oplus a_1 \oplus a_2 \oplus a_3)$ of the 12th unit. This gives $(a_1 \oplus a_2 \oplus a_3)$. Step 3: XORing this with a_2 and a_3 gives a_1. Thus both a_1 and b_1 are reconstructed by using only 13 bytes, as opposed to 20 bytes in RS codes, resulting in a saving of 35%.

Let us now consider the reconstruction of any unit $i \in \{1, \ldots, 10\}$, which requires reconstruction of $\{a_i, b_i\}$. We shall first describe what data (from the other units) is required for the reconstruction, following which we describe the decoding operation. Any data unit $i \in \{1, 2, 3\}$ is reconstructed using the following 13 bytes: the bytes of both the substripes of units $\{1, 2, 3\} \backslash \{i\}$, and the bytes belonging to only the second substripe from units $\{4, \ldots, 12\}$.[5] Any data unit $i \in \{4, 5, 6\}$ is also reconstructed using only 13 bytes: the bytes belonging to both the substripes of units $\{4, 5, 6\} \backslash \{i\}$, and the bytes belonging to only the second substripe of units $\{1, 2, 3, 7, \ldots, 11, 13\}$. Any data unit $i \in \{7, 8, 9, 10\}$ is reconstructed using 14 bytes: both substripes of units $\{7, 8, 9, 10\} \backslash \{i\}$, and only the second substripe of units $\{1, \ldots, 6, 11, 14\}$.

Three-step decoding procedure:

Step 1: The set of 10 bytes $\{b_1, \ldots, b_{10}, f_1(\boldsymbol{b})\} \backslash \{b_i\}$ belonging to the second substripe of the units $\{1, \ldots, 11\} \backslash \{i\}$ is identical to the 10 corresponding encoded bytes in the RS code. Perform RS decoding of these 10 bytes to get $\boldsymbol{b}$ (which includes one of the desired bytes b_i).

Step 2: In the other bytes accessed, subtract out all components that involve $\boldsymbol{b}$.

Step 3: XOR the resulting bytes to get a_i.

Observe that during the reconstruction of any data unit, the remaining data units *do not* perform any computation. In the parlance of coding theory, this property is called 'repair-by-transfer' [26]. This property carries over to all three versions of Hitchhiker's erasure code.

3.2 Hitchhiker-XOR+

Hitchhiker-XOR+ further reduces the amount of data required for reconstruction as compared to Hitchhiker-XOR, and employs only additional XOR operations. It however requires the underlying RS code to possess a certain property. This property, which we term the *all-XOR-parity* property, requires at least one parity function of the RS code to be an XOR of all the data units. That is, a $(k,\ r)$ RS code satisfying all-XOR-parity will have one of the r parity bytes as an XOR of all the k data bytes. For $(k = 10,\ r = 4)$, Hitchhiker-XOR+ requires 35% lesser data for reconstruction of any of the data units as compared to RS codes.

[5] For any set $\mathcal{A}$ and any element $i \in \mathcal{A}$, the notation $\mathcal{A} \backslash \{i\}$ denotes all elements of $\mathcal{A}$ except i.

3.2.1 Encoding

The $(k = 10,\ r = 4)$ Hitchhiker-XOR+ code is shown in Fig. 5. The Hitchhiker-XOR+ code is obtained by performing one additional XOR operation on top of Hitchhiker-XOR: in the second parity of Hitchhiker-XOR, the byte of the second substripe is XORed onto the byte of the first substripe to give Hitchhiker-XOR+. The underlying RS code in this example satisfies the all-XOR-parity property with its second parity function f_2 being an XOR of all the inputs.

We now argue that this additional XOR operation does not violate the fault tolerance level and storage efficiency. To see fault tolerance, observe that the data of the second parity unit of Hitchhiker-XOR+ can always be converted back to that under Hitchhiker-XOR by XORing its second substripe with its first substripe. It follows that the data in any unit under Hitchhiker-XOR+ is equivalent to the data in the corresponding unit in Hitchhiker-XOR. The fault tolerance properties of Hitchhiker-XOR thus carry over to Hitchhiker-XOR+. Storage efficiency is retained because the additional XOR operation does not increase the space requirement.

3.2.2 Decoding

The recovery of any unit i requires 13 bytes from the other units. The choice of the bytes to be accessed depends on the value of i, and is described below. The bytes required for the reconstruction of any data unit $i \in \{1, \ldots, 6\}$ are identical to that in Hitchhiker-XOR. Any data unit $i \in \{7, 8, 9\}$ is reconstructed using the following 13 bytes: the bytes of both substripes of units $\{7, 8, 9\} \backslash \{i\}$, and the bytes of only the second substripes of units $\{1, \ldots, 6, 10, 11, 14\}$. The tenth unit is also reconstructed using only 13 bytes: the bytes of only the second substripes of units $\{1, \ldots, 9, 11, 13, 14\}$, and the byte of only the first substripe of unit 12. The decoding procedure that operates on these 13 bytes is identical to the three-step decoding procedure described in §3.1.2.

3.3 Hitchhiker-nonXOR

We saw that Hitchhiker-XOR+ results in more savings as compared to Hitchhiker-XOR, but requires the underlying RS code to have the all-XOR-parity property. Hitchhiker-nonXOR presented here guarantees the same savings as Hitchhiker-XOR+ even when the underlying RS code does not possess the all-XOR-parity property, but at the cost of additional finite-field arithmetic. Hitchhiker-nonXOR can thus be built on top of any RS code. It offers a saving of 35% during the reconstruction of any data unit.

3.3.1 Encoding

The code for $(k = 10,\ r = 4)$ is shown in Fig. 6. As in Hitchhiker-XOR, in the second parity, the first byte is XORed with the second byte. The final value of the second parity as shown in Fig. 6 is a consequence of the fact that $f_2(\boldsymbol{a}) \oplus f_2(a_1, a_2, a_3, 0, \ldots, 0) = f_2(0, 0, 0, a_4, \ldots, a_{10})$ due to the linearity of RS encoding (this is discussed in greater detail in §5.2.2).

3.3.2 Decoding

Recovery of any unit requires only 13 bytes from other units. This set of 13 bytes is the same as in Hitchhiker-

unit 1	a_1	b_1
$\vdots$	$\vdots$	$\vdots$
unit 10	a_{10}	b_{10}
unit 11	$f_1(\boldsymbol{a})$	$f_1(\boldsymbol{b})$
unit 12	$\bigoplus_{i=4}^{10} a_i \oplus \bigoplus_{i=1}^{10} b_i$	$\bigoplus_{i=1}^{10} b_i \oplus \bigoplus_{i=1}^{3} a_i$
unit 13	$f_3(\boldsymbol{a})$	$f_3(\boldsymbol{b}) \oplus \bigoplus_{i=4}^{6} a_i$
unit 14	$f_4(\boldsymbol{a})$	$f_4(\boldsymbol{b}) \oplus \bigoplus_{i=7}^{9} a_i$

Figure 5: Hitchhiker-XOR+ for $(k{=}10, r{=}4)$. Parity 2 of the underlying RS code is all-XOR.

a_1	b_1
$\vdots$	$\vdots$
a_{10}	b_{10}
$f_1(\boldsymbol{a})$	$f_1(\boldsymbol{b})$
$f_2(0,0,0,a_4,\ldots,a_{10}) \oplus f_2(\boldsymbol{b})$	$f_2(\boldsymbol{b}) \oplus f_2(a_1,a_2,a_3,0,\ldots,0)$
$f_3(\boldsymbol{a})$	$f_3(\boldsymbol{b}) \oplus f_2(0,0,0,a_4,a_5,a_6,0,\ldots,0)$
$f_4(\boldsymbol{a})$	$f_4(\boldsymbol{b}) \oplus f_2(0,\ldots,0,a_7,a_8,a_9,0)$

Figure 6: Hitchhiker-nonXOR code for $(k{=}10, r{=}4)$. This can be built on any RS code. Each row is one unit of data.

XOR+. The decoding operation is a three-step procedure. The first two steps are identical to the first two steps of the decoding procedure of Hitchhiker-XOR described at the end of §3.1.2. The third step is slightly different, and requires an RS decoding operation (for units 1 to 9), as described below.

During reconstruction of any unit $i \in \{1, 2, 3\}$, the output of the second step is the set of three bytes $\{a_1, a_2, a_3, f_1(a_1, a_2, a_3, 0, \ldots, 0)\} \backslash \{a_i\}$. This is equivalent to having some 10 of the 11 bytes of the set $\{a_1, a_2, a_3, 0, \ldots, 0, f_1(a_1, a_2, a_3, 0, \ldots, 0)\}$. Now, this set of 11 bytes is equal to the set of first 11 bytes of the RS encoding of $\{a_1, a_2, a_3, 0, \ldots, 0\}$. An RS decoding operation thus gives $\{a_1, a_2, a_3\}$ which contains the desired byte a_i. Recovery of any other unit $i \in \{4, \ldots, 9\}$ follows along similar lines.

During the reconstruction of unit 10, the output of the second step is $f_1(0, \ldots, 0, a_{10})$. Hence the third step involves only a single (finite-field) multiplication operation.

3.4 Generalization to any $(k,\ r)$

The encoding and decoding procedures for the general case follow along similar lines as the examples discussed above, and are formally described in the Appendix. In each of the three versions, the amount of data required for reconstruction is reduced by 25% to 45% as compared to RS codes, depending on the values of the parameters k and r. For instance, $(k = 6, r = 3)$ provides a saving of 25% with Hitchhiker-XOR and 34% with Hitchhiker-XOR+ and Hitchhiker-nonXOR; $(k = 20, r = 5)$ provides a savings of 37.5% with Hitchhiker-XOR and 40% with Hitchhiker-XOR+ and Hitchhiker-nonXOR.

4. "HOP-AND-COUPLE" FOR DISK EFFICIENCY

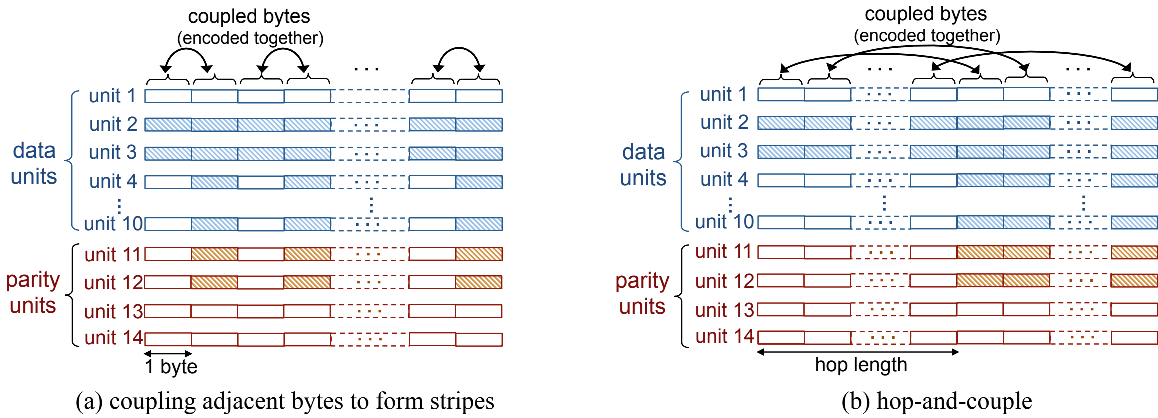
The description of the codes in §2 and §3 considers only two bytes per data unit. We now move on to consider the more realistic scenario where each of the k data units to be encoded is larger (than two bytes). In the encoding process, these k data units are first partitioned into stripes, and identical encoding operations are performed on each of the stripes. The RS code considers one byte each from the k data units as a stripe. On the other hand, Hitchhiker's erasure code has two substripes within a stripe (§3) and hence *couples* pairs of bytes within each of the k data units to form the substripes of a stripe. We will shortly see that the choice of the bytes to be coupled plays a crucial role in determining the efficiency of disk reads during reconstruction.

A natural strategy for forming the stripes for Hitchhiker's erasure code is to couple adjacent bytes within each unit, with the first stripe comprising the first two bytes of each of the units, the second stripe comprising the next two bytes, and so on. Fig. 7a depicts such a method of coupling for $(k = 10, r = 4)$. In the figure, the bytes accessed during the reconstruction of the first data unit are shaded. This method of coupling, however, results in highly discontiguous reads during reconstruction: alternate bytes are read from units 4 to 12 as shown in the figure. This high degree of discontinuity is detrimental to disk read performance, and forfeits the potential savings in disk IO during data reconstruction. The issue of discontiguous reads due to coupling of adjacent bytes is not limited to the reconstruction of the first data unit - it arises during reconstruction of any of the data units.

In order to ensure that the savings offered by Hitchhiker's erasure codes in the amount of data read during reconstruction are effectively translated to gains in disk read efficiency, we propose a coupling technique for forming stripes that we call *hop-and-couple*. This technique aims to minimize the degree of discontinuity in disk reads during the reconstruction of data units. The hop-and-couple technique couples a byte with another byte within the same unit that is a certain distance ahead (with respect to the natural ordering of bytes within a unit), i.e., it couples bytes after "hopping" a certain distance. We term this distance as the *hop-length*. This technique is illustrated in Fig. 7b, where the hop-length is chosen to be half the size of a unit.

The hop-length may be chosen to be any number that divides $\frac{B}{2}$, where B denotes the size of each unit. This condition ensures that all the bytes in the unit are indeed coupled. Coupling adjacent bytes (e.g., Fig. 7a) is a degenerate case where the hop-length equals 1. The hop-length significantly affects the contiguity of the data read during reconstruction of the data units, in that the data is read as contiguous chunks of size equal to the hop-length. For Hitchhiker's erasure codes, a hop-length of $\frac{B}{2}$ minimizes the total number of discontiguous reads required during the reconstruction of data units. While higher values of hop-length reduces the number of discontiguous reads, it results in bytes further apart being coupled to form stripes. This is a trade-off to be considered when choosing the value of the hop-length, and is discussed further in §7.

We note that the reconstruction operation under RS codes reads the entire data from k of the units, and hence trivially, the reads are contiguous. On the other hand, any erasure code that attempts to make reconstruction more efficient by downloading *partial* data from the units (e.g., [11, 15, 19, 21, 26, 27, 29]) will encounter the issue of discontiguous



(a) coupling adjacent bytes to form stripes

(b) hop-and-couple

Figure 7: Two ways of coupling bytes to form stripes for Hitchhiker's erasure code. The shaded bytes are read and downloaded for the reconstruction of the first unit. While both methods require the same amount of data to be read, the reading is discontiguous in (a), while (b) ensures that the data to be read is contiguous.

reads, as in Hitchhiker's erasure code. Any such erasure code would have multiple (say, α) substripes in every stripe and would read a subset of these substripes from each of the units during reconstruction. The hop-and-couple technique can be applied to any such erasure code to translate the network savings to disk savings as well. The hop-length can be chosen to be any number that divides $\frac{B}{\alpha}$. As in the case of Hitchhiker's erasure codes (where $\alpha = 2$), this condition ensures that all the bytes are indeed coupled. If the bytes to be coupled are chosen with hop-length equal to B/α, then the hop-and-couple technique would ensure that all the bytes of a substripe are contiguous within any unit. Reading a substripe from a unit would then result in a contiguous disk read, thereby minimizing the total degree of discontiguity in disk reads during reconstruction.

5. IMPLEMENTATION

We have implemented Hitchhiker in the Hadoop Distributed File System (HDFS). HDFS-RAID [1] is a module in HDFS that deals with erasure codes and is based on [8]. This module forms the basis for the erasure-coded storage system employed in the data-warehouse cluster at Facebook, and is open sourced under Apache. HDFS-RAID deployed at Facebook is based on RS codes, and we will refer to this system as RS-based HDFS-RAID. Hitchhiker builds on top of RS codes, and the present implementation uses the RS encoder and decoder modules of RS-based HDFS-RAID as its building blocks.

5.1 Brief description of HDFS-RAID

HDFS stores each file by dividing it into *blocks* of a certain size. By default, the size of each block is 256MB, and this is also the value that is typically used in practice. In HDFS, three replicas of each block are stored in the system by default. HDFS-RAID offers RS codes as an alternative to replication for maintaining redundancy.

The relevant modules of HDFS-RAID and the execution flows for relevant operations are depicted in Fig. 8. The *RAID-Node* manages all operations related to the use of erasure codes in HDFS. It has a list of files that are to be converted from the replicated state to the erasure-coded state, and periodically performs encoding of these files via MapReduce jobs. Sets of k blocks from these files are encoded to generate r parity blocks each.[6] Once the r parity blocks of a set are successfully written into the file system, the replicas of the k data blocks of that set are deleted. The MapReduce job calls the *Encoder* of the erasure code to perform encoding. The Encoder uses the *Parallel-Reader* to read the data from the k blocks that are to be encoded. The Parallel-Reader opens k parallel streams, issues HDFS read requests for these blocks, and puts the data read from each stream into different buffers. Typically, the buffers are 1MB each. When one buffer-sized amount of data is read from each of the k blocks, the Encoder performs the computations pertaining to the encoding operation. This process is repeated until the entire data from the k blocks are encoded.

The RAID-Node also handles recovery operations, i.e., reconstructing missing blocks in order to maintain the reliability of the system. The RAID-Node has a list of blocks that are missing and need to be *recovered*. It periodically goes through this list and reconstructs the blocks by executing a MapReduce job. The MapReduce job calls the *Decoder* of the erasure code to perform the reconstruction operation. The Decoder uses the Parallel-Reader to read the data required for reconstruction. For an RS code, data from k blocks belonging to the same set as that of the block under reconstruction is read in parallel. As in the encoding process, data from the k blocks are read into buffers, and the computations are performed once one buffer size amount of data is read from each of the k blocks. This is repeated until the entire block is reconstructed.

HDFS directs any request for a *degraded read* (i.e., a read request for a block that is unavailable) to the *RAID File System*. The requested block is reconstructed on the fly by the RAID-Node via a MapReduce job. The execution of this reconstruction operation is identical to that in the reconstruction process discussed above.

5.2 Hitchhiker in HDFS

We implemented Hitchhiker in HDFS making use of the new erasure code (§3) and the hop-and-couple technique (§4) proposed in this paper. We implemented all three versions of the proposed storage code: Hitchhiker-XOR, Hitchhiker-XOR+, and Hitchhiker-nonXOR. This required implementing new Encoder, Decoder and Parallel-Reader

[6] In the context of HDFS, the term block corresponds to a unit and we will use these terms interchangeably.

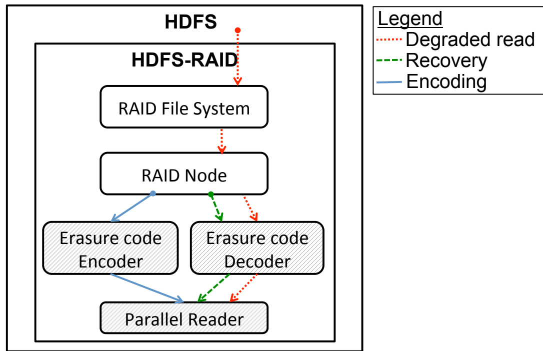


Figure 8: Relevant modules in HDFS-RAID. The execution flow for encoding, degraded reads and reconstruction operations are shown. Hitchhiker is implemented in the shaded modules.

modules (shaded in Fig. 8), entailing $7k$ lines of code. The implementation details described below pertain to parameters ($k = 10$, $r = 4$) which are the default parameters in HDFS. We emphasize that, however, Hitchhiker is generic and supports all values of the parameters k and r.

5.2.1 Hitchhiker-XOR and Hitchhiker-XOR+

The implementation of these two versions of Hitchhiker is exactly as described in §3.1 and §3.2 respectively.

5.2.2 Hitchhiker-nonXOR

Hitchhiker-nonXOR requires finite field arithmetic operations to be performed in addition to the operations performed for the underlying RS code. We now describe how our implementation executes these operations.

Encoder: As seen in Fig. 6, in addition to the underlying RS code, the Encoder needs to compute the functions $f_2(a_1,a_2,a_3,0...,0)$, $f_2(0,...,0,a_4,a_5,a_6,0...,0)$, and $f_2(0,...,0,a_7,a_8,a_9,0)$, where f_2 is the second parity function of the RS code. One way to perform these computations is to simply employ the existing RS encoder. This approach, however, involves computations that are superfluous to our purpose: The RS encoder uses an algorithm based on polynomial computations [16] that inherently computes *all* the four parities $f_1(\boldsymbol{x})$, $f_2(\boldsymbol{x})$, $f_3(\boldsymbol{x})$, and $f_4(\boldsymbol{x})$ for any given input $\boldsymbol{x}$. On the other hand, we require only one of the four parity functions for each of the three distinct inputs $(a_1,a_2,a_3,0...,0)$, $(0,...,0,a_4,a_5,a_6,0...,0)$ and $(0,...,0,a_7,a_8,a_9,0)$. Furthermore, these inputs are sparse, i.e., most of the input bytes are zeros.

Our Encoder implementation takes a different approach, exploiting the fact that RS codes have the property of *linearity*, i.e., each of the parity bytes of an RS code can be written as a linear combination of the data bytes. Any parity function f_ℓ can thus be specified as $f_\ell(\boldsymbol{x}) = \bigoplus_{j=1}^{10} c_{\ell,j} x_j$ for some constants $c_{\ell,1}, \ldots, c_{\ell,10}$. We first inferred the values of these constants for each of the parity functions (from the existing "black-box" RS encoder) in the following manner. We first fed the input $[1\ 0\ \cdots\ 0]$ to the encoder. This gives the constants $c_{1,1}, \ldots, c_{4,1}$ as the values of the four parities respectively in the output from the encoder. The values of the other constants were obtained in a similar manner by feeding different unit vectors as inputs to the encoder. Note that obtaining the values of these constants from the "black-box" RS encoder is a one-time task. With these constants, the encoder computes the desired functions simply as linear combinations of the given data units (for example, we compute $f_2(a_1, a_2, a_3, 0 \ldots, 0)$ simply as $c_{2,1}a_1 \oplus c_{2,2}a_2 \oplus c_{2,3}a_3$).

Decoder: As seen in §3.3, during reconstruction of any of the first nine units, the first byte is reconstructed by performing an RS decoding of the data obtained in the intermediate step. One straightforward way to implement this is to use the existing RS decoder for this operation. However, we take a different route towards a computationally cheaper option. We make use of two facts: (a) Observe from the description of the reconstruction process (§3.3) that for this RS decoding operation the data is known to be 'sparse', i.e., contains many zeros, and (b) the RS code has the linearity property, and furthermore, any linear combination of zero-valued data is zero. Motivated by this observation, our Decoder simply inverts this sparse linear combination to reconstruct the first substripe in a more efficient manner.

5.2.3 Hop-and-couple

Hitchhiker uses the proposed hop-and-couple technique to couple bytes during encoding. We use a hop-length of half a block since the granularity of data read requests and recovery in HDFS is typically an entire block. As discussed in §4, this choice of the hop-length minimizes the number of discontiguous reads. However, the implementation can be easily extended to other hop-lengths as well.

When the block size is larger than the buffer size, coupling bytes that are half a block apart requires reading data from two different locations within a block. In Hitchhiker, this is handled by the Parallel-Reader.

5.2.4 Data read patterns during reconstruction

During reconstruction in Hitchhiker, the choice of the blocks from which data is read, the seek locations, and the amount of data read are determined by the identity of the block being reconstructed. Since Hitchhiker uses the hop-and-couple technique for coupling bytes to form stripes during encoding, the reads to be performed during reconstruction are always contiguous within any block (§4).

For reconstruction of any of the first nine data blocks in Hitchhiker-XOR+ or Hitchhiker-nonXOR or for reconstruction of any of the first six data blocks in Hitchhiker-XOR, Hitchhiker reads and downloads two full blocks (i.e., both substripes) and nine half blocks (only the second substripes). For example, the read patterns for decoding blocks 1 and 4 are as shown in Fig. 9a and 9b respectively. For reconstruction of any of the last four data blocks in Hitchhiker-XOR, Hitchhiker reads and downloads three full blocks (both substripes) and nine half blocks (only the second substripes). For recovery of the tenth data block in Hitchhiker-XOR+ or Hitchhiker-nonXOR, Hitchhiker reads half a block each from the remaining thirteen blocks. This read pattern is shown in Fig. 9c.

6. EVALUATION

6.1 Evaluation Setup and Metrics

We evaluate Hitchhiker using two HDFS clusters at Facebook: (i) the data-warehouse cluster in production comprising multiple thousands of machines, with ongoing real-time traffic and workloads, and (ii) a smaller test cluster comprising around 60 machines. In both the clusters, machines are connected to a rack switch through 1Gb/s Ether-

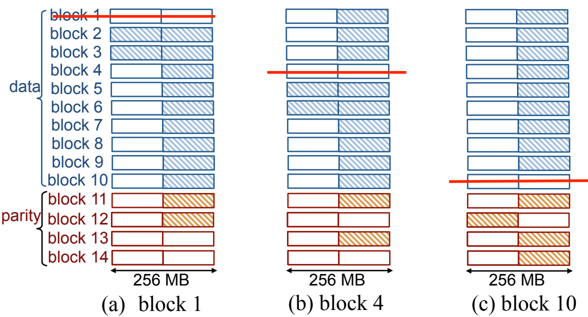


Figure 9: Data read patterns during reconstruction of blocks 1, 4 and 10 in Hitchhiker-XOR+: the shaded bytes are read and downloaded.

net links. The higher levels of the network tree architecture have 8Gb/s Ethernet connections.

We compare Hitchhiker and RS-based HDFS-RAID in terms of the time taken for computations during encoding and reconstruction, the time taken to read the requisite data during reconstruction, and the amount of data that is read and transferred during reconstruction.[7] Note that in particular, the computation and the data read times during reconstruction are vital since they determine the performance of the system during degraded reads and recovery.

6.2 Evaluation Methodology

The encoding and reconstruction operations in HDFS-RAID, including those for degraded reads and recovery, are executed as MapReduce jobs. The data-warehouse cluster does not make any distinction between the MapReduce jobs fired by the RAID-Node and those fired by a user. This allows us to perform evaluations of the timing metrics for encoding, recovery, and degraded read operations by running them as MapReduce jobs on the production cluster. Thus evaluation of all the timing metrics is performed in the presence of real-time production traffic and workloads.

We deployed Hitchhiker on a 60-machine test cluster in one of the data centers at Facebook, and evaluated the end-to-end functionality of the system. Tests on this cluster verified that savings in network and disk traffic during reconstruction are as guaranteed in theory by Hitchhiker's erasure code.

For all the evaluations, we consider the encoding parameters ($k = 10$, $r = 4$), a block size of 256MB (unless mentioned otherwise), and a buffer size of 1MB. These are the default parameters in HDFS-RAID. Moreover, these are the parameters employed in the data-warehouse cluster in production at Facebook to store multiple tens of Petabytes.

In the evaluations, we show results of the timing metrics for *one buffer size*. This is sufficient since the same operations are performed repeatedly on one buffer size amount of data until an entire block is processed.

6.3 Computation time for degraded reads & recovery

Fig. 10 shows the comparison of computation time during data reconstruction (from 200 runs each). Note that (i) in both Hitchhiker-XOR+ and Hitchhiker-nonXOR, reconstruction of any of the first nine data blocks entails same amount of computation, (ii) reconstruction of any data block in Hitchhiker-XOR is almost identical to reconstruction of block 1 in Hitchhiker-XOR+. Hence, for brevity, no separate measurements are shown.

[7]Both systems read the same data during encoding, and are hence trivially identical on this metric.

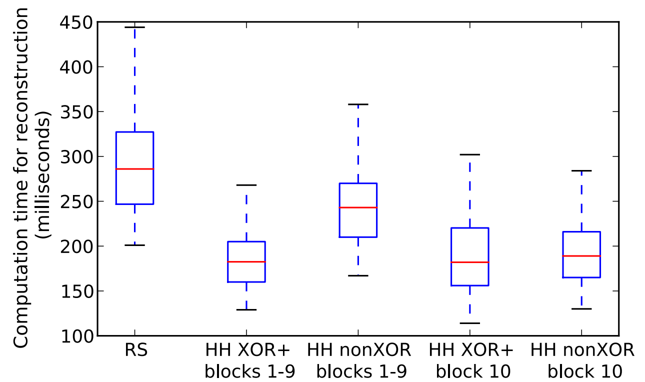


Figure 10: A box plot comparing the computation time for reconstruction of 1MB (buffer size) from 200 runs each on Facebook's data-warehouse cluster with real-time production traffic and workloads. (HH = Hitchhiker.)

We can see that Hitchhiker's erasure codes perform faster reconstruction than RS-based HDFS-RAID for any data block. This is because (recall from §3) the reconstruction operation in Hitchhiker requires performing the resource intensive RS decoding only for *half* the substripes as compared to RS-based HDFS-RAID. In Hitchhiker, the data in the other half of substripes is reconstructed by performing either a few XOR operations (under Hitchhiker-XOR and Hitchhiker-XOR+) or a few finite-field operations (under Hitchhiker-nonXOR). One can also see that Hitchhiker-XOR+ has 25% lower computation time during reconstruction as compared to Hitchhiker-nonXOR for the first nine data blocks; for block 10, the time is almost identical in Hitchhiker-XOR+ and Hitchhiker-nonXOR (as expected from theory (§3)).

6.4 Read time for degraded reads & recovery

Fig. 11a and Fig. 11b respectively compare the median and the 95th percentile of the read times during reconstruction for three different block sizes: 4MB, 64MB, and 256MB in Hitchhiker-XOR+. The read patterns for reconstruction of any of the first nine blocks are identical (§3.2).

In the median read times for reconstruction of blocks 1-9 and block 10 respectively, in comparison to RS-based HDFS-RAID, we observed a reduction of 41.4% and 41% respectively for 4MB block size, 27.7% and 42.5% for 64MB block size, and 31.8% and 36.5% for 256MB block size. For the 95*th* percentile of the read time we observed a reduction of 35.4% and 48.8% for 4MB, 30.5% and 29.9% for 64MB, and 30.2% and 31.2% for 256MB block sizes.

The read pattern of Hitchhiker-nonXOR is identical to Hitchhiker-XOR+, while that of Hitchhiker-XOR is the same for the first six blocks and almost the same for the remaining four blocks. Hence for brevity, we plot the statistics only for Hitchhiker-XOR+.

Although Hitchhiker reads data from more machines as compared to RS-based HDFS-RAID, we see that it gives a superior performance in terms of read latency during reconstruction. The reason is that Hitchhiker reads only *half* a block size from most of the machines it connects to (recall from §5) whereas RS-based HDFS-RAID reads entire blocks.

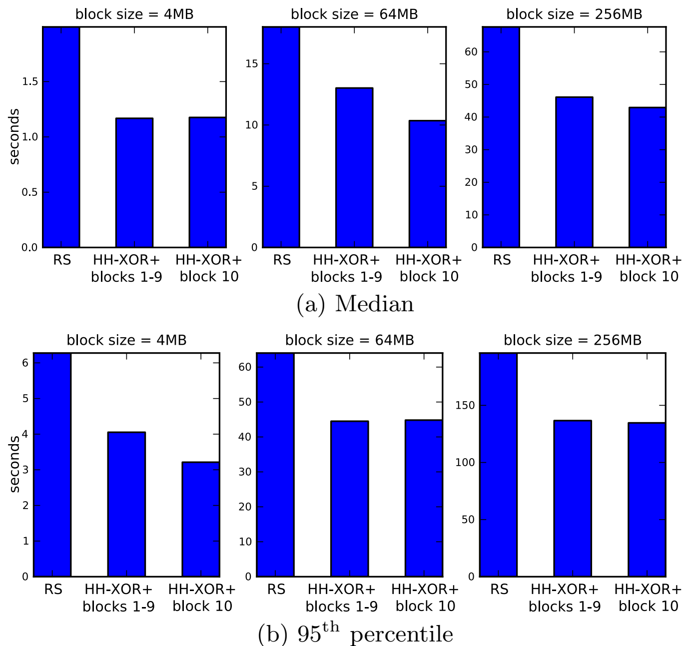


Figure 11: Total read time (in seconds) during reconstruction from 200 runs each on Facebook's data-warehouse cluster with real-time production traffic and workloads. (HH = Hitchhiker.)

6.5 Computation time for encoding

Fig. 12 compares the computation time for the encoding operation. Hitchhiker entails higher computational overheads during encoding as compared to RS-based systems, and Hitchhiker-XOR+ and Hitchhiker-XOR are faster than the Hitchhiker-nonXOR. This is expected since Hitchhiker's encoder performs computations in addition to those of RS encoding. §7 discusses the tradeoffs between higher encoding time and savings in other metrics.

6.6 Statistics of single block reconstruction

Hitchhiker optimizes data reconstruction in scenarios when only one block of a stripe is unavailable. If multiple blocks belonging to a stripe are unavailable, Hitchhiker performs reconstruction in a manner identical to RS-based HDFS-RAID, by reading and downloading 10 entire blocks. We collected measurements of the number of missing blocks per stripe across six months in the data-warehouse cluster in production at Facebook, which stores multiple Petabytes of RS-coded data. We observed that among all the stripes that had at least one block to be reconstructed, 98.08% of them had exactly one such block missing, 1.87% had two blocks missing, and the number of stripes with three or more such blocks was 0.05%. The measurements thus reveal single block reconstructions to be by far the most common scenario in the system at hand.

One might alternatively think of *lazily* performing reconstruction by waiting for multiple blocks in a stripe to fail and then reconstructing them all at once, in order to amortize the cost of disk reads and download during reconstruction. However, such a waiting approach may not be feasible when read performance is key: a failed systematic block must be repaired quickly to serve future read requests. Furthermore, degraded reads are performed on a single block in real time, and there is no equivalent of waiting for more failures.

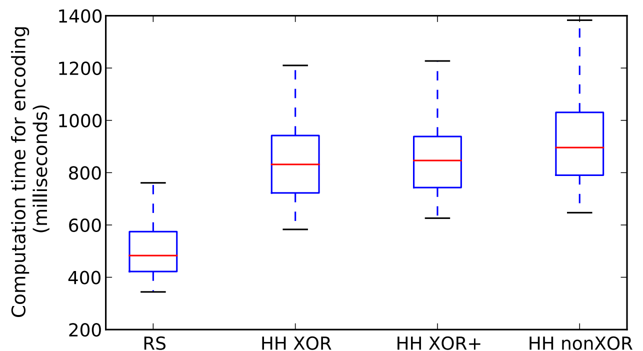


Figure 12: A box plot comparing the computation time for encoding of 1MB (buffer size) from 200 runs each on Facebook's data-warehouse cluster with real-time production traffic and workloads. (HH = Hitchhiker.)

6.7 Deployment and testing

We deployed HDFS-RAID with both Hitchhiker and the RS-based system on a 60-machine test cluster at Facebook in order to verify Hitchhiker's end-to-end functionality. We created multiple files with a block size of 256MB, and encoded them separately using Hitchhiker and RS-based HDFS-RAID. The block placement policy of HDFS-RAID ensures that the 14 blocks of an encoded stripe are all stored on different machines. We then forced some of these machines to become unavailable by stopping HDFS related scripts running on them, and collected the logs pertaining to the MapReduce jobs that performed the reconstruction operations. We verified that all reconstruction operations were successful. We also confirmed that the amount of data read and downloaded in Hitchhiker was 35% lower than in RS-based HDFS-RAID, as guaranteed by the proposed codes. We do not perform timing measurements on the test cluster since the network traffic and the workload on these machines do not reflect the real (production) scenario. Instead, we evaluated these metrics directly on the production cluster itself (as discussed earlier).

7. DISCUSSION ON TRADE-OFFS

a) Three versions of the code: During encoding and reconstruction, Hitchhiker-XOR requires performing only XOR operations in addition to the operations of the underlying RS code. Hitchhiker-XOR+ allows for more efficient reconstruction in terms of network and disk resources in comparison to Hitchhiker-XOR, while still using only XOR operations in addition to the RS code's operations. Hitchhiker-XOR+, however, requires the underlying RS code to satisfy the all-XOR-parity property (§3.2). Hitchhiker-nonXOR provides the same efficiency in reconstruction as Hitchhiker-XOR+ without imposing the all-XOR-parity requirement on the RS code, but entails additional finite-field arithmetic during encoding and reconstruction.

b) Connecting to more machines during reconstruction: Reconstruction in RS-coded systems requires connecting to exactly k machines, whereas Hitchhiker entails connecting to more than k machines. In certain systems, depending on the setting at hand, this may lead to an increase in the read latency during reconstruction. However, in our experiments on the production data-warehouse cluster at Facebook, we saw no such increase in read latency. On the contrary, we

consistently observed a significant reduction in the latency due to the significantly lower amounts of data required to be read and downloaded in Hitchhiker (see Fig. 11).

c) Option of operating as an RS-based system: The storage overheads and fault tolerance of Hitchhiker are identical to RS-based systems. Moreover, a reconstruction operation in Hitchhiker can alternatively be performed as in RS-based systems by downloading any k entire blocks. Hitchhiker thus provides an option to operate as an RS-based system when necessary, e.g., when increased connectivity during reconstruction is not desired. This feature also ensures Hitchhiker's compatibility with other alternative solutions proposed outside the erasure-coding component (e.g., [3,5,18]).

d) Choice of hop-length: As discussed in §4, a larger hop-length leads to more contiguous reads, but requires coupling of bytes that are further apart. Recall that reconstructing any byte also necessitates reconstructing its coupled byte. In a scenario where only a part of a block may need to be reconstructed, all the bytes that are coupled with the bytes of this part must also be reconstructed even if they are not required. A lower hop-length reduces the amount such frivolous reconstructions.

e) Higher encoding time vs. improvement in other metrics: Hitchhiker trades off a higher encoding time for improvement along other dimensions (Table 1). Encoding of raw data into erasure-coded data is a one time task, and is often executed as a background job. On the other hand, reconstruction operations are performed repeatedly, and degraded read requests must be served in real time. For these reasons, the gains in terms of other metrics achieved by Hitchhiker outweigh the additional encoding cost in the systems we consider.

8. RELATED WORK

Erasure codes have many pros and cons over replication [25,31]. The most attractive feature of erasure codes is that while replication entails a minimum of 2× storage redundancy, erasure codes can support significantly smaller storage overheads for the same levels of reliability. Many storage systems thus employ erasure codes for various application scenarios [2, 3, 9, 24]. Traditional erasure codes however face the problem of inefficient reconstruction. To this end, several works (e.g., [3,5,18]) propose system level solutions that can be employed to reduce data transfer during reconstruction operations, such as caching the data read during reconstruction, batching multiple recovery operations in a stripe, or delaying the recovery operations. While these solutions consider the erasure code as a black-box, Hitchhiker modifies this black box, employing the new erasure code of this paper to address the reconstruction problem. Note that Hitchhiker retains all the properties of the underlying RS-based system. Hence any solution proposed outside of the erasure-code module can be employed in conjunction with Hitchhiker to benefit from both the solutions.

The problem of reducing the amount of data accessed during reconstruction through the design of new erasure codes has received much attention in the recent past [7,11,12,17,19,21,22,27,30,32]. However, all existing practical solutions either require the inclusion of additional parity units, thereby increasing the storage overheads [7,12,17,21,27], or are applicable in very limited settings [11,15,30,32].

The idea of connecting to more machines and downloading smaller amounts of data from each node was proposed in [6] as a part of the 'regenerating codes model'. However, all existing practical constructions of regenerating codes necessitate a high storage redundancy in the system, e.g., codes in [21] require $r \geq (k-1)$. Rotated-RS [15] is another class of codes proposed for the same purpose. However, it supports at most 3 parities, and moreover, its fault tolerance is established via a computer search. Recently, optimized recovery algorithms [30,32] have been proposed for EVEN-ODD and RDP codes, but they support only 2 parities. For the parameters where [15,30,32] exist, Hitchhiker performs at least as good, while also supporting an arbitrary number of parities. An erasure-coded storage system which also optimizes for data download during reconstruction is presented in [11]. While this system achieves minimum possible download during reconstruction, it supports only 2 parities. Furthermore, [11] requires decode operation to be performed for every read request since it cannot reconstruct an identical version of a failed unit but only reconstruct a functionally equivalent version.

The systems proposed in [7,12,17] employ another class of codes called local-repair codes to reduce the number blocks accessed during reconstruction. This, in turn, also reduces the total amount of data read and downloaded during reconstruction. However, these codes necessitate addition of at least 25% to 50% more parity units to the code, thereby increasing the storage space requirements.

9. CONCLUSION

We have introduced a systematically-designed, new and novel storage system called *Hitchhiker* that "rides" on top of existing Reed-Solomon based erasure-coded systems. *Hitchhiker* retains the key benefits of RS-coded systems over replication-based counterparts, namely that of (i) optimal storage space needed for a targeted level of reliability, as well as (ii) fine-grained flexibility in the design choice for the system. We show how Hitchhiker can *additionally* reduce both network traffic and disk traffic by 25% to 45% over that of RS-coded systems during reconstruction of missing or otherwise unavailable data. Further, our implementation and evaluation of Hitchhiker on two HDFS clusters at Facebook also reveals savings of 36% in the computation time and 32% in the time taken to read data during reconstruction.

As we look to scale next-generation data centers and cloud storage systems, a primary challenge is that of sustaining this massive growth in the volume of data needing to be stored and retrieved reliably and efficiently. Replication of data, while ideal from the viewpoint of flexible access and efficient reconstruction when faced with missing or unavailable nodes, is clearly not a sustainable option for all but a small fraction of the massive volume of data needing to be stored. Specifically, replication of data costs a redundancy factor of at least 2×. Not surprisingly, therefore, RS-coded systems, which can offer near-arbitrary fine-grained redundancy factors between 1× and 2×, have received more traction in data centers, despite their shortcomings with regard to large network and disk traffic requirements when faced with reconstruction of missing or unavailable data. This underscores the importance of Hitchhiker which aims at getting the best of both worlds in a systematic and scalable manner.

10. REFERENCES

[1] HDFS-RAID. http://wiki.apache.org/hadoop/HDFS-RAID.

[2] Seamless reliability. http://www.cleversafe.com/overview/reliable, Feb. 2014.

[3] R. Bhagwan, K. Tati, Y. C. Cheng, S. Savage, and G. Voelker. Total recall: System support for automated availability management. In *NSDI*, 2004.

[4] D. Borthakur. HDFS and Erasure Codes (HDFS-RAID). http://hadoopblog.blogspot.com/2009/08/hdfs-and-erasure-codes-hdfs-raid.html, Aug. 2009.

[5] B.-G. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, M. F. Kaashoek, J. Kubiatowicz, and R. Morris. Efficient replica maintenance for distributed storage systems. In *NSDI*, 2006.

[6] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. Wainwright, and K. Ramchandran. Network coding for distributed storage systems. *IEEE Trans. Inf. Th.*, Sept. 2010.

[7] K. Esmaili, L. Pamies-Juarez, and A. Datta. CORE: Cross-object redundancy for efficient data repair in storage systems. In *IEEE International Conf. on Big data*, 2013.

[8] B. Fan, W. Tantisiriroj, L. Xiao, and G. Gibson. Diskreduce: RAID for data-intensive scalable computing. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, pages 6–10. ACM, 2009.

[9] B. Fan, W. Tantisiriroj, L. Xiao, and G. Gibson. Diskreduce: RAID for data-intensive scalable computing. In *ACM Workshop on Petascale Data Storage*, 2009.

[10] S. Ghemawat, H. Gobioff, and S. Leung. The Google file system. In *ACM SOSP*, 2003.

[11] Y. Hu, H. C. Chen, P. P. Lee, and Y. Tang. Nccloud: Applying network coding for the storage repair in a cloud-of-clouds. In *USENIX FAST*, 2012.

[12] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure coding in Windows Azure storage. In *USENIX ATC*, 2012.

[13] S. Jiekak, A. Kermarrec, N. Scouarnec, G. Straub, and A. Van Kempen. Regenerating codes: A system perspective. *arXiv:1204.5028*, 2012.

[14] G. Kamath, N. Silberstein, N. Prakash, A. Rawat, V. Lalitha, O. Koyluoglu, P. Kumar, and S. Vishwanath. Explicit MBR all-symbol locality codes. In *ISIT*, 2013.

[15] O. Khan, R. Burns, J. Plank, W. Pierce, and C. Huang. Rethinking erasure codes for cloud file systems: minimizing I/O for recovery and degraded reads. In *FAST*, 2012.

[16] S. Lin and D. Costello. *Error control coding*. Prentice-hall Englewood Cliffs, 2004.

[17] S. Mahesh, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. Xoring elephants: Novel erasure codes for big data. In *VLDB*, 2013.

[18] J. Mickens and B. Noble. Exploiting availability prediction in distributed systems. In *NSDI*, 2006.

[19] D. Papailiopoulos, A. Dimakis, and V. Cadambe. Repair optimal erasure codes through hadamard designs. *IEEE Trans. Inf. Th.*, May 2013.

[20] K. V. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran. A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the Facebook warehouse cluster. In *Proc. USENIX HotStorage*, June 2013.

[21] K. V. Rashmi, N. B. Shah, and P. V. Kumar. Optimal exact-regenerating codes for the MSR and MBR points via a product-matrix construction. *IEEE Trans. Inf. Th.*, 2011.

[22] K. V. Rashmi, N. B. Shah, and K. Ramchandran. A piggybacking design framework for read-and download-efficient distributed storage codes. In *IEEE International Symposium on Information Theory*, 2013.

[23] I. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of SIAM*, 1960.

[24] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: The OceanStore prototype. In *USENIX FAST*, 2003.

[25] R. Rodrigues and B. Liskov. High availability in DHTs: Erasure coding vs. replication. In *IPTPS*, 2005.

[26] N. Shah, K. Rashmi, P. Kumar, and K. Ramchandran. Distributed storage codes with repair-by-transfer and non-achievability of interior points on the storage-bandwidth tradeoff. *IEEE Trans. Inf. Theory*, 2012.

[27] N. B. Shah. On minimizing data-read and download for storage-node recovery. *IEEE Communications Letters*, 2013.

[28] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop distributed file system. In *IEEE MSST*, 2010.

[29] I. Tamo, Z. Wang, and J. Bruck. Zigzag codes: MDS array codes with optimal rebuilding. *IEEE Trans. Inf. Th.*, 2013.

[30] Z. Wang, A. Dimakis, and J. Bruck. Rebuilding for array codes in distributed storage systems. In *ACTEMT*, 2010.

[31] H. Weatherspoon and J. D. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *IPTPS*, 2002.

[32] L. Xiang, Y. Xu, J. Lui, and Q. Chang. Optimal recovery of single disk failure in RDP code storage systems. In *ACM SIGMETRICS*, 2010.

APPENDIX

Hitchhiker-XOR: The encoding procedure of Hitchhiker-XOR first divides the k data units into $(r-1)$ disjoint sets of roughly equal sizes. For instance, in the $(k=10, r=4)$ code of Fig. 4, the three sets are units $\{1,2,3\}$, units $\{4,5,6\}$ and units $\{7,8,9,10\}$. For each set $j \in \{1,\ldots,r-1\}$, the bytes of the first substripe of all units in set j are XORed, and the resultant is XORed with the second substripe of the $(j+1)^{\text{th}}$ parity unit.

Reconstruction of a data unit belonging to any set j requires the bytes of both the substripes of the other data units in set j, only the second byte of all other data units, and the second bytes of the first and $(j+1)^{\text{th}}$ parity units. The decoding procedure for reconstruction of any data unit i is executed in three steps:

Step 1: The k bytes $\{b_1,\ldots,b_k,f_1(\boldsymbol{b})\}\backslash\{b_i\}$ belonging to the second substripe of the units $\{1,\ldots,k+1\}\backslash\{i\}$ are identical to the k corresponding encoded bytes in the underlying RS code. Perform RS decoding of these k bytes to get $\boldsymbol{b}$ (which includes one of the desired bytes b_i).

Step 2: In the other bytes accessed, subtract out all components that involve $\boldsymbol{b}$.

Step 3: XOR the resulting bytes to get a_i.

If the size of a set is s, reconstruction of any data unit in this set requires $(k+s)$ bytes (as compared to $2k$ under RS).

Hitchhiker-XOR+: Assume without loss of generality that, in the underlying RS code, the all-XOR property is satisfied by the second parity. The encoding procedure first selects a number $\ell \in \{0,\ldots,k\}$ and partitions the first $(k-\ell)$ data units into $(r-1)$ sets of roughly equal sizes. On these $(k-\ell)$ data units and r parity units, it performs an encoding identical to that in Hitchhiker-XOR. Next, in the second parity unit, the byte of the second substripe is XORed onto the byte of the first substripe.

Reconstruction of any of the first $(k-\ell)$ data units is performed in a manner identical to that in Hitchhiker-XOR. Reconstruction of any of the last ℓ data units requires the byte of the first substripe of the second parity and the bytes of the second substripes of all other units. The decoding procedure remains identical to the three-step procedure of Hitchhiker-XOR stated above.

For any of the first $(k-\ell)$ data units, if the size of its set is s then reconstruction of that data unit requires $(k+s)$ bytes (as compared to $2k$ under RS). The reconstruction of any of the last ℓ units requires $(k+r+\ell-2)$ bytes (as compared to $2k$ under RS). The parameter ℓ can be chosen to minimize the average or maximum data required for reconstruction as per the system requirements.

Hitchhiker-nonXOR: The encoding procedure is identical to that of Hitchhiker-XOR+, except that instead of XORing the bytes of the first substripe of the data units in each set, these bytes are encoded using the underlying RS encoding function considering all other data units that do not belong to the set as zeros.

The collection of data bytes required for the reconstruction of any data unit is identical to that under Hitchhiker-XOR+. The decoding operation for reconstruction is a three-step procedure. The first two steps are identical to the first two steps of the decoding procedure of Hitchhiker-XOR described above. The third step requires an RS decoding operation (recall the (10, 4) case from §3.3). In particular, the output of the second step when reconstructing a data unit i will be equal to k bytes that would have been obtained from the RS encoding of the data bytes in the units belonging to that set with all other data bytes set to zero. An RS decoding operation performed on these bytes now gives a_i, thus recovering the ith data unit (recall that b_i is reconstructed in Step 1 itself.) The data access patterns during reconstruction and the amount of savings under Hitchhiker-nonXOR are identical to that under Hitchhiker-XOR+.