A Piggybacking Design Framework for Read-and Download-efficient Distributed Storage Codes

K. V. Rashmi, Nihar B. Shah, Kannan Ramchandran, *Fellow, IEEE*

***Abstract*—We present a new *piggybacking* framework for designing distributed storage codes that are efficient in the amount of data read and downloaded during node-repair. We illustrate the power of this framework by constructing explicit codes that attain the smallest amount of data to be read and downloaded for repair among all existing solutions for three important settings: (a) codes meeting the constraints of being maximum distance separable (MDS), high-rate, and having a small number of substripes, (b) binary MDS codes for all parameters where binary MDS codes exist, and (c) MDS codes with the smallest repair-locality. In addition, we show how to use this framework to enable efficient repair of parity nodes in existing codes that are constructed to address the repair of only the systematic nodes. The basic idea behind this framework is to take multiple stripes of existing codes and add carefully designed functions of the data of one stripe to other stripes. Typical savings in the amount of data read and downloaded during repair are 25% to 50% depending on the choice of the system parameters.**

I. Introduction

Distributed storage systems are increasingly employing erasure codes for introducing redundancy, since erasure codes provide much better storage efficiency and reliability as compared to replication. Adding redundancy is essential to combating frequent node failures witnessed by these systems in their day-to-day operation [1]. Upon failure of a node, it is replaced by a new node, and this new node must obtain the data that was stored in the failed node by downloading data from the remaining nodes. Two primary metrics that determine the efficiency of repair are the amount of data read at the remaining nodes and the amount downloaded from them.

The most commonly employed codes in distributed storage systems are Reed-Solomon (RS) codes. These codes require large amounts of data to be read and downloaded during repair, and are hence highly wasteful of system resources. For instance, [1] presents measurements from Facebook's warehouse cluster that stores tens of petabytes of RS-encoded data, showing that the repair of RS-encoded data contributes a median of 180 terabytes of cross-rack traffic per day.

Let n denote the number of (storage) nodes. The data to be stored across these nodes is termed the *message*. A *maximum distance separable (MDS)* code (e.g., the RS code) is associated to another parameter k: an $(n,\ k)$ MDS code guarantees that the message can be recovered from *any* k of the n nodes and requires a storage of $\frac{1}{k}$ of the size of the message at every node. We denote the number of parity nodes by $r=(n-k)$. The rate of a code is defined as the ratio of the size of the message to the total size of the encoded data.

An independent instance of a code is termed a *stripe*: different stripes of a code are identical encodings of different messages. For example, Fig. 1a depicts two stripes of a $(6,4)$ MDS code, one with $\{a_i\}_{i=1}^4$ and the other with $\{b_i\}_{i=1}^4$ as the message. The *number of substripes* of a (vector or array) code is defined as the length of the vector of symbols that a node stores in a single stripe of the code. For example, the number of substripes of the code in Fig. 1a is 1. The code in Fig. 1b (details of which will be discussed later) depicts one stripe of a code with two substripes: the two columns together constitute a single stripe since they are not identical encodings.

Under the codes and repair algorithms typically employed in distributed storage systems, each node stores only a fraction $\frac{1}{k}$ of the message, but the repair of a node requires the *entire message* to be read and downloaded.

In this paper, we present a new *piggybacking* framework for the design of repair-efficient storage codes. In a nutshell, this framework considers multiple stripes of an existing code, and performs *piggybacking* on it, i.e., adds (carefully designed)

	Base Code		Intermediate Step		Piggybacked Code	
Node 1	a_1	b_1	a_1	b_1	a_1	b_1
Node 2	a_2	b_2	a_2	b_2	a_2	b_2
Node 3	a_3	b_3	a_3	b_3	a_3	b_3
Node 4	a_4	b_4	a_4	b_4	a_4	b_4
Node 5	$\sum_{i=1}^4 a_i$	$\sum_{i=1}^4 b_i$	$\sum_{i=1}^4 a_i$	$\sum_{i=1}^4 b_i$	$\sum_{i=1}^4 a_i$	$\sum_{i=1}^4 b_i$
Node 6	$\sum_{i=1}^4 ia_i$	$\sum_{i=1}^4 ib_i$	$\sum_{i=1}^4 ia_i$	$\sum_{i=1}^4 ib_i+\sum_{i=1}^2 ia_i$	$\sum_{i=3}^4 ia_i-\sum_{i=1}^4 ib_i$	$\sum_{i=1}^4 ib_i+\sum_{i=1}^2 ia_i$

(a) (b) (c)

Fig. 1: A (6,4) MDS code that can perform repair of systematic nodes efficiently, constructed using the piggybacking framework. Shaded cells indicate symbols modified by the piggybacking.

The authors are with the Dept. of EECS, UC Berkeley. E-mail: {rashmikv, nihar, kannanr}@eecs.berkeley.edu. K. V. Rashmi was supported by a Facebook Fellowship and N. B. Shah was supported by a Berkeley Fellowship. This work was also supported in part by NSF grant CCF-0964018.

Node 1	a_1	b_1	c_1	d_1
Node 2	a_2	b_2	c_2	d_2
Node 3	a_3	b_3	c_3	d_3
Node 4	a_4	b_4	c_4	d_4
Node 5	$\sum_{i=1}^{4} a_i$	$\sum_{i=1}^{4} b_i$	$\sum_{i=1}^{4} c_i + \sum_{i=1}^{4} ib_i + \sum_{i=1}^{2} ia_i$	$\sum_{i=1}^{4} d_i$
Node 6	$\sum_{i=3}^{4} ia_i - \sum_{i=1}^{4} ib_i$	$\sum_{i=1}^{4} ib_i + \sum_{i=1}^{2} ia_i$	$\sum_{i=3}^{4} ic_i - \sum_{i=1}^{4} id_i$	$\sum_{i=1}^{4} id_i + \sum_{i=1}^{2} ic_i$

Fig. 2: A code for efficient repair of parity nodes constructed using the piggybacking framework. It uses two stripes of the code of Fig. 1c and a piggyback is added from the first stripe to the second stripe.

functions of the data of one stripe to the other. We design these functions with the goal of reducing the amount of read and download performed during repairs. Piggybacking preserves many of the properties of the underlying code, such as the minimum distance and the field of operation.

The piggybacking framework offers a rich design space for constructing codes for various settings. We illustrate the power of this framework by providing the following four classes of explicit code constructions.

(Class 1) A class of codes meeting the constraints of being MDS, high-rate, and having a small number of substripes, with the smallest known average amount of data read and downloaded for repair: The cost of many current day data-centers is dominated by the cost of the storage capacity [1]. In these storage-limited systems, it is critical to minimize the storage overhead incurred in achieving the desired level of reliability. In light of this, it is important for the erasure code employed to be MDS and have high-rate (typically $\gtrsim 2/3$). In addition, for suitability of implementation in these systems, it is desirable for the code to have a small number of substripes. There has recently been considerable work [2]–[14] on the design of distributed storage codes that require a smaller amount of data to be read and downloaded during repair. However, most of these constructions do not meet the aforementioned requirements because they are either non-MDS [11]–[14] or low-rate [2], [7] or need the number of substripes to be exponential in the system parameters [3]–[6]. To the best of our knowledge, the only explicit codes that meet the stated requirements are the Rotated-RS [8] codes and the (repair-optimized) EVENODD [9] and RDP [10] codes. Rotated-RS codes exist only for $r \in \{2, 3\}$ and $k \leq 36$, and the (repair-optimized) EVENODD and RDP codes exist only for $r = 2$. Using the piggybacking framework, we construct a class of codes that are MDS, high-rate, have a small number of substripes, and require the least amount of data read and downloaded for repair among all other known codes in this setting. Furthermore, our codes support all values of the system parameters n and k. We are currently implementing one of these codes in the Hadoop Distributed File System (HDFS) [1], the most popular distributed storage framework employed in the industry today.

(Class 2) Binary MDS codes with the lowest known average amount of data read and download for repair, for all parameters where binary MDS codes exist: Binary MDS codes are extensively used in disk arrays. Through the piggybacking framework, we construct binary MDS codes that attain (to the best of our knowledge) the lowest known average amount of data read and download for repair among all existing binary MDS codes [8]–[10]. Our codes support all parameters for which binary MDS codes are known to exist.

(Class 3) Repair-efficient MDS codes with smallest possible repair-locality: Repair-locality is the number of nodes that need to be contacted during the repair of a node. While several recent works [11]–[14] present codes optimizing on locality, these codes are not MDS, and hence mandate additional storage overheads. In an MDS code, $(k + 1)$ is the smallest possible locality that permits any reduction in the amount of data to be read or downloaded. Previous constructions [3]–[5] of MDS codes with this locality exist only for $r = 2$ parities. In this paper, we present MDS codes with efficient repair properties with a locality of $(k + 1)$, which support an arbitrary number of parity nodes.

(Class 4) A method for reducing the amount of data read and downloaded during repair of parity nodes in existing codes that address only the repair of systematic nodes: The problem of efficient node-repair in distributed storage systems has attracted considerable attention in the recent past. However, many of the codes proposed [3]–[7] have algorithms for efficient repair of *only* the systematic nodes, and require the download of the entire message for repair of any parity node. In this paper, we show how the piggybacking framework can be used as a tool to enable efficient repair of parity nodes in such codes, while retaining efficiency in the repair of systematic nodes. The resulting piggybacked codes enable 25% to 50% savings in the amount of data read and download required for repair of parity nodes, depending on the choice of the system parameters.

Two Examples: The following examples highlight the key ideas behind the piggybacking framework.

Example 1: This example illustrates one method of piggybacking for reducing the amount of data read and downloaded during systematic node repair. Consider two stripes of a $(6, 4)$ MDS code as shown in Fig. 1a. The first step of piggybacking involves adding $\sum_{i=1}^{2} ia_i$ to the second symbol of node 6 as shown in Fig. 1b. The second step in this construction involves subtracting the second symbol of node 6 in the code of Fig. 1b from its first symbol. The resulting code is shown in Fig. 1c.

We now present the repair algorithm for the piggybacked code of Fig. 1c. Consider repair of node 1. Under our repair algorithm, the symbols b_2, b_3, b_4 and $\sum_{i=1}^{4} b_i$ are downloaded from the other nodes, and b_1 is decoded. The symbol $(\sum_{i=1}^{4} ib_i + \sum_{i=1}^{2} ia_i)$ of node 6 is also downloaded. Subtracting out $\{b_i\}_{i=1}^{4}$ gives the piggyback $\sum_{i=1}^{2} ia_i$. Symbol a_2 is downloaded from node 2 and subtracted to obtain a_1. Thus, node 1 is repaired by reading only 6 symbols which is 25% less than the size of the message. Node 2 is repaired in *a similar manner. Repair of nodes* 3 *and* 4 *follow on similar lines except that the first symbol of node* 6 *is read instead of the second.*

One can verify that the entire message can be recovered from any 4 *nodes. Thus the piggybacked code is MDS.*

Example 2: This example illustrates the use of piggybacking to reduce the amount of data read and downloaded during repair of parity nodes. The code depicted in Fig. 2 takes two stripes of the code of Fig. 1c, and adds the 2^{nd} *symbol of node* 6, $(\sum_{i=1}^{4} ib_i + \sum_{i=1}^{2} ia_i)$ *(which belongs to the first stripe), to the* 3^{rd} *symbol* $\sum_{i=1}^{4} c_i$ *of node* 5 *(which belongs to the second stripe). Repair of node* 6 *involves downloading* $\{a_i,c_i,d_i\}_{i=1}^{4}$ *and symbol* $(\sum_{i=1}^{4} c_i + \sum_{i=1}^{4} ib_i + \sum_{i=1}^{2} ia_i)$*, using which the requisite data can be recovered. Thus, the repair of node* 6 *requires read and download of only* 13 *symbols instead of the entire message of size* 16*. Node* 5 *is repaired by downloading all* 16 *message symbols. The modification in Fig. 2 does not change the algorithm or the efficiency of the repair of systematic nodes: the modification only changes the first symbol of node* 5*, which is never used in the repair of any systematic node in the code of Fig. 1c. The code retains the MDS property: the entire message can be recovered from any* 4 *nodes by first decoding* $\{a_i,b_i\}_{i=1}^{4}$ *using the decoding algorithm of the code of Fig. 1c, which then allows for removal of the piggyback* $(\sum_{i=1}^{4} ib_i + \sum_{i=1}^{2} ia_i)$ *from the second stripe, making the remainder identical to the code of Fig. 1c.*

The rest of the paper is organized as follows. Section II describes the general piggybacking framework. Section III then presents explicit codes and repair algorithms using this framework for the aforementioned problems. Section IV draws conclusions and discusses open problems.

II. The Piggybacking Framework

The piggybacking framework operates on an existing code, which we term the *base code*. The choice of the base code is arbitrary. The base code is associated to n encoding functions $\{f_i\}_{i=1}^{n}$, which encode any message $\mathbf{u}$ to n coded symbols $\{f_1(\mathbf{u}),\ldots,f_n(\mathbf{u})\}$. Node i $(1 \leq i \leq n)$ stores the data $f_i(\mathbf{u})$.

The piggybacking framework operates on multiple stripes of the base code. Consider α stripes of the base code. Letting $\mathbf{a}$, …, $\mathbf{z}$ denote the (independent) messages encoded under these α stripes, the encoded symbols in the α stripes of the base code can be written as in Fig. 3a.

We now describe the *piggybacking* of this code. For every $i \in \{2,\ldots,\alpha\}$, one can add an arbitrary function of the message symbols of all the previous stripes, $\{1,\ldots,(i-1)\}$, to the data stored under stripe i. These functions are termed *piggyback* functions, and the values so added are termed *piggybacks*. Denoting the piggyback functions by $g_{i,j}$ $(i \in \{2,\ldots,\alpha\}$, $j \in \{1,\ldots,n\})$, the resulting code is shown in Fig. 3b

The decoding properties (such as the minimum distance or the MDS property) of the base code are retained upon piggybacking. In particular, the piggybacked code allows for decoding the message from any set of nodes from which the base code allowed decoding. To see this, observe that the first substripe of the piggybacked code is identical to a single stripe of the base code. Thus the message $\mathbf{a}$ can be recovered directly using the decoding procedure of the base code. The piggyback functions $\{g_{2,i}(\mathbf{a})\}_{i=1}^{n}$ in the second substripe can now be subtracted out. The remainder of this substripe is precisely another stripe of the base code, allowing recovery of message $\mathbf{b}$. Continuing in this fashion, for any stripe i $(2 \leq i \leq n)$, the piggybacks (which are always a function of the previously decoded substripes $\{1,\ldots,i-1\}$) can be subtracted out to obtain the corresponding stripe of the base code, which can then be decoded.

The decoding properties of the code are thus not hampered by the choice of the piggyback functions $g_{i,j}$'s. This allows for an arbitrary choice of the piggyback functions, and these need to be picked cleverly to achieve the desired goals (such as efficient repair, which is the focus of this paper).

The piggybacking procedure described above was followed in Example 1 to obtain the code of Fig. 1b from the base code. This procedure was followed again in Example 2 to obtain the code of Fig. 2 with the code of Fig. 1c as the base code.

The piggybacking framework also allows *any* invertible linear transformation of the data stored in any individual node. In other words, each node of the piggybacked code (e.g., each row in Fig. 1b) can separately undergo *any* invertible transformation. Clearly, upon any invertible transformation of data within the nodes, the message can be recovered from any set of nodes from which it could be recovered in the base code. In Example 1, the code of Fig. 1c is obtained from Fig. 1b via an invertible transformation of the data of node 6.

Theorem 1 below formally proves that piggybacking does not reduce the amount of information stored in any set of nodes. The proof is available in [15].

Theorem 1: Let $U_1,\ldots,U_\alpha$ be random variables corresponding to the messages associated to the α stripes of the base code. For $i \in \{1,\ldots,n\}$, let X_i denote the (encoded) data stored in node i under the base code. Let Y_i denote the (encoded) data stored in node i upon piggybacking of that base code. Then for any subset of nodes $S \subseteq \{1,\ldots,n\}$,

$$I\left(\{Y_i\}_{i\in S};U_1,\ldots,U_\alpha\right) \geq I\left(\{X_i\}_{i\in S};U_1,\ldots,U_\alpha\right) \ . \qquad (1)$$

Corollary 2: Piggybacking a code does not decrease its minimum distance; piggybacking an MDS code preserves the MDS property.

III. Piggybacking Function and Code Design

We first provide three explicit piggybacking designs, which we call designs 1, 2 and 3. The savings achieved under the

Node 1	$f_1(\mathbf{a})$	$f_1(\mathbf{b})$	$\cdots$	$f_1(\mathbf{z})$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
Node n	$f_n(\mathbf{a})$	$f_n(\mathbf{b})$	$\cdots$	$f_n(\mathbf{z})$

(a) Base code

$f_1(\mathbf{a})$	$f_1(\mathbf{b}) + g_{2,1}(\mathbf{a})$	$f_1(\mathbf{c}) + g_{3,1}(\mathbf{a},\mathbf{b})$	$\cdots$	$f_1(\mathbf{z}) + g_{\alpha,1}(\mathbf{a},\ldots,\mathbf{y})$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$f_n(\mathbf{a})$	$f_n(\mathbf{b}) + g_{2,n}(\mathbf{a})$	$f_n(\mathbf{c}) + g_{3,n}(\mathbf{a},\mathbf{b})$	$\cdots$	$f_n(\mathbf{z}) + g_{\alpha,n}(\mathbf{a},\ldots,\mathbf{y})$

(b) Piggybacked code

Fig. 3: The general piggybacking framework.

three designs are then plotted and compared to existing codes. Using an RS code (or any other high-rate MDS code) as the base code in designs 1 and 2 results in the first class of codes promised in Section I, and using the EVENODD or RDP (or any other binary MDS code) as the base code in these designs leads to the second class. The third class is obtained by using any MDS code in design 3. We then show how the piggybacking framework, with existing regenerating codes as base codes, result in the fourth class.

A. *Piggybacking Design 1*

This design reduces the amount of data read and downloaded during repair while having a small number of substripes. For instance, even when the number of substripes is as small as 2, we can achieve savings of 25% to 35% during repair of systematic nodes. We shall first present the design optimizing the repair of systematic nodes, and subsequently address repair of parity nodes.

For simplicity of exposition, we assume that the base codes are linear, scalar, MDS and systematic. Using vector codes as base codes is a straightforward extension. The base code operates on a k-length message vector, with each symbol of this vector drawn from some finite field. We use $\{\mathbf{a},\mathbf{b},\ldots\}$ to denote the k-length message vectors corresponding to different stripes of the base code. Since the code is systematic, node i $(1 \le i \le k)$ stores $\{a_i,b_i,\ldots\}$ under the base code. We use $\mathbf{p}_1,\ldots,\mathbf{p}_r$ to denote the r encoding vectors corresponding to the r parities, i.e., if $\mathbf{a}$ denotes the k-length message vector then the r parity nodes under the base code store $\{\mathbf{p}_1^T\mathbf{a},\mathbf{p}_1^T\mathbf{b},\ldots\},\ldots,\{\mathbf{p}_r^T\mathbf{a},\mathbf{p}_r^T\mathbf{b},\ldots\}$.

1) Efficient repair of systematic nodes: This design operates on $\alpha = 2$ stripes of the base code. The message comprises of $2k$ symbols $\{\mathbf{a},\mathbf{b}\}$. We first partition the k systematic nodes into r sets, $S_1,\ldots,S_r$, of equal size (or nearly equal size if k is not a multiple of r). For ease of understanding, let us assume that k is a multiple of r, which fixes the size of each of these sets as $\frac{k}{r}$. Then, let $S_1 = \{1,\ldots,\frac{k}{r}\}$, $S_2 = \{\frac{k}{r}+1,\ldots,\frac{2k}{r}\}$, and so on.

Let $\mathbf{p}_r^T = [p_{r,1},\ldots,p_{r,k}]$. Define $k-$length vectors $\mathbf{q}_2,\ldots,\mathbf{q}_r$ as the following projections of $\mathbf{p}_r$:

$$\begin{aligned}
\mathbf{q}_2 &= [p_{r,1} \cdots p_{r,\frac{k}{r}} \; 0 \qquad\qquad \cdots \qquad\qquad 0]\\
\mathbf{q}_3 &= [\;0 \;\; \cdots \;\; 0 \;\; p_{r,\frac{k}{r}+1} \cdots p_{r,\frac{2k}{r}} \; 0 \qquad \cdots \qquad 0]\\
&\;\;\vdots \qquad\qquad \vdots\\
\mathbf{q}_r &= [\;0 \qquad \cdots \qquad 0 \;\; p_{r,\frac{k}{r}(r-2)+1} \;\cdots\; p_{r,\frac{k}{r}(r-1)} \; 0 \cdots 0].
\end{aligned}$$

The base code is piggybacked in the following manner:

Node 1	a_1	b_1
$\vdots$	$\vdots$	$\vdots$
Node k	a_k	b_k
Node k+1	$\mathbf{p}_1^T\mathbf{a}$	$\mathbf{p}_1^T\mathbf{b}$
Node k+2	$\mathbf{p}_2^T\mathbf{a}$	$\mathbf{p}_2^T\mathbf{b}+\mathbf{q}_2^T\mathbf{a}$
$\vdots$	$\vdots$	$\vdots$
Node k+r	$\mathbf{p}_r^T\mathbf{a}$	$\mathbf{p}_r^T\mathbf{b}+\mathbf{q}_r^T\mathbf{a}$

The code in Fig. 1b is a result of such a piggybacking.

The second step in the design takes an invertible transformation of the two symbols stored in node $(k+r)$: the first symbol of node $(k+r)$ in the code above is replaced with the difference of this symbol from its second symbol, i.e., node $(k+r)$ now stores

Node k+r	$\mathbf{p}_r^T\mathbf{a}-(\mathbf{p}_r^T\mathbf{b}+\mathbf{q}_r^T\mathbf{a})$	$\mathbf{p}_r^T\mathbf{b}+\mathbf{q}_r^T\mathbf{a}$

Note that $\big(\mathbf{p}_r^T\mathbf{a}-(\mathbf{p}_r^T\mathbf{b}+\mathbf{q}_r^T\mathbf{a})\big) =$
$\Big([p_{r,1} \;\; \cdots \;\; p_{r,\frac{k}{r}(r-2)} \; 0\cdots0 \; p_{r,\frac{k}{r}(r-1)+1}\cdots p_{r,k}]^T\mathbf{a}-\mathbf{p}_r^T\mathbf{b}\Big)$.
All other symbols in the code remain intact. The code in Fig. 1c is a result of such a step.

We now present the algorithm for repair of any systematic node ℓ $(\in \{1,\ldots,k\})$. This entails recovery of the two symbols a_ℓ and b_ℓ from the remaining nodes. We will use the fact that each $p_{i,j}$ is non-zero as the base code is MDS.

Case 1 ($\ell \notin S_r$): Without loss of generality let $\ell \in S_1$. The k symbols $\{b_1,\ldots,b_{\ell-1},b_{\ell+1},\ldots,b_k,\,\mathbf{p}_1^T\mathbf{b}\}$ are downloaded from the remaining nodes, and the entire vector $\mathbf{b}$ is decoded (using the MDS property of the base code). It now remains to recover a_ℓ. The symbol $(\mathbf{p}_2^T\mathbf{b}+\mathbf{q}_2^T\mathbf{a})$ is downloaded from node $(k+2)$, and $\mathbf{p}_2^T\mathbf{b}$ is subtracted out to obtain the piggyback $\mathbf{q}_2^T\mathbf{a}$. The symbols $\{a_i\}_{i\in S_1\setminus\{\ell\}}$ are also downloaded from the other systematic nodes in set S_1. The specific structure of $\mathbf{q}_2$ allows for recovering a_ℓ from these downloaded symbols. Thus the total amount of data read and downloaded during the repair of node ℓ is $(k+\frac{k}{r})$.

Case 2 *($S = S_r$):* As in the previous case, $\mathbf{b}$ is completely decoded by downloading $\{b_1,\ldots,b_{\ell-1},b_{\ell+1},\ldots,b_k,\,\mathbf{p}_1^T\mathbf{b}\}$. The first symbol $\big(\mathbf{p}_r^T\mathbf{a}-(\mathbf{p}_r^T\mathbf{b}+\mathbf{q}_r^T\mathbf{a})\big)$ of node $(k+r)$ is also downloaded. The second symbols $\{\mathbf{p}_i^T\mathbf{b}+\mathbf{q}_i^T\mathbf{a}\}_{i\in\{2,\ldots,r-1\}}$ of parities $(k+2),\ldots,(k+r-1)$ are also downloaded, and subtracted from the first symbol of node $(k+r)$. This gives $(\tilde{\mathbf{p}}^T\mathbf{a}+\tilde{\mathbf{q}}^T\mathbf{b})$ for some arbitrary $\tilde{\mathbf{q}}$ and with $\tilde{\mathbf{p}} = [0 \;\cdots\; 0 \; p_{r,\frac{k}{r}(r-1)+1} \;\cdots\; p_{r,k}]^T\mathbf{a}$. Finally, a_ℓ is recovered by downloading $\{a_{\frac{k}{r}(r-1)+1},\ldots,a_k\}\setminus\{a_\ell\}$ from other systematic nodes in S_r and using $\mathbf{b}$. The total amount of data read and downloaded in this case is $(k+\frac{k}{r}+r-2)$.

Observe that the repair of systematic nodes in S_r require a greater amount of data read and download as compared to those in the other sets. Hence, we choose the sizes of these sets so as to minimize the average amount of data read and download required. For $i = 1,\ldots,r$, denoting the size of the set S_i by t_i, the optimal sizes of the sets are $t_1 = \cdots = t_{r-1} = \lceil \frac{k}{r} + \frac{r-2}{2r} \rceil := t$ and $t_r = k-(r-1)t$. The amount of data read and downloaded for repair of any systematic node in the first $(r-1)$ sets is $(k+t)$, and that in the last set is $(k+t_r+r-2)$. Thus, the amount γ_1^{sys} of data read and downloaded on average for repair of systematic nodes, as a fraction of the total number $2k$ of message symbols, is

$$\gamma_1^{\text{sys}} = \frac{1}{2k^2}\left[(k-t_r)(k+t)+t_r(k+t_r+r-2)\right] \; .$$

2) Reducing amount of data read and downloaded during repair of parity nodes: The code constructed in Section III-A1 can be piggybacked to introduce efficiency in the repair of parity nodes as well. Due to lack of space, details are relegated to [15]. The amount γ_1^{par} of data read and downloaded on average for the repair of parity nodes, as a fraction of the total message symbols, is

$$\gamma_1^{\text{par}} = \frac{1}{2kr}\left[2k + (r-1)\left(\left(1+\frac{1}{m}\right)k + \left(1-\frac{1}{m}\right)(r-1)\right)\right]$$

B. Piggybacking Designs 2 and 3

Piggybacking design 2 provides a higher efficiency of repair as compared to design 1. However, design 2 also requires a greater number of substripes: the minimum number of substripes required under the design of Section III-A1 is 2 and under that of Section III-A2 is 4, while design 2 requires $(2r-3)$. Design 2 also requires $r \geq 3$. Design 3 focuses on the locality of repair, and supports all choices of system parameters. Due to lack of space, details of these constructions are relegated to [15].

C. Comparison between different codes

We now compare the repair-efficiency of the piggyback constructions with other codes. As discussed in Section I, we are interested in codes that are MDS, high-rate, and have a small number of substripes. Rotated-RS codes, (repair-optimized) EVENODD and RDP codes, and the piggyback codes (with RS codes as the base codes) satisfy these conditions. Fig. 4 shows a plot comparing the repair properties of these codes. Here, piggyback 1 has 8 substripes, piggyback 2 has $4(2r-3)$, piggyback 3 has 16, and the rotated-RS code has 8. One can see that piggyback codes require a lesser (average) amount of data read and download as compared to the other codes.

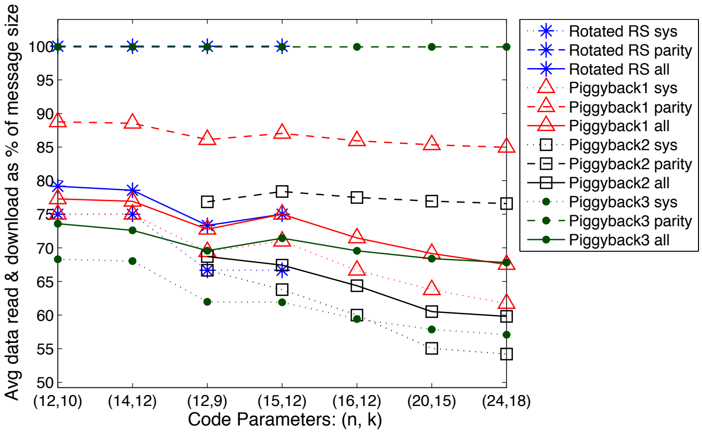

Fig. 4: Average amount of data read and downloaded during repair of systematic, parity and all nodes in the three piggybacking designs and in Rotated-RS codes [8]. The performance of [9], [10] for EVENODD and RDP codes are identical to that of rotated-RS codes (with $r = 2$).

D. Efficient parity repair in regenerating codes

Many codes [3]–[7] constructed under the *regenerating codes* model [16] can repair only the systematic nodes efficiently, and require the download of the entire message for repair of any parity node. In this section, we show how the piggybacking framework can be employed as a tool to reduce the read and download during parity-repair (by $\approx 25\%$ to 50%), while retaining the efficiency in the repair of systematic nodes. We discuss only the key ideas here due to lack of space, and refer the reader to [15] for more details.

The codes in [3]–[7] possess the following property. For the repair of any systematic node $i \in \{1,\ldots,k\}$, each parity node passes the inner product of its own data with some matrix A_i that depends only on i. Consider any such regenerating code as the base code $\mathcal{C}_B$. Consider $2m$ stripes of code $\mathcal{C}_B$, for some $m \geq 1$, and apply the piggybacking design of Section III-A2. Repair of a systematic node can continue to be accomplished using the repair algorithm of $\mathcal{C}_B$, for the following reason. For repair of any systematic node i, each stripe of each parity node passes the inner product with A_i. Since A_i is the same across all the stripes and all the nodes, the linear combinations added for piggybacking can be inverted at the new node, thus obtaining data identical to what is obtained under $\mathcal{C}_B$. The average amount of data read and downloaded for repair of a parity node is identical to that obtained under the design of Section III-A2.

IV. Conclusions And Open Problems

We have presented a new *piggybacking* framework for constructing repair-efficient storage codes. We have provided a few designs of piggybacking and specialized it to existing codes to obtain four classes of code constructions. We believe that this simple framework has much greater potential: clever designs of other piggybacking functions and application to other base codes could potentially lead to efficient codes for various other settings as well. Further exploration of this rich design space is a part of our future work. Finally, while this paper presented only achievable schemes for reducing the amount of data read and downloaded during repair, determining the optimal repair-efficiency under the settings considered remains open.

References

[1] 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*, Jun. 2013.

[2] 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.*, Aug. 2011.

[3] Z. Wang, I. Tamo, and J. Bruck, "Long mds codes for optimal repair bandwidth," in *ISIT*, 2012.

[4] V. Cadambe, C. Huang, and J. Li, "Permutation code: optimal exact-repair of a single failed node in MDS code based distributed storage systems," in *IEEE ISIT*, 2011.

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

[6] V. Cadambe, C. Huang, J. Li, and S. Mehrotra, "Polynomial length MDS codes with optimal repair in distributed storage," 2011.

[7] N. B. Shah, K. V. Rashmi, P. V. Kumar, and K. Ramchandran, "Explicit codes minimizing repair bandwidth for distributed storage," in *Proc. IEEE ITW*, Cairo, Jan. 2010.

[8] 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 *USENIX FAST*, 2012.

[9] Z. Wang, A. G. Dimakis, and J. Bruck, "Rebuilding for array codes in distributed storage systems," in *ACTEMT*, Dec. 2010.

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

[11] P. Gopalan, C. Huang, H. Simitci, and S. Yekhanin, "On the locality of codeword symbols," *IEEE Trans. on Inf. Th.*, Nov. 2012.

[12] D. Papailiopoulos and A. Dimakis, "Locally repairable codes," in *ISIT*, 2012.

[13] G. M. Kamath, N. Prakash, V. Lalitha, and P. V. Kumar, "Codes with local regeneration," *arXiv:1211.1932*, 2012.

[14] F. Oggier and A. Datta, "Self-repairing homomorphic codes for distributed storage systems," in *INFOCOM, 2011 Proceedings IEEE*, 2011, pp. 1215–1223.

[15] K. V. Rashmi, N. B. Shah, and K. Ramchandran, "A piggybacking design framework for read-and download-efficient distributed storage codes," 2013. [Online]. Available: arXiv:1302.5872

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