Efficient Updates in Cross-Object Erasure-Coded Storage Systems

Kyumars Sheykh Esmaili
School of Computer Engineering
Nanyang Technological University
Singapore
Email: kyumarss@ntu.edu.sg

Aatish Chiniah
Computer Science and Engineering Department
University of Mauritius
Mauritius
Email: a.chiniah@uom.ac.mu

Anwitaman Datta
School of Computer Engineering
Nanyang Technological University
Singapore
Email: anwitaman@ntu.edu.sg

***Abstract*—In the past few years erasure codes have been increasingly embraced by distributed storage systems as an alternative for replication, since they provide high fault-tolerance for low overheads. Erasure codes, however, have few shortcomings that need to be addressed to make them a complete solution for networked storage systems. Lack of support for efficient data *repair* and data *update* are the two most notable shortcomings.**

We recently proposed to use a 2-dimensional product code – Reed-Solomon coding per object and simple XORing across objects – and showed that at a reasonable storage overhead, it can greatly reduce the repair cost. In this paper we propose an efficient approach to handle data updates in cross-object erasure-coded storage systems. Our proposed solution has been implemented and experimentally evaluated. Our results show that compared to the naive approach (re-encoding the data), our proposed scheme can considerably decrease the update cost, especially for when the number of updated blocks is small.

I. Introduction

In the past few years erasure codes, most prominently Reed Solomon (RS) codes, have been increasingly embraced by distributed storage systems as an alternative for replication. In the coding scheme of RS(n,k), an object consisting of k blocks is encoded into n blocks ($k < n$) in a way that the original object can be recreated from any k subset of the n encoded pieces.

The main advantage of erasure codes compared to replication is that they provide higher fault-tolerance for lower overheads. However, as erasure codes were originally designed for a different environment (error control in transmission of one-time messages over an erasure channel), they do not consider two of the essential constraints/properties of distributed storage systems: (i) data is scattered among a large number of storage nodes connected through a network with limited bandwidth, and (ii) data has a long lifespan, during which its content may be updated. These constraints, result in several shortcomings that need to be addressed to make erasure codes a complete solution for networked storage systems. Two of the most notable such shortcomings are lack of support for efficient data **repair** and **update**.

The naive approach to address these problems are costly. More specifically, to repair one single block in a storage system encoded with RS(n,k), k other blocks must be fetched and then decoded. Likewise, upon updating one single block in the same system, all the k blocks are fetched and re-encoded again. These are clearly inefficient solutions, especially in cases where the repair/update size (i.e., number of affected data blocks) is small, and the costs are not amortized.

Recently, we proposed [1] to use cross-object redundancy to reduce repair cost by combining simple and mature techniques – juxtaposing RS codes with RAID-4 like parity. This proposal was later realized [2] as CORE storage primitive (hereafter CORE) and its benefits in terms of repairability over the other alternatives were demonstrated through analytical and experimental studies.

In this paper we aim at addressing the update problem in CORE by designing an update scheme that is more efficient than re-encoding the whole data. At a very high level, our solution belongs to the family of **parity update** solutions [3], [4], [5] in which first data *diff-blocks* –i.e., the difference between the old and the new versions of the updated blocks– are computed and then they are used to compute the new versions of the affected parity blocks. However, as we argue in Section II-C, apart from the general idea, the existing proposals on parity update are not applicable to CORE and the representation of RS codes that it uses. To address these issues we make the following contributions in this paper:

- we define the theoretical foundations for parity update in the *generator polynomial* representation of RS codes,
- we implement the re-encoding as well the parity update approaches and integrate them into the CORE system,
- we study the effectiveness of the aforementioned update techniques analytically and experimentally.

We would like to note that the applicability of theoretical aspects reported in this paper (particularly, the first contribution in the list above) is not limited to the CORE system and can in fact be adopted by any system that uses similar codes/representation, and specifically this includes the standard HDFS-RAID [6] implementation.

The rest of this paper is structured as follows. We first give a short overview of the relevant background in Section II. Then we explain the details of CORE's update handling approaches in Section III. A brief description of the implementation is

This work was supported by A*Star SERC Grant No: 102 158 0038. Aatish Chiniah contributed to this work while he was a student at NTU.

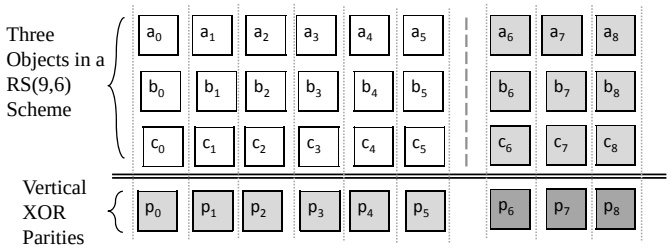


Fig. 1: An example illustrating the basic idea of CORE

given in Section IV and the subsequent experimental results are presented in Section V. Finally, the paper is concluded in Section VI.

II. Background

This section gives a short overview of the background i.e., the CORE storage primitive (II-A), the two representations of RS codes (II-B), and a very brief survey of the related work on efficient updates in erasure coded storage systems (II-C).

A. The CORE Storage Primitive

CORE builds upon a simple observation [1]: by introducing a RAID-4 like parity over t erasure encoded pieces –resulting in what's called CORE matrix of size *(n,k,t)*– it is possible to achieve significant reduction in the expected cost to repair missing/corrupt blocks. In fact, in the average case, not only fewer blocks are needed to carry out repairs but also the required computations are simpler and cheaper (XOR instead of RS decoding).

Figure 1 shows an example of CORE(9,6,3): three distinct objects – a, b, and c – each comprising of 6 blocks are first individually encoded using a (9,6) Reed-Solomon code. Note that each data object's parity blocks are depicted next to it (in gray color). Additionally, a simple XOR parity check is computed over each column's blocks and thereby a new row is added at the bottom of the matrix. In this example, repairing any single failure would only require XORing 3 blocks.

It is worth noting that the price paid for CORE's repair efficiency is the extra storage overhead (the last row in Figure 1). However, as the detailed analysis and discussions in [2] show, in realistic settings this extra overhead is reasonably low and comparable with those of the existing repairable code alternatives.

This idea, along with a set of related algorithms (e.g., efficient scheduling of multiple repairs) were later implemented as the CORE storage primitive [2] and its performance was thoroughly evaluated.

B. Reed-Solomon Representations

The construction method proposed in the original Reed-Solomon code paper [7] is based on Vandermonde matrices, and is *non-systematic*, hence it is seldom used for storage systems. However, two other methods – one using a *matrix* representation and the other one using a *polynomial* representation – were later proposed to construct *systematic* Reed-Solomon codes:

- **Cauchy Generator Matrix** [8]: this representation is based on a Cauchy matrix $\mathbf{G_{n\times k}}$ which contains the Identity matrix $\mathbf{I_{k\times k}}$. As shown below, to encode a message M of size k, it is multiplied by $\mathbf{G}$, resulting in a codeword composed of the original data message M and the parity message P of size $n-k$:

$$\begin{pmatrix} 1 & \cdot & \cdot & 0 \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ 0 & \cdot & \cdot & 1 \\ g_{0,0} & \cdot & \cdot & g_{0,k-1} \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ g_{n-k-1,0} & \cdot & \cdot & g_{n-k-1,k-1} \end{pmatrix} \times \begin{pmatrix} m_0 \\ \cdot \\ \cdot \\ m_{k-1} \end{pmatrix} = \begin{pmatrix} m_0 \\ \cdot \\ \cdot \\ m_{k-1} \\ p_0 \\ \cdot \\ \cdot \\ p_{n-k-1} \end{pmatrix}$$

  For a detailed example of the process see [9].

- **Generator Polynomial** [10]: This approach uses a generator polynomial, $g(x)$ which consists of $n-k+1$ factors and its roots are consecutive elements of the Galois Field:

$$g(x) = \sum_{i=0}^{n-k} g_i x^i$$

  Moreover, the message elements are also represented as coefficients of a polynomial, $m(x)$. To encode $m(x)$, it is first multiplied by x^{n-k} and then the result is divided by the generator polynomial, $g(x)$. The coefficients of the remainder polynomial $r(x)$ are the output parity elements:

$$m(x) \times x^{n-k} \equiv r(x) \mod g(x) \quad (1)$$

  or

$$\sum_{i=0}^{k-1} m_i x^{i+n-k} \equiv \sum_{i=0}^{n-k-1} r_i x^i \bmod \sum_{i=0}^{n-k} g_i x^i \quad (2)$$

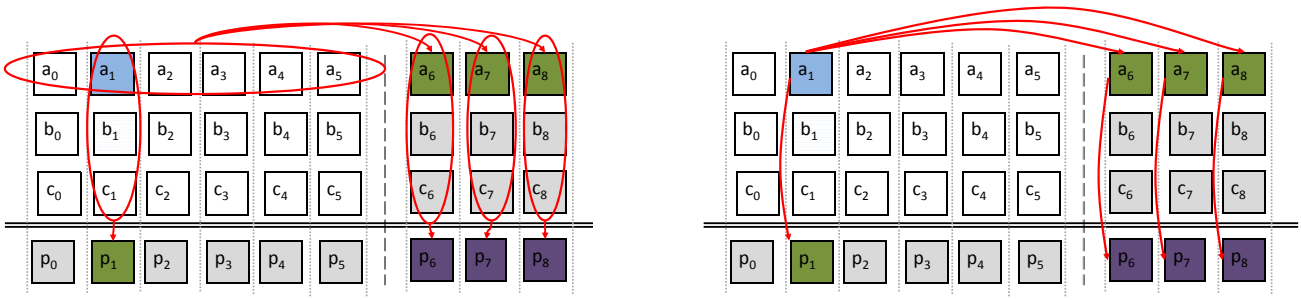
  A set of numerical example of the process can be found in [11].

Except for few recent instances (including HDFS-RAID [6] which the CORE implementation is built upon), the storage community has been traditionally using the generator matrix representation. On the other hand, as noted in [10], in the error control literature the generator polynomial representation is more commonly used.

C. Related Work

Research on updates in erasure coded storage systems has been centered around size-preserving[1] block updates [12], [5], [13], [4], [14], [15]. The naive approach to handle updates in erasure-coded storage system is to re-encode the data blocks and generate new parity blocks from scratch. This approach is clearly quite expensive, especially when the update size (number of updated blocks) is small. Hence, improving the efficiency of update handling in erasure-coded storage systems is crucial. The need for efficient update solutions in even more crucial in storage systems like CORE that maintain extra redundancies as well. Apart from few recent proposals [14], [15] that aim at designing new and inherently update-efficient erasure codes, the majority of previous work [12], [5], [13], [4] on update in erasure-coded storage systems use the classic

[1] Size-increasing and size-decreasing updates can be handled through adding new blocks and zero-padding of the updated blocks, respectively.



(a) Re-encoding (b) Parity Update

Fig. 2: The Two Update Approaches in CORE

Reed-Solomon codes and are primarily concerned with the concurrency/consistency tradeoff.

There are, nonetheless, a number of papers [3], [4], [5] in which the idea of efficient update of erasure coded data has been discussed. However, these existing solutions are not directly applicable to the CORE system for the following reasons: (i) they all target the *generator matrix* representation of RS code (see [3] for detailed explanations and formulas) and not the *generator polynomial* representation which is used in HDFS-RAID and consequently in our CORE implementation. In fact, to the best of our knowledge, designing parity update schemes for the generator polynomial representation of RS codes remains an open problem, and (ii) CORE has an extra set of vertical, XOR-based parities.

In this paper, we address these issues and analyze the effectiveness our solution through analytical and empirical studies.

III. Update Handling in CORE

In the following, we first give an overview of both re-encoding and parity update approaches in CORE (III-A). Then, after presenting the formal foundations of parity update approach for horizontal (III-B) and vertical (III-C) parities, we analytically compare both approaches (III-D).

A. Overview

Figure 2 highlights the differences between the two update approaches while handling one single update (a_1) in CORE*(n=9,k=6,t=3)*. Here are the set of steps taken in each case:

- **re-encoding**: (i) fetch $a_0...a_5$, the k blocks of the updated object, and encode them to generate $a_6..a_8$, the new $n-k$ parities, (ii) fetch a_1, b_1, c_1, the t data blocks on the updated column and XOR them to generate p_1, the new vertical parity, (iii) similarly, generate $p_6..p_8$, the other $n-k$ vertical parities.
- **parity update**: (i) first calculate Δa_1, the diff-block of the updated data block, and use it to generate $\Delta a_6...\Delta a_8$, the diff-blocks of the horizontal parities, (ii) combine $\Delta a_6...\Delta a_8$ and the old versions of $a_6..a_8$ to compute their new versions, (iii) use Δa_1 and the old version of p_1 to compute its new version, (iv) likewise, update $p_6...p_8$, the $n-k$ vertical parities.

B. Updating Horizontal Parities

Let's assume that the updated message is represented by $m^{'}(x)$, meaning:

$$m^{'}(x) \times x^{n-k} \equiv r^{'}(x) \mod g(x) \tag{3}$$

without the loss of generality, let's also assume that only the first data block of the message $m(x)$ has been updated. More precisely:

$$m_0 \neq m_0^{'} \quad ; \quad \forall i \neq 0 : m_i = m_i^{'}$$

now, by summing up equation (1) and equation (3), we obtain:

$$(m(x) + m^{'}(x)) \times x^{n-k} \equiv (r(x) + r^{'}(x)) \mod g(x)$$

or in the Σ form[2]:

$$\sum_{i=0}^{k-1}(\Delta m_i)x^{i+n-k} \equiv \sum_{i=0}^{n-k-1}(\Delta r_i)x^i \bmod \sum_{i=0}^{n-k} g_i x^i$$

but since

$$\forall i \neq 0 : m_i = m_i^{'} \Longrightarrow \forall i \neq 0 : \Delta m_i = 0$$

then

$$(\Delta m_0)x^{n-k} \equiv \sum_{i=0}^{n-k-1}(\Delta r_i)x^i \bmod \sum_{i=0}^{n-k} g_i x^i \tag{4}$$

In other words, the parity update mechanism when only the first data block (m_0) has been updated is as follows: (i) compute Δm_0, (ii) **encode** Δm_0; the outputs are $\{\Delta r_i\}, 0 \leq i \leq n-k-1$, (iii) for each i, use Δr_i and the old version of r_i to generate its new version ($\Delta r_i + r_i = r'_i$).

Note that the formula in 4 can easily be generalized:

$$\sum_{i:m_i \neq m'_i}(\Delta m_i)x^{i+n-k} \equiv \sum_{i=0}^{n-k-1}(\Delta r_i)x^i \bmod \sum_{i=0}^{n-k} g_i x^i \tag{5}$$

C. Updating Vertical Parities

Due to the simple and commutative nature of the XOR operation, updating the vertical parities of the CORE matrix is straightforward. If p is the vertical parity of a given column of t blocks:

$$\sum_{j=0}^{t-1} m_j = p$$

[2]Notice that in the Galois Field arithmetic $m_0 + m'_0 = m_0 - m'_0 = \Delta m_0$.

then it can be immediately inferred that:

$$\sum_{j=0}^{t-1}(\Delta m_j) + p = p^{'} \quad (6)$$

In other words, the new version of the parity block can be obtained by XORing its old version with the diff-blocks of the updated data blocks.

D. Analytical Comparison

Next, we present a set of cost functions that reflect the analytical cost of using the re-encoding and the parity update approaches. For the sake of simplicity, we limit our study to centralized update managers. Moreover, to measure the update cost, we consider the statically-computable *read traffic*, i.e., the amount of data read by the update manager during the process (in the centralized case, the write traffics of both approaches are identical).

	Re-encoding	Parity Update
Single Column	t	$2.u + 1$
Single Row	k	$2.u + (n-k)$
Full Matrix*	$k + u.t + (n-k).t$	$2.u + (n-k) + u + (n-k)$
Full Matrix**	$u.k + t + (n-k).t$	$2.u + u.(n-k) + 1 + (n-k)$

TABLE I: Amount of Data to Fetch to Perform u Updates

Table I shows the generalized cost functions of both approaches for XOR parities (**Single Column**), the RS codes (**Single Row**), and the complete CORE matrix (**Full Matrix**). In this table u denotes the update size. Furthermore, since in the full matrix, there are many possible distributions for u updates, we have considered only two special cases: all in one row (denoted by *) and all in one column (denoted by **). Finally, it's worth noting that the coefficient 2 in the parity update cost functions represent the fact that to compute each diff-block, both the old and the new versions of that particular block must be read.

Let's again consider the example given in Figure 2, where in the CORE matrix of size *(n=9,k=6,t=3)* one data block has been updated *(u=1)*. In this example, the overall cost (read traffic) of the re-encoding and the parity update approaches are 18 and 9 blocks respectively.

One important conclusion from these cost functions is that the benefits of parity update approach are more visible in case of small updates. In fact, as shown in [3] and confirmed later in our experiments, for a given object, if the number of updated blocks is high (e.g., more than half of its blocks), the naive alternative (re-encoding from scratch) can be not only competitive but even preferred since the costs are amortized.

IV. Implementation

The update approaches explained in Section III, have been implemented and integrated into the CORE storage primitive [16], which itself has been developed on top of Facebook's HDFS-RAID [6]. HDFS-RAID embeds the Apache HDFS [17] inside an erasure code-supporting wrapper file system named Distributed Raid File System (DRFS). DRFS supports both Reed-Solomon coding as well as simple XOR parity files. One of the main components of HDFS-RAID is RaidNode which is a daemon responsible for the creation and maintenance of parity files.

Since neither Apache HDFS nor HDFS-RAID support data updates, we first added a feature to the CORE implementation which replaces certain blocks of a file with some other pre-defined blocks. Our implementation of the re-encoding approach uses extended versions of the CORE functionalities to recompute the horizontal and vertical parities of the affected rows and columns. The parity update approach first deals with the horizontal parities that are computed with Reed-Solomon codes (according to equation 5) and then with the vertical parities that are computed using XOR (according to equation 6). In both cases, all the computations are performed in a centralized fashion at RaidNode.

The correctness of our implementation was verified through multiple test cases in which the MD5 hash values of the updated parity files generated by the re-encoding and the parity update approaches were compared against each other. The source codes, binary distribution, documentations, and a visualized demo based on the actual implementation functions are available at http://sands.sce.ntu.edu.sg/StorageCORE.

V. Experiments

We benchmarked the implementation with experiments run on a cluster of 20 nodes which has one powerful PC (4×3.2GHz Xeon Processors with 4GB of RAM) hosting the NameNode/RaidNode and 19 HP t5745 ThinClients acting as DataNodes (each with an Intel Atom N280 Processor 1.66 GHz with 1 GB of RAM and a solid state drive of 2 GB). The average bandwidth of this cluster is 12MB/s.

We ran two sets of experiments. In the first set we compare the performance of the re-encoding versus parity update schemes for one block update in the context of one single object (row) as well as the full CORE matrix. In the second set we study the impact of update size parameter and identify the sweet spots of each of the update approaches. In both sets, we used two sets of CORE matrix parameters: *(n=9,k=6,t=3)* and *(n=14,k=12,t=5)*, inspired by the code length and storage overheads of Google's GFS and Microsoft Azure, respectively. Moreover, the block size used was 64MB (HDFS's default value).

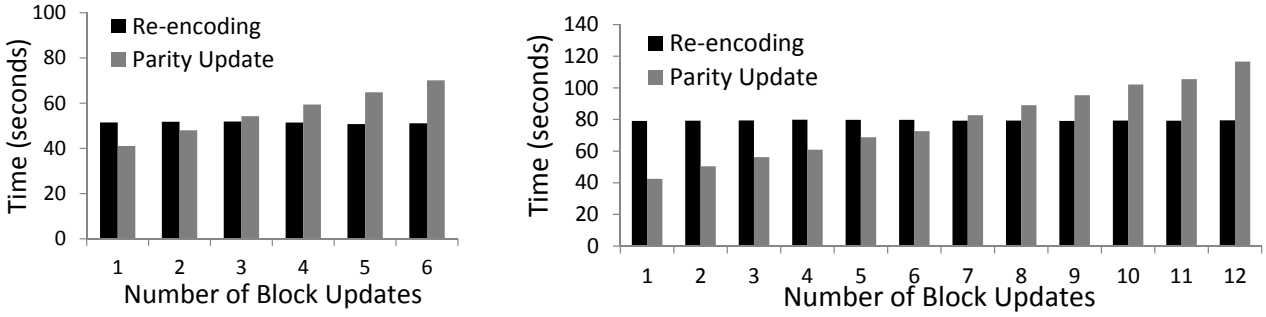
Finally, although in each experiment we measured both **completion time** and **data traffic**, but due to space limitation, we only present the time results (the data results are in line with the cost functions of Section III-D).

A. Efficiency of the Parity Update Approach

We first compare the efficiency of the update schemes for one block update (*u=1*). Figure 3 shows the completion time of a single update at the level of a row (part 3a) and the whole CORE matrix (part 3b). In accordance with the cost functions of Table I, the results show a significant gain in the update time, specially in the CORE(14,12,5).

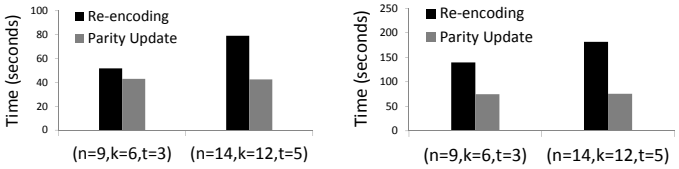
B. Impact of Update Size

As highlighted in Section III-D, for large number of block updates, the naive data re-encoding approach can be competitive. To obtain some insights about the crossover point



(a) In RS*(9,6)* (b) In RS*(14,12)*

Fig. 4: Impact of Update Size on the Performance of the Update Approaches in the Polynomial Representation of RS Codes



(a) Within a Row (b) Within the Full Matrix

Fig. 3: Completion Time of Handling One Block Update in CORE

between the two approaches, we ran an experiment in which the update size parameter u was varied between 1 (minimum) and k (maximum, the full object size).

A subset of the results (only for the RS parities) are depicted in Figure 4 and show that when around more than half of the blocks of an object are updated ($u \geq k/2$) the naive re-encoding approach can outperform the parity update approach. These results are similar to those reported in [3], where the authors measured the impact of update size in the matrix representation of RS codes.

Based on this observation, we ultimately incorporate in our CORE implementation a hybrid approach that can adaptively decide between the two schemes based on the values of u and k. Such approach can offer the best update performance under all circumstances.

VI. Conclusions and Future Work

In this paper we addressed the problem of efficient updates in the CORE storage primitive, a cross-object erasure-coded storage system. We defined and implemented a parity update scheme which outperforms the naive approach of data re-encoding. The benefits of our solution are especially pronounced in cases where the number of updated blocks is small.

Our current implementation of the update manager has a centralized design. In future, we plan to further optimize the update process by exploiting the computational resources of the storage nodes, and in doing so, reduce the update traffic over network and further lower the update time.

References

[1] A. Datta and F. Oggier, "Redundantly Grouped Cross-object Coding for Repairable Storage," in *Proceedings of the Asia-Pacific Workshop on Systems*, 2012, p. 2.

[2] K. S. Esmaili, L. Pamies-Juarez, and A. Datta, "The CORE Storage Primitive: Cross-Object Redundancy for Efficient Data Repair & Access in Erasure Coded Storage," *CoRR*, vol. abs/1302.5192, 2013.

[3] F. Zhang, J. Huang, and C. Xie, "Two Efficient Partial-Updating Schemes for Erasure-Coded Storage Clusters," in *Proceedings of the 2012 IEEE Seventh International Conference on Networking, Architecture, and Storage*, ser. NAS '12, 2012, pp. 21–30.

[4] K. Peter and A. Reinefeld, "Consistency and Fault Tolerance for Erasure-coded Distributed Storage Systems," in *Proceedings of the fifth international workshop on Data-Intensive Distributed Computing Date*, 2012, pp. 23–32.

[5] M. K. Aguilera, R. Janakiraman, and L. Xu, "Using Erasure Codes Efficiently for Storage in a Distributed System," in *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*. IEEE, 2005, pp. 336–345.

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

[7] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the Society for Industrial & Applied Mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[8] J. Bloemer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman, "An XOR-Based Erasure-Resilient Coding Scheme," International Computer Science Institute, Tech. Rep. TR-95-048, 1995.

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

[10] S. B. Wicker and V. K. Bhargava, *Reed-Solomon Codes and Their Applications*. Wiley-IEEE Press, 1999.

[11] U. of New Brunswick, "Ee4253 digital communications," http://www.ee.unb.ca/cgi-bin/tervo/rscodes.pl, Last retrieved 2013.

[12] S. Frolund, A. Merchant, Y. Saito, S. Spence, and A. Veitch, "A Decentralized Algorithm for Erasure-coded Virtual Disks," in *Dependable Systems and Networks, 2004 International Conference on*, 2004, pp. 125–134.

[13] P. Sobe, "A Partial-Distribution-Fault-Aware Protocol for Consistent Updates in Distributed Storage Systems," in *Storage Network Architecture and Parallel I/Os, 2008. SNAPI '08. Fifth IEEE International Workshop on*, 2008, pp. 54–61.

[14] A. Rawat, S. Vishwanath, A. Bhowmick, and E. Soljanin, "Update Efficient Codes for Distributed Storage," in *Information Theory Proceedings (ISIT), 2011 IEEE International Symposium on*, 2011, pp. 1457–1461.

[15] Y. Han, H.-T. Pai, R. Zheng, and P. K. Varshney, "Update-Efficient Error-Correcting Regenerating Codes," *arXiv preprint arXiv:1301.4620*, 2013.

[16] CORE, http://sands.sce.ntu.edu.sg/StorageCORE.

[17] HDFS, http://hadoop.apache.org/hdfs/.