TriEC: Tripartite Graph Based Erasure Coding NIC Offload

Haiyang Shi and Xiaoyi Lu

Department of Computer Science and Engineering, The Ohio State University
{shi.876,lu.932}@osu.edu

ABSTRACT

Erasure Coding (EC) NIC offload is a promising technology for designing next-generation distributed storage systems. However, this paper has identified three major limitations of current-generation EC NIC offload schemes on modern SmartNICs. Thus, this paper proposes a new EC NIC offload paradigm based on the tripartite graph model, namely *TriEC*. *TriEC* supports both encode-and-send and receive-and-decode operations efficiently. Through theorem-based proofs, co-designs with memcached (i.e., TriEC-Cache), and extensive experiments, we show that *TriEC* is correct and can deliver better performance than the state-of-the-art EC NIC offload schemes (i.e., *BiEC*). Benchmark evaluations demonstrate that *TriEC* outperforms *BiEC* by up to 1.82x and 2.33x for encoding and recovering, respectively. With extended YCSB workloads, *TriEC* reduces the average write latency by up to 23.2% and the recovery time by up to 37.8%. *TriEC* outperforms *BiEC* by 1.32x for a full-node recovery with 8 million records.

KEYWORDS

Erasure Coding, Bipartite, Tripartite, NIC Offload

ACM Reference Format:
Haiyang Shi and Xiaoyi Lu. 2019. TriEC: Tripartite Graph Based Erasure Coding NIC Offload . In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '19), November 17–22, 2019, Denver, CO, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3295500.3356178

1 INTRODUCTION

Erasure Coding (EC), a promising alternative of replication, has been brought into many popular distributed storage systems, such as HDFS 3.x [14], Ceph [2], QFS [30], Google Colossus [12], Facebook f4 [27], Baidu Atlas [20], and Backblaze [1]. Compared with replication, EC is capable of utilizing storage space more efficiently while providing equal or more data reliability and durability [45].

Prevalent erasure codes such as *Reed-Solomon (RS)* codes [36] are *Maximum Distance Separable (MDS)*. An MDS code (e.g., $RS(k, m)$), generating m parity chunks from k original data chunks, is able to recover up to m corrupt or lost chunks from any k of $(k+m)$ chunks. The EC-based resiliency has a storage overhead of m/k to tolerate up to m chunk corruptions. By contrast, to guarantee the same data reliability for a single data chunk, replication needs to store $m+1$ replicas, and thus the storage overhead is as high as m. However, the crucial challenge of employing EC in distributed storage systems is how to efficiently alleviate the encoding overhead to generate the parity chunks and the decoding cost to reconstruct the original data chunks for failures.

To alleviate the EC overhead, advanced EC schemes have been proposed in the community. Broadly, these EC schemes can be divided into two categories – On-CPU EC and Off-CPU EC. Intel ISA-L [18] and Jerasure [31] are quite popular On-CPU EC libraries for storage systems. On-CPU EC has good performance but it consumes more CPU cycles for finishing EC tasks, which could incur additional overhead due to possible CPU contentions with applications. On the other hand, Off-CPU EC is an emerging and promising feature, which provides the capabilities of performing EC tasks on advanced hardware devices, such as GPGPUs [6], FPGAs [35], and smart network interface cards (SmartNICs). For instance, the host channel adapters (HCA) of Mellanox ConnectX-4/5 and later SmartNICs offer an EC calculation engine [25], which allows applications to offload EC tasks on the NIC. Compared with On-CPU EC, Off-CPU EC is a new design methodology which brings computation offloading capabilities (low CPU utilization) and more overlapping opportunities to the applications.

1.1 Motivation

Among different kinds of Off-CPU EC schemes, the EC NIC offload technology is an exciting direction, because it provides the possibility of offloading both EC calculations and data transmissions to the NIC. More specifically, there are two kinds of schemes to take advantage of the EC offload capability on SmartNICs (e.g., Mellanox InfiniBand): (1) *Incoherent EC Calculation and Networking*, and (2) *Coherent EC Calculation and Networking* [25].

The incoherent EC calculation and networking scheme means the storage systems have to take care of EC calculations and data transmissions separately. Taking EC encoding for example, an initiator node first encodes data chunks on the NIC and then sends data chunks and generated parity chunks to other receiver nodes (termed as "encode-then-send"). Figure 1a describes the detailed steps of the encode-then-send scheme. As we can see here, even though EC computations are offloaded to the NIC in this scheme, the CPU still needs to be involved in multiple steps, such as posting send operations (i.e., *post_send*) for original data chunks, posting the EC operation, and then posting send operations again for the generated parity chunks. Not only for higher CPU involvements, but there are also many Direct Memory Access (DMA) operations happening in the encode-then-send workflow. We believe this is not the optimal way of utilizing the EC NIC offload technology on SmartNICs.

This research is supported in part by National Science Foundation grant #CCF-1822987.

SC '19, November 17–22, 2019, Denver, CO, USA

ACM ISBN 978-1-4503-6229-0/19/11…$15.00
https://doi.org/10.1145/3295500.3356178

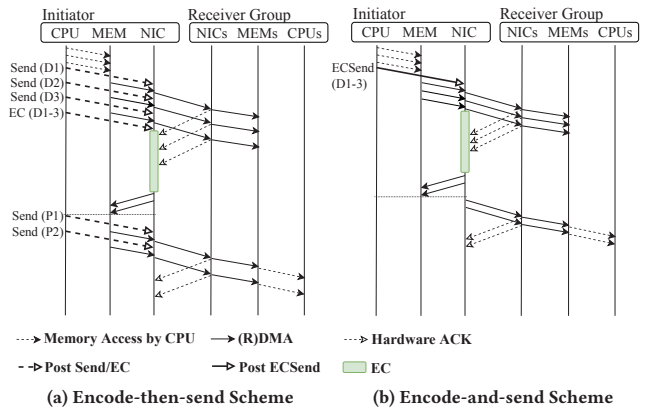


Figure 1: Overview of Existing EC NIC Offload Schemes

On the other hand, with the coherent EC calculation and networking scheme, the storage systems can offload both EC calculations and data transmissions simultaneously to the NIC. As shown in Figure 1b, the initiator node only needs to post one network operation (i.e., *post_encode_and_send*), which can fully offload the EC computation and communication stages to the NIC (termed as "encode-and-send"). In contrast to the encode-then-send scheme, the encode-and-send scheme has the potential to deliver higher performance, because it ideally could save several (or a few) *post_send* and DMA operations in the pipeline. Thus, in this paper, we take the encode-and-send scheme (i.e., coherent EC calculation and networking scheme) as the start-of-the-art baseline for our study.

While the coherent EC calculation and networking scheme on modern SmartNICs is very powerful as discussed above, we have identified three major limitations with its current-generation designs.

Limitation 1: We find that current-generation encode-and-send scheme follows a **Bipartite graph based EC** encoding paradigm. We call this existing encoding paradigm as *BiEC* in this paper, which means the EC encoding process is decomposed into two stages running on two sets of nodes, such that one set of nodes (e.g., clients) just perform EC encodings and the other set of nodes (e.g., data nodes) just receive encoded chunks. *BiEC* inherits the traditional On-CPU EC methodology (e.g., EC schemes in [1, 2, 12, 14, 27, 30]) to simply use NICs as processors or accelerators, which can not fully exploit networked computing powers to achieve high parallelism and overlapping. (See Figure 3a in Section 3)

Limitation 2: Current-generation EC NIC offload only supports to offload the encode-and-send primitive, but it is lack of the support for "receive-and-decode" primitive on the NIC, which should be the other essential offload primitive (i.e., *post_recv_and_decode*). Because of the missing of the "receive-and-decode" primitive, distributed storage systems can only first receive all the required chunks and then perform the decode operation on them with current EC NIC offload design. This is again a bipartite graph-based decoding, which can not achieve enough parallelism and overlapping. (See Figure 5a in Section 3)

Limitation 3: The semantics of decoding with current-generation EC NIC offload can only guarantee the readers will get the correct data, but it can not guarantee that the unhealthy nodes will automatically recover the lost or corrupt data. Thus, applications typically need to design an out-of-band recovery mechanism to reconstruct the lost or corrupt data. This is a semantic mismatch between application requirements and primitives provided by the state-of-the-art EC NIC offload mechanism. (See Section 5)

1.2 Contribution

To address the identified limitations as discussed above, in this paper, we first propose a new EC NIC offload paradigm based on the tripartite graph model, called **Tripartite graph based EC NIC Offload** or ***TriEC***. The idea of *TriEC* comes from the insightful observations on the EC computation process and how it can be executed on networked computing resources in parallel. *TriEC* decomposes a full EC calculation pipeline into three stages, and each stage only handles a subset of EC tasks. From the graph perspective, these concurrent sub-EC tasks on three sets of nodes construct a tripartite graph. *TriEC* enables these sub-EC tasks to be executed on different nodes in parallel. Thus, *TriEC* can gain more parallelism and overlapping compared with *BiEC* for both encoding and decoding data. To prove the correctness and efficiency of *TriEC*, we systematically model both *BiEC* and *TriEC*, and then we prove several important theorems and corollaries for *TriEC* (See Section 4).

Secondly, we propose a new receive-and-decode primitive on top of Mellanox EC NIC offload APIs. The receive-and-decode primitive is also designed by the guidance of *TriEC* principles. It is a significant complement to the state-of-the-art EC NIC offload APIs and enables upper-layer applications to leverage EC NIC offload APIs for data reconstructions fully.

Thirdly, to show the benefits of *TriEC*, we co-design a new key-value store based on `memcached` with *TriEC*, called TriEC-Cache. With TriEC-Cache, this paper further presents how *TriEC* can help distributed storage systems to achieve efficient in-band data recovery through the proposed receive-and-decode primitive. We also discuss several optimization techniques (e.g., avoidance of sending unrequested chunks and EC calculator cache) to fully take advantage of the performance potential of Mellanox EC NIC offload capabilities. These optimizations could be integrated into next-generation Mellanox EC NIC offload APIs or inspire the researchers and engineers to refine the EC NIC offload APIs in the future.

Benchmark evaluations demonstrate that *TriEC* outperforms *BiEC* by up to 1.82x and 2.33x for encoding and recovering, respectively. To further understand the performance benefits of our proposed designs in TriEC-Cache with real workloads, we extend the existing Yahoo! Cloud Serving Benchmark (YCSB) [5] application-level workloads to support a new operation (i.e., *read_with_erasures*) for evaluating the performance impact of EC recoveries. We also extend YCSB to guarantee the occurrence of *read_with_erasures* within normal reads follows the Weibull distribution, which is the typical failure distribution for real-world HPC and data center systems [11, 15, 24, 38].

The performance evaluations on up to 25 nodes with the extended YCSB reveal that *TriEC* designs can achieve lower latency and higher throughput, compared with the baseline implemented

with *BiEC*. Specifically, with only 1% failure occurrences, *TriEC* reduces the average write latency by up to 23.2% and 12.0% for equal-shares and read-mostly workloads, respectively, and cuts down the average latency of *read_with_erasure* by up to 37.8%, 37.5%, and 36.1% for equal-shares, read-mostly, and read-only workloads, respectively. On the other hand, *TriEC* improves the overall system throughput by up to 13.3% for equal-shares, 14.8% for read-mostly, and 13.9% for read-only workloads. Moreover, *TriEC* outperforms *BiEC* by 1.32x for a full-node recovery with 8 million records.

In summary, this paper makes the following key contributions:

(1) We propose a novel tripartite graph based EC NIC offload paradigm (i.e., *TriEC*) and its associated efficient designs for both encoding and decoding;
(2) We prove that *TriEC* is correct and has the potential to deliver better performance than current-generation *BiEC* based NIC offload schemes;
(3) Through co-designed TriEC-Cache system and extensive evaluations, we verify the theorems of *TriEC* and demonstrate its benefits for real-world workloads.

To the best of our knowledge, this is the first work to propose a tripartite graph based EC paradigm (i.e., *TriEC*) and demonstrate the efficiency of *TriEC* on SmartNIC's EC offload schemes.

2 ERASURE CODING BASICS

Reed-Solomon (RS) codes are the most widely-used erasure codes and have been employed in many storage systems [1, 2, 12, 14, 27, 30]. For an RS code (e.g., $RS(k, m)$), it encodes on k original data chunks to generate k data chunks and m parity chunks. As illustrated in Figure 2a, the k original data chunks are identical to the generated k data chunks. This property is named as *systematic*, which enables applications to read data directly from encoded data if no corruptions occur. RS codes are *Maximum Distance Separable (MDS)*. An MDS code (e.g., $RS(k, m)$) is able to recover up to m corrupt or lost chunks from any k of the generated $(k + m)$ chunks.

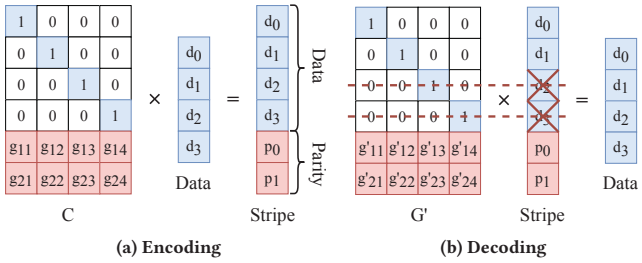


Figure 2: Reed-Solomon Coding for $k = 4$ and $m = 2$

In a typical erasure-coded storage system, it divides data into *blocks* and applies EC independently on a per-block basis. For an RS code (e.g., $RS(k, m)$), a block is split into k equal-size uncoded chunks (i.e., data chunks), and m additional parity chunks are calculated from the k data chunks. As illustrated in Figure 2a, the calculation involved in $RS(4, 2)$'s encoding is $\boldsymbol{C} \times [d_0, d_1, d_2, d_3]^\intercal = [d_0, d_1, d_2, d_3, p_0, p_1]^\intercal$. The collection of $k+m$ chunks forms a *stripe*. Each stripe is an independent entity for EC, which allows the design of distributed storage systems to be flexible. A common way to persistently store stripes is to store the $k + m$ chunks within a stripe on $k + m$ separate nodes, since this arrangement allows recovery from any combination of up to m simultaneous corruptions or failures, which achieves the theoretically best for MDS codes. To recover corrupt or missing data chunks in a stripe, the system first collects any k survived chunks in the same stripe and then reconstructs unhealthy chunks by computing the decoding calculation as shown in Figure 2b. For example, to recover d_2 and d_3, the system constructs decoding matrix $\boldsymbol{G'}$ by choosing four rows corresponding to the remaining four healthy chunks (i.e., d_0, d_1, p_0, and p_1) and taking the inverse of it, and then multiplies $\boldsymbol{G'}$ with the four survived chunks to reconstruct the four original data chunks (i.e., d_0 to d_3).

3 TriEC DESIGN

This section presents our proposed *TriEC* paradigm and compares it with the default *BiEC* paradigm. To help describe the EC process precisely, we use the following symbols in this section, including: n: number of iterations; k: number of data chunks; m: number of parity chunks; D: chunk size; N_y^x: Node y in layer x; T_{EC}^l: execution time of erasure coding operation on l chunks; T_{enc}^l: execution time for erasure encoding l chunks; T_{dec}^l: execution time for erasure decoding l chunks; T_{comm}: latency for sending chunks to remote peers; and T_{XOR}^l: execution time for XORing l chunks. Note that we omit the superscripts in the aforementioned symbols if $l = 1$, e.g., T_{enc} denotes the execution time for encoding one chunk with D bytes.

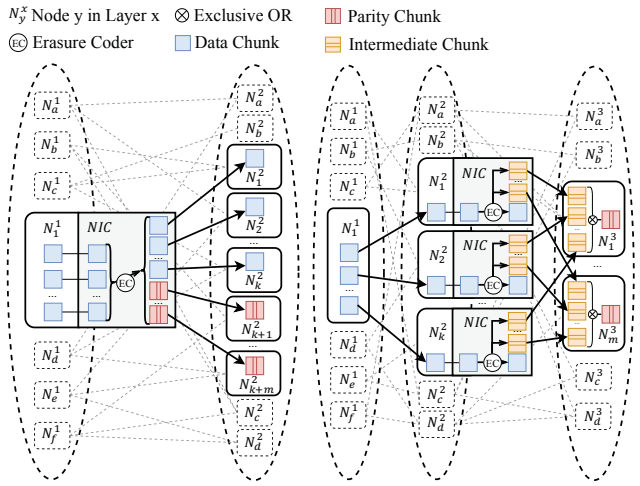
3.1 Encoding Paradigm for NIC Offload

A typical encoding procedure of *BiEC* is depicted in Figure 3a. In each iteration, when there are k data chunks available, the initiator (i.e., N_1^1), which is going to write chunks to remote peers, offloads encoding calculation and networking (i.e., *encode-and-send*) to its NIC. The NIC first sends out the k data chunks and then encodes on the available k data chunks to generate m parity chunks. Once completing performing the encoding calculation, the NIC sends these parity chunks to different remote peers as previously determined by N_1^1. It is evident that N_1^1 is highly possible to be a bottleneck of the entire system, since N_1^1 carries out both computation and data transmission while other nodes only receive data packets and wait. If taking multiple iterations into account, we observe that there is no overlapping between iterations (as shown in Figure 4a). The reason behind our observation is that, to guarantee data reliability and availability, N_1^1 cannot move forward until all remote peers acknowledge that all chunks are safely stored. This observation also means the bipartite encoding paradigm for NIC offload is lack of enough parallelism and overlapping.

Since there is no overlapping between iterations, the execution time for *BiEC* (i.e., $T_{\text{bi-enc}}$) of n iterations is modeled as Equation 1.

$$\begin{aligned} T_{\text{bi-enc}} &= n \cdot \max\left\{T_{\text{enc}}^k + T_{\text{comm}}, T_{\text{comm}}\right\} \\ &= n \cdot (T_{\text{enc}}^k + T_{\text{comm}}) \end{aligned} \tag{1}$$

For simplicity, we assume the time costs (i.e., T_{comm}) of communicating data chunks and parity chunks are identical in our models, since the chunk sizes of data chunks and parity chunks are the same and these chunks are all transmitted in parallel. High-performance



(a) Bipartite Paradigm (Default) (b) Tripartite Paradigm (Proposed)

Figure 3: Overview of Two Erasure Encoding Paradigms for NIC Offload

interconnects typically have enough bandwidth to transmit these chunks efficiently. Thus, we believe this is a reasonable assumption for modern high-performance SmartNICs.

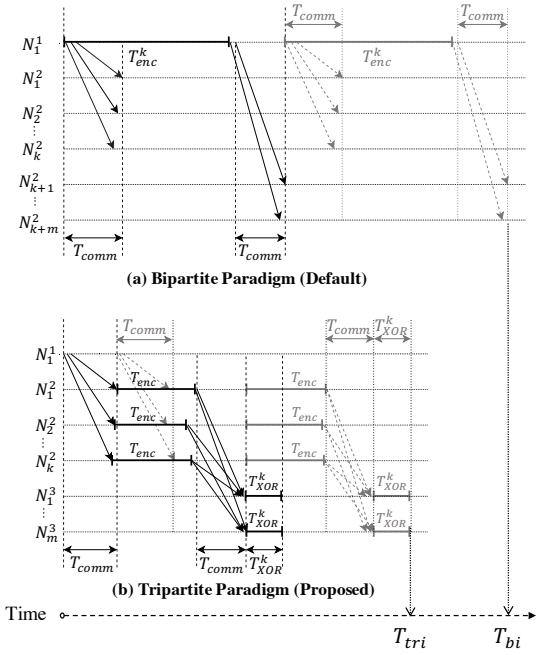


Figure 4: Activities of Two Erasure Encoding Paradigms for NIC Offload over Time

By contrast, our proposed *TriEC* makes it possible to conduct EC calculations and networking in parallel and to overlap between iterations. Figure 3b illustrates a typical workflow of *TriEC*. After collecting k data chunks, the initiator (N_1^1) distributes them onto k data peers (i.e., N_1^2 to N_k^2). After the k data peers indicate that all k data chunks are safely stored, N_1^1 steps to next iteration. Meanwhile, each NIC of N_1^2 to N_k^2 performs a relatively smaller data encoding calculation, which outputs m intermediate chunks. Each node of the m parity peers (i.e., N_1^3 to N_m^3) receives one intermediate chunk from each of the k data peers (N_1^2 to N_k^2), and finally gets the wanted parity chunk by XORing k intermediate chunks (see Section 4 for proof and more details). Since a large EC calculation is decomposed into several smaller ones being computed by multiple nodes in parallel, our proposed *TriEC* is expected to achieve more efficient resource utilization and higher performance. Moreover, the nodes in each layer can move forward to the next iteration as long as the chunks sent by them are acknowledged as stored. Therefore, the overlap between iterations happens and *TriEC* delivers more performance benefits.

Figure 4b briefly depicts the pipeline of *TriEC*, which shows how activities of different layers overlap in *TriEC*. In the three layers of *TriEC*, layer-1 carries out communication (T_{comm}), layer-2 performs EC and communication ($T_{\text{enc}} + T_{\text{comm}}$), and layer-3 finally XORs received intermediate chunks to generate parity chunks (T_{XOR}^k). Though the activities of each layer must be performed iteration by iteration, the activities between any two adjacent layers can be overlapped. Assume there are n iterations, after the first iteration, each of the other $n-1$ iterations could be completed within $\max\left\{T_{\text{comm}}, T_{\text{enc}} + T_{\text{comm}}, T_{\text{XOR}}^k\right\}$ since the activities of the three layers are overlapped. Thus, Equation 2 describes the execution time ($T_{\text{tri-enc}}$) of *TriEC* encoding with overlapping.

$$\begin{aligned} T_{\text{tri-enc}} &= 2 \cdot T_{\text{comm}} + T_{\text{enc}} + T_{\text{XOR}}^k \\ &\quad + (n-1) \cdot \max\left\{T_{\text{comm}}, T_{\text{enc}} + T_{\text{comm}}, T_{\text{XOR}}^k\right\} \\ &= 2 \cdot T_{\text{comm}} + T_{\text{enc}} + T_{\text{XOR}}^k \\ &\quad + (n-1) \cdot \max\left\{T_{\text{enc}} + T_{\text{comm}}, T_{\text{XOR}}^k\right\} \end{aligned} \tag{2}$$

BiEC and *TriEC* have different fault tolerance mechanisms. For *BiEC*, if an iteration begins, the stripe involved in the previous iteration is persistent; thus, the k data chunks are able to tolerate up to m failures. By contrast, for *TriEC*, the previous stripe is able to tolerate m failures after parity peers acknowledge that m parity chunks are persistent. Therefore, when designing distributed storage systems with *TriEC*, we need to take care of this difference in the I/O pipelines.

3.2 Decoding Paradigm for NIC Offload

For a typical storage system, the initiator has to go through three steps to complete the data recovery. The three steps are reading data chunks, detecting data corruptions, and reconstructing data. To enable the decoding paradigms to achieve the best overlap, we assume that the initiator has already issued enough read requests before the decoding paradigms start to process. We also omit these read requests in the activity figures to make them simpler and clearer. Since the latency to detect data corruptions is very low, we treat the corruption detecting as an event, named corruption detecting event, and it is denoted as a pink diamond in the activity figures.

For each iteration in bipartite decoding, when N_1^1 (i.e., N_1^1 in Figures 5a and 6a) detects that there are data corruptions, it fetches

several survived chunks from corresponding remote peers. When N_1^1 has k survived chunks, it then decodes on the k chunks to recover the k original data chunks. Since the initiator has to carry out corruption detecting and data reconstructing sequentially, there is no overlap between iterations. Thus, as shown in Figure 6a, we can model the execution time ($T_{\text{bi-dec}}$) of n iterations of bipartite decoding by Equation 3.

$$T_{\text{bi-dec}} = n \cdot (T_{\text{comm}} + T_{\text{dec}}^k) \tag{3}$$

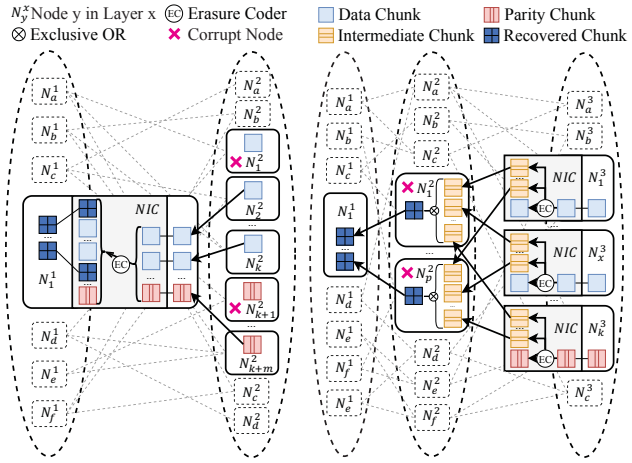


Figure 5: Overview of Two Erasure Decoding Paradigms for NIC Offload

On the other hand, at the beginning of the decoding workflow in *TriEC*, the initiator (N_1^1) talks to the remote peers (i.e., N_1^2 to N_p^2), which are responsible for storing the corrupt chunks. Each node of N_1^2 to N_p^2 detects data corruptions and fetches k intermediate chunks from k survived peers (i.e., N_1^3 to N_k^3), then XORs on the k intermediate chunks to reconstruct the corrupt chunk. After recovering the corrupt chunks, N_1^2 to N_p^2 update local chunks and send them back to N_1^1. Each node on the third layer generates p (number of corrupt chunks) intermediate chunks and sends them to nodes of N_1^2 to N_p^2 with *encode-and-send* primitive (see Section 4 for proof and more details), i.e., the *recv-and-decode* primitive proposed in this paper is implemented with *encode-and-send* primitive to provide high-programmability.

TriEC is able to offload corruption detection to the nodes in the second layer, and it can also decouple corruption detection and major data decoding processes between the second layer and the third layer in the tripartite graph, respectively. For instance, nodes N_1^2 to N_p^2 on the second layer take care of corruption detecting, and nodes N_1^3 to N_k^3 carry out the major parts of data reconstructing. Therefore, each layer in the tripartite graph structure only depends on the next adjacent layer, which enables overlap to occur within the entire graph (as shown in Figure 6b). In addition, since a large decoding computation is split into several smaller ones being computed by multiple nodes in parallel, *TriEC* gains more parallelism.

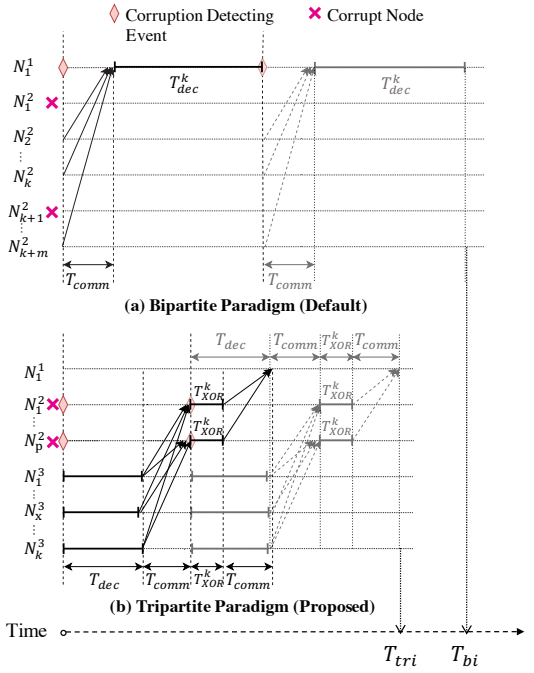


Figure 6: Activities of Two Erasure Decoding Paradigms for NIC Offload over Time

Furthermore, as illustrated in Figure 6b, the recoveries on N_1^2 to N_p^2 are in-band, i.e., the nodes involved in reconstructing corrupt chunks become healthy at the end (refer to Section 5.2 for more details). The in-band recovery does not require the initiator N_1^1 to detect or realize any data corruptions, which can lead to better performance. The execution time ($T_{\text{tri-dec}}$) of n times of *TriEC* decodes is as shown in Equation 4.

$$\begin{aligned} T_{\text{tri-dec}} &= T_{\text{dec}} + 2 \cdot T_{\text{comm}} + T_{\text{XOR}}^k \\ &\quad + (n-1) \cdot \max\left\{T_{\text{dec}} + T_{\text{comm}}, T_{\text{XOR}}^k + T_{\text{comm}}\right\} \end{aligned} \tag{4}$$

Note that, in this section, we consider the best overlapping cases for both *BiEC* and *TriEC* for comparing them fairly. *TriEC* achieves overlapping and parallelism by decomposing EC procedures into multiple stages which are able to be performed on separate nodes in parallel. Thus *TriEC* is able to deliver better performance than *BiEC*.

To summarize, our proposed *TriEC* brings more parallelism and overlapping, and we show that *TriEC* can outperform *BiEC* in both encoding and decoding data in the next section.

4 THEOREMS OF TriEC

According to *RS(k,m)* encoding scheme, m additional parity chunks are calculated from the k original data chunks. The encoding operation can be represented as a matrix-vector multiplication, where the vector of k data chunks is multiplied by a particular matrix $C = \begin{bmatrix} I \\ G \end{bmatrix}$. The matrix C is of size $(k+m) \times k$, where I is a $k \times k$ identity matrix and G is an $m \times k$ matrix called the generator matrix, which yields the MDS property. Let's denote the k data

chunks by $[D_1, D_2, \cdots, D_k]$, and denote the m parity chunks by $[P_1, P_2, \cdots, P_m]$. Such that, the encoding operation is represented by Equation 5.

$$\underbrace{\begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \\ \hline g_{11} & g_{12} & \cdots & g_{1k} \\ g_{21} & g_{22} & \cdots & g_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ g_{m1} & g_{m2} & \cdots & g_{mk} \end{bmatrix}}_{C} \times \begin{bmatrix} D_1 \\ D_2 \\ D_3 \\ \vdots \\ D_k \end{bmatrix} = \begin{bmatrix} D_1 \\ D_2 \\ \vdots \\ D_k \\ P_1 \\ \vdots \\ P_m \end{bmatrix} \tag{5}$$

In *RS(k,m)* decoding scheme, corrupt chunks are reconstructed by solving the reduced system of linear equations obtained by removing the rows corresponding to corrupt chunks in Equation 5. Each decoding operation takes two steps: (1) constructing the decoding matrix (denoted by $\boldsymbol{G'}$) by choosing k survived rows of $\boldsymbol{C}$, and then taking the inverse of the matrix comprising of the chosen k rows, and (2) multiplying the decoding matrix with the vector composed of data and parity chunks corresponding to the chosen k rows. All additions and multiplications involved in encoding and decoding are based on Galois Field arithmetic over w-bit units (termed $GF(2^w)$). Note that additions on $GF(2^w)$ are associative and equivalent to bitwise XOR.

4.1 Theorem of Equivalence

Lemma 1. *BiEC is a correct EC implementation.*

Proof. The implementation of *BiEC* strictly follows the calculation procedure (i.e., EC definition) as shown in Equation 5; thus, *BiEC* is correct obviously. ■

Theorem 1. *TriEC is equivalent to BiEC with respect to EC outputs.*

Proof. (Encode) Let P_i denote a parity chunk, where $1 \le i \le m$. According to Equation 5, we can get the P_i for *BiEC* as following:

$$P_i = \sum_{j=1}^{k} g_{ij} \cdot D_j \tag{6}$$

In *TriEC*, node s ($1 \le s \le k$) on the second layer offloads the calculation as shown in Equation 7 onto its NIC, and then sends the intermediate chunks (i.e., I_{1s} to I_{ms}) to m parity nodes on the third layer.

$$\begin{bmatrix} g_{1s} \\ g_{2s} \\ \vdots \\ g_{ms} \end{bmatrix} \times D_s = \begin{bmatrix} I_{1s} \\ I_{2s} \\ \vdots \\ I_{ms} \end{bmatrix} \tag{7}$$

Let P_t ($1 \le t \le m$) denote a parity chunk finally stored on a third-layer node. As explained in Section 3.1, the third-layer node constructing the parity chunk by XORing k intermediate chunks from k second-layer nodes. Such that,

$$P_t = \sum_{s=1}^{k} I_{ts} = \sum_{s=1}^{k} g_{ts} \cdot D_s, \tag{8}$$

which is identical to Equation 6.

(Decode) Suppose $\boldsymbol{S} = [S_1, S_2, \cdots, S_k]^\intercal$ are the k survived chunks selected to reconstruct up to m corrupt chunks, and $\boldsymbol{G'}$ in Equation 9 is the corresponding decoding matrix.

$$\boldsymbol{G'} = \begin{bmatrix} g'_{11} & g'_{12} & \cdots & g'_{1k} \\ g'_{21} & g'_{22} & \cdots & g'_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ g'_{k1} & g'_{k2} & \cdots & g'_{kk} \end{bmatrix} \tag{9}$$

Let $\boldsymbol{R} = [R_{c_1}, R_{c_2}, \cdots, R_{c_p}]$ ($\|\boldsymbol{R}\| \le m$)[1] denote the chunks to be reconstructed, where c_t ($1 \le t \le p$) is the position of the corresponding row of R_{c_t} in $\boldsymbol{G'}$.

If $\boldsymbol{R}$ is recovered with *BiEC*, then

$$R_{c_t} = \sum_{j=1}^{k} g'_{c_t j} \cdot S_j \tag{10}$$

On the other hand, with *TriEC*, as illustrated in Section 3.2, the main reconstruction is carried out by the third-layer nodes. Let I_{rs} ($R_{c_r} \in \boldsymbol{R}, 1 \le s \le k$) denote the intermediate chunk which is constructed by the node storing chunk S_s ($S_s \in \boldsymbol{S}$) and is required to reconstruct R_{c_r}. Suppose a node on the third layer which stores chunk S_s ($S_s \in \boldsymbol{S}$) is selected to recover $\boldsymbol{R}$, it offloads the calculation in Equation 11 to its NIC, and then sends the generated intermediate chunks (i.e., I_{1s} to I_{ps}) to the nodes which are reconstructing corrupt chunks.

$$\begin{bmatrix} g'_{c_1 s} \\ g'_{c_2 s} \\ \vdots \\ g'_{c_p s} \end{bmatrix} \times S_s = \begin{bmatrix} I_{1s} \\ I_{2s} \\ \vdots \\ I_{ps} \end{bmatrix} \tag{11}$$

A second-layer node which is responsible for recovering chunk R_{c_r} ($R_{c_r} \in \boldsymbol{R}$) fetches k intermediate chunks from k survived third-layer nodes and performs XOR on them to reconstruct the corrupt chunk. Thus,

$$R_{c_r} = \sum_{j=1}^{k} I_{c_r j} = \sum_{j=1}^{k} g'_{c_r j} \cdot S_j, \tag{12}$$

which is equivalent to Equation 10.

Since both encoding and decoding are identical, *TriEC* is equivalent to *BiEC*. ■

The theorem of equivalence implies the following two corollaries:

Corollary 1.1. *(Correctness) TriEC is correct.*

Proof. Since *TriEC* is equivalent to *BiEC*, and *BiEC* is correct (Lemma 1), *TriEC* is correct as well. ■

To guarantee the correctness in the implementation of *TriEC* is challenging, because of two aspects: (1) implementing the *recv-and-decode* primitive with *encode-and-send* primitive, and (2) constructing sub-matrices (i.e., $[g_{1s}, \cdots]^\intercal$ in Equation 7 and $[g'_{c_1 s}, \cdots]^\intercal$ in Equation 11) for encoding and decoding in parallel.

Corollary 1.2. *(Compatibility) TriEC is compatible with BiEC.*

[1] $RS(k, m)$ can recover at most m corrupt chunks.

Proof. *TriEC* is compatible with *BiEC* because they are identical w.r.t EC outputs. ■

Because *TriEC* is compatible with *BiEC*, the implementation of *TriEC* can be validated by comparing its results with those of *BiEC*; thus, we can guarantee the implementation correctness of *TriEC* based on Corollary 1.2. Corollary 1.2 is also an essential guideline for designing distributed storage systems with *TriEC*. It implies that *TriEC* is able to work together with *BiEC*, and thus offers design flexibility and more co-design opportunities to the upper-layer applications. For instance, an application does not have to store any metadata to differentiate chunks erasure-coded by *BiEC* or *TriEC*, because *TriEC* is able to reconstruct corrupt chunks erasure-coded by *BiEC* and vice versa.

4.2 Theorem of Performance

Theorem 2. *If* $T^k_{\text{EC}} > T_{\text{EC}} > T_{\text{comm}}$, $T_{\text{EC}} > T^k_{\text{XOR}}$, *and* $n > \epsilon$ *, then TriEC outperforms BiEC.*

Proof. (Encode) Since $T_{\text{enc}} > T^k_{\text{XOR}}$, Equation 2 is reduced to

$$T_{\text{tri-enc}} = n \cdot (T_{\text{comm}} + T_{\text{enc}}) + T_{\text{comm}} + T^k_{\text{XOR}} \tag{13}$$

Let $\Delta = T_{\text{bi-enc}} - T_{\text{tri-enc}} = n \cdot (T^k_{\text{enc}} - T_{\text{enc}}) - T_{\text{comm}} - T^k_{\text{XOR}}$. Given that $T^k_{\text{enc}} > T_{\text{enc}}$, if

$$n > \epsilon = \frac{T^k_{\text{XOR}} + T_{\text{comm}}}{T^k_{\text{enc}} - T_{\text{enc}}}, \tag{14}$$

we obtain $\Delta > 0$, i.e., *TriEC* performs better than *BiEC*.

Therefore, *TriEC* outperforms *BiEC* in encoding if $T^k_{\text{enc}} > T_{\text{enc}}$, $T_{\text{enc}} > T^k_{\text{XOR}}$, and $n > \epsilon$, where ϵ is bound by Equation 14.

(Decode) Given that $T^k_{\text{dec}} > T_{\text{dec}} > T_{\text{comm}}$, we can reduce Equation 4 to Equation 15.

$$T_{\text{tri-dec}} = n \cdot (T_{\text{dec}} + T_{\text{comm}}) + T_{\text{comm}} + T^k_{\text{XOR}} \tag{15}$$

To obtain $\Delta = T_{\text{bi-dec}} - T_{\text{tri-dec}} = n \cdot (T^k_{\text{dec}} - T_{\text{dec}}) - T_{\text{comm}} - T^k_{\text{XOR}} > 0$, where $T^k_{\text{dec}} > T_{\text{dec}}$, we have

$$n > \epsilon = \frac{T^k_{\text{XOR}} + T_{\text{comm}}}{T^k_{\text{dec}} - T_{\text{dec}}}. \tag{16}$$

Thus, for decoding, *TriEC* delivers better performance than *BiEC* if $T^k_{\text{dec}} > T_{\text{dec}} > T_{\text{comm}}$, and $n > \epsilon$, where ϵ is bound by Equation 16.

Since Equations 14 and 16 have the same form, and symbols T^k_{enc} and T^k_{dec} can be unified as T^k_{EC}, we conclude that *TriEC* outperforms *BiEC* if $T^k_{\text{EC}} > T_{\text{EC}} > T_{\text{comm}}$, $T_{\text{EC}} > T^k_{\text{XOR}}$, and

$$n > \frac{T^k_{\text{XOR}} + T_{\text{comm}}}{T^k_{\text{EC}} - T_{\text{EC}}}. \tag{17}$$

■

Corollary 2.1. *TriEC outperforms BiEC on modern HPC clusters for large data storage and analytics workloads.*

Proof. Our profiling results on modern HPC clusters show that $T^k_{\text{EC}} > T_{\text{EC}} > T_{\text{comm}}$, $T_{\text{EC}} \gg T^k_{\text{XOR}}$ (in Table 1). The lower bound of n in Table 1 derived by Equation 17 can be easily satisfied in real-world storage systems, because the reads and writes in real-world storage systems can easily incur enough number of iterations (i.e., a big n is very common for real-world storage systems). According to Theorem 2, *TriEC* can perform better than *BiEC* on modern HPC clusters for large data storage and analytics workloads. ■

Table 1: Profiling Numbers on Modern HPC Clusters. LB(n) refers to the lower bound of n derived by Equation 17. Please refer to Section 6.1 for cluster specifications.

RS(3, 2)	OSU RI2 Cluster			OSC Pitzer Cluster		
unit: us	1KB	16KB	1MB	1KB	16KB	1MB
T_{comm}	1.63	4.62	95	1.99	4.97	100.47
T_{EC}	13	43	2244	12	41	2196
T^k_{EC}	14	46	3087	13	44	3001
T^k_{XOR}	1	4	300	1	4	253
LB(n)	3	3	1	3	3	1

4.3 Discussion

As a generic EC paradigm, *TriEC* does not only work on SmartNICs, and it can also be generalized to other hardware technologies. This paper mainly focuses on applying *TriEC* on SmartNICs, because:

(1) High-performance SmartNICs has low latency (i.e., T_{comm} is small); thus, the lower bound of n derived from Equation 17 can be small even for small messages. Therefore, *TriEC* with high-performance SmartNICs is more efficient.
(2) The capability of *coherent EC calculation and networking* is a promising technique to deliver high performance to upper-layer applications.
(3) It is more challenging to fully deliver potential performance capability of SmartNICs to EC-based storage systems and applications.

It will be our future work to apply *TriEC* on other hardware devices.

5 TriEC-Cache

To understand the performance implication of *TriEC* on real-world storage systems and to show how to integrate *TriEC* into existing storage systems, we implement a key-value store, namely **TriEC-Cache**, based on `memcached` (v1.5.12).

5.1 Architecture

TriEC-Cache is designed with an interleaved architecture, which interleaves multiple EC groups into a cluster to balance workload and resource utilizations. Figure 7 depicts an architecture example for *RS(3,2)*. We refer to processes which store parity chunks as parity processes, and processes which store data chunks as data processes. As shown in Figure 7, each node in the cluster runs both parity processes and data processes. Therefore, TriEC-Cache achieves balance with respect to CPU, memory, and network utilization. Each EC group has a leader, which serves requests from clients. To achieve load balance, the leaders of multiple EC groups distribute

*The value of ϵ is bound by Equation 17.

around the cluster uniformly as well. Data processes and parity processes belong to the same EC group are assigned to different nodes to guarantee the best reliability and availability.

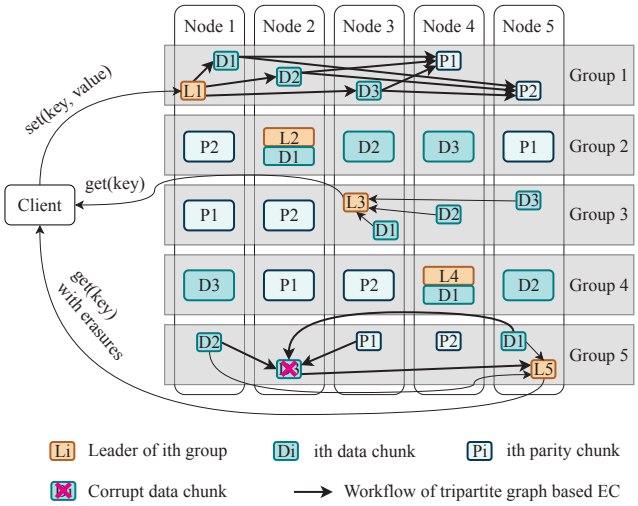


Figure 7: Interleaved Architecture of TriEC-Cache

As illustrated in Figure 7, the node layout of each group is static and pre-defined. For instance, in Group 1, Nodes 1, 2, and 3 store D_1, D_2, and D_3 in each stripe, respectively; and Nodes 4, 5 store P_1 and P_2 in each stripe, respectively. Therefore, the role of each node in a group for encoding procedure is static and pre-defined as well. As shown in the example of "*set(key, value)*", the nodes storing D_1, D_2, and D_3 are on the second layer, while the nodes storing P_1 and P_2 are on the third layer. On the other hand, the tripartite graph for decoding is dynamic and virtual; the role of each node depends on the category and status of the chunk it stores. For instance, in the example of "*get(key) with erasures*" shown in Figure 7, D_3 is assigned to be a second-layer node and responsible for reconstructing missing chunk, because it is the node originally storing the missing chunk. While D_1, D_2, and P_1 become third-layer nodes taking care of constructing and sending out intermediate chunks because they store necessary chunks for recovery and are chosen to participate in the decoding procedure. However, *TriEC* is a flexible paradigm, and applications are free to choose their strategies with *TriEC*.

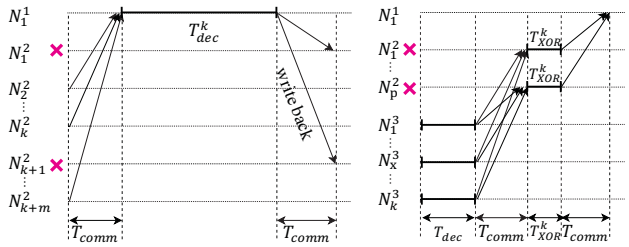
In TriEC-Cache, there are two types of metadata to be maintained: (1) key-value mappings, and (2) metadata of EC groups, such as topology information, connection states, etc. The metadata has to be updated frequently and efficiently; thus, TriEC-Cache maintains metadata separately from data with the replication scheme. Key-value mappings have to be stored with replication scheme, such that any process in an EC group is functionally complete.

5.2 In-Band Recovery of *TriEC*

As revealed in Figure 5, recovery with *BiEC* only recovers the requested chunks on the initiator side, and the nodes storing those corrupt chunks are left unrecovered and unhealthy. Thus, applications need to design an out-of-band recovery mechanism to reconstruct the lost or corrupt data and to bring these nodes back to healthy. Typically, there are two major out-of-band recovery approaches: (1) the initiator who completes constructing corrupt chunks writes them back to the corresponding nodes (as depicted in Figure 8a), and (2) the node who realizes that itself is unhealthy performs recovery in the background.

By contrast, *TriEC* provides an in-band recovery approach (as shown in Figure 8b), such that the nodes involved in reconstructing corrupt chunks become healthy at the end. Applications co-designed with *TriEC* do not have to design other mechanisms to recover unhealthy nodes and thus perform more efficiently.

To evaluate TriEC-Cache, we also implement *BiEC* on top of `memcached` (v1.5.12). To make fair comparisons, we choose write-back as the out-of-band recovery mechanism to make all unhealthy nodes recovered after decoding, e.g., N_1^1 writes the recovered chunks to N_1^2 and N_{k+1}^2 in parallel through Remote Direct Memory Access (RDMA) channels. The choose of write-back approach is based on the fact that write-back with a high-performance network is simple and fast, which can outperform the background-recovery counterpart.



(a) Out-of-Band Recovery (BiEC) (b) In-Band Recovery (TriEC)

Figure 8: Overview of Out-of-Band and In-Band Recoveries

5.3 Implementation and Optimization

TriEC-Cache supports two basic operations: *value* ← *get(key)* and *set(key, value)*, where *key* and *value* are arbitrary strings. The workflow of `set` and `get` operations are depicted in Figure 7.

When a leader receives a *set(key, value)* request from a client, it first stores the metadata, which is a record of *{key, size of value}*, and then starts a *TriEC* encoding procedure. The topology of all involved processes is a tripartite graph. Take $L_1, D_1, D_2, D_3, P_1, P_2$ in Group 1 for example, the corresponding tripartite graph is $\{\{L_1\}, \{D_1, D_2, D_3\}, \{P_1, P_2\}\}$. Within the encoding procedure, L_1 chunks the value into $k = 3$ pieces and sends them to $\{D_1, D_2, D_3\}$. Each process of $\{D_1, D_2, D_3\}$ receives a data chunk, computes $m = 2$ intermediate chunks, and delivers the intermediate chunks to $\{P_1, P_2\}$. Note that the computation and transmission of intermediate chunks are performed by the *encode-and-send* offload primitive. Each of $\{P_1, P_2\}$ gets k intermediate chunks and finally generates a parity chunk by XORing the intermediate chunks. L_1 only stores metadata, while data processes and parity processes store both metadata and data. The metadata stored in TriEC-Cache is a mapping of key→chunk.

When a request of *get(key)* comes to an EC group, the leader fetches data chunks from all data processes in its EC group. If all data chunks are healthy, the leader just responses the request with these chunks. By contrast, if some chunks are corrupt, the fetch request from leader triggers a *TriEC* decoding procedure. As shown

in the example in Group 5, the topology of the processes participating in the decoding procedure is a tripartite graph as well (i.e., $\{\{L_5\},\{D_3\},\{D_1,D_2,P_1\}\}$). After receiving the fetch request from L_5, D_3 realizes that the chunk stored by itself is corrupt or missing, and then it speculatively reads from other processes. Other processes including $\{D_1,D_2,P_1\}$ calculate the requested intermediate chunks and then send them back to D_3. The calculation and transmission of intermediate chunks are performed with the *recv-and-decode* offload primitive. D_3 proactively cancels other reads once received sufficient intermediate chunks. Prior studies [7, 29] indicate that proactive cancellation can help reduce bandwidth overheads of speculative reads. Since D_3 has all necessary intermediate chunks, it recovers the wanted chunk by XORing all the intermediate chunks. Once D_3 is back to healthy, it responses the fetch request from L_5 with the reconstructed chunk. After the triggered *TriEC* decoding completed, L_5 responses the request normally as usual.

There are two kinds of communication channels in TriEC-Cache. Data transmissions among leaders, data processes, and parity processes pass through RDMA channels, while other communications go through socket channels (i.e., IPoIB on our platforms). This design approach enables TriEC-Cache to serve any kind of `memcached` client without any code conversion on the client side, and upper-layer applications can get benefit from *TriEC* transparently.

In our implementation, there are several optimization techniques to overcome the limitations of Mellanox ConnectX-5 and to accelerate TriEC-Cache. Note that these optimizations are not relevant to *BiEC*, such that they are not applied to the *BiEC* baseline.

5.3.1 Avoidance of Sending Unrequested Chunks. Our design and implementation of *TriEC* highly depend on intermediate chunks. The NIC generating intermediate chunks does not need to send out the input chunks. However, with Mellanox's *encode-and-send* primitive, we have to assign a valid RDMA channel to each chunk involved in EC. To avoid sending out the input chunks which are not requested, an optimization of eliminating the work requests (i.e., nullifying the work requests) to the corresponding RDMA channels is applied in our implementation.

5.3.2 EC Calculator Cache. With Mellanox's EC offload APIs, initializing EC calculators is very expensive. Mellanox's driver registers a piece of memory for holding the EC matrix for every EC calculator, and the memory cannot be pre-allocated and pre-registered by upper-layer applications. To alleviate the expensive cost in initializing EC calculators, an EC calculator cache is used in our implementation. With EC calculator cache, calculations with the same EC matrix are able to reuse the same calculator. Thus the use of calculator cache saves a lot in calculator initializations. There are two cache techniques: (1) static EC calculator cache, and (2) dynamic EC calculator cache. Static EC calculator cache means that all calculators possibly to be used in the future are pre-initialized at the very beginning, and thus delivers the best latency performance to incoming calculations. However, the maximum number of EC calculators in the flight supported by current-generation Mellanox's driver is limited (i.e., about 512). Therefore, the static approach is not an appropriate design for large scale configurations like $RS(12,4)$, etc. By contrast, dynamic EC calculator cache is an approach that allocates and initializes EC calculators on demand, and the calculators in the cache are managed by the configured cache eviction policy. Though the calculator initializations are completed at runtime, the costs are amortized by incoming calculations. We finally employ dynamic EC calculator cache in the implementation of TriEC-Cache. We suggest that Mellanox can expose more flexible APIs for calculator initializations in the future, and thus, the costs of calculator initializations can be reduced to the least.

6 EVALUATION

In this section, we evaluate our proposed *TriEC* with microbenchmarks and evaluate its real-world implementation TriEC-Cache with extended YCSB. The base-lines for all experiments are *BiEC* and our *BiEC* implementation on top of `memcached` ($v1.5.12$).

6.1 Experimental Setup

The experiments in the paper are conducted on two clusters (i.e., A and B) as listed in Table 2. In this paper, we typically evaluate with five widely-used EC configurations, i.e., $RS(3,2)$ [14], $RS(6,3)$ [12, 14, 30], $RS(8,3)$ [47], $RS(10,4)$ [10, 14], and $RS(12,4)$ [17].

Table 2: Specifications of Clusters

Specification	OSU RI2 Cluster	OSC Pitzer Cluster
Processor	Intel Broadwell E5-2680 v4	Intel Skylake 6148
Frequency	2.4 GHZ	2.4 GHZ
RAM (DDR)	128 GB	192 GB
Interconnect	ConnectX-5 IB-EDR (100 Gbps)	ConnectX-5 IB-EDR (100 Gbps)
OS	CentOS 7.4	Red Hat 7.5
OFED	OFED-4.5-1.0.1	OFED-4.4-1.0.0
Scale	up to 25 nodes	up to 17 nodes

6.2 Microbenchmark

To evaluate the performance of different EC paradigms for NIC offload, we propose a microbenchmark suite consisting of an encoding microbenchmark and a decoding microbenchmark. In the encoding microbenchmark, each EC paradigm encodes a large in-memory file which is filled up with random strings. While in the decoding microbenchmark, an in-memory file is encoded first, and several encoded parts are erased based on the specified configuration, then each EC paradigm decodes the remaining parts to recover the original file. The execution time of encoding/decoding the entire file is reported by our microbenchmark suite.

Figures 9 and 10 illustrate the performance improvement gained by *TriEC* for encoding and decoding workload across multiple EC configurations and varied chunk sizes. Compared with *BiEC* for encoding workload, *TriEC* can reduce the overall execution time by up to 22.6%, 30.2%, 37.8%, 42.0% and 45.1% for $RS(3,2)$, $RS(6,3)$, $RS(8,3)$, $RS(10,4)$ and $RS(12,4)$, respectively. On the other hand, *TriEC* outperforms *BiEC* in terms of recovering m corrupt data chunks by $1.18-2.33$x for $RS(3,2)$, $1.24-2.26$x for $RS(6,3)$, $1.18-2.17$x for $RS(8,3)$, $1.21-2.16$x for $RS(10,4)$, and $1.23-2.83$x for $RS(12,4)$. The encoding experiment for $RS(3,2)$ with the chunk size of $512B$ in Figure 9 indicates that *BiEC* is also performing well with small EC configurations and chunk sizes.

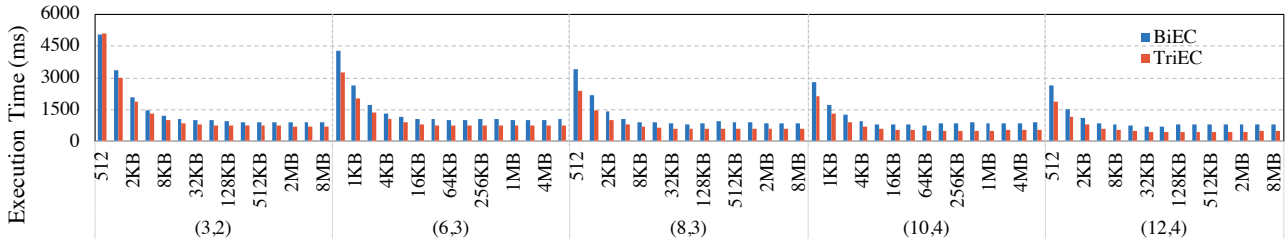


Figure 9: Encoding Performance Comparisons with Varied Configurations and Chunk Sizes (OSU RI2 Cluster). The (k, m) (e.g., (3, 2)) refers to an EC configuration of $RS(k, m)$. The performance improvements achieved by *TriEC* are 1.0-1.29x for RS(3, 2), 1.25-1.43x for RS(6, 3), 1.32-1.61x for RS(8, 3), 1.30-1.72x for RS(10, 4), and 1.31-1.82x for RS(12, 4). $k + m + 1$ nodes (One client, $k + m$ servers).

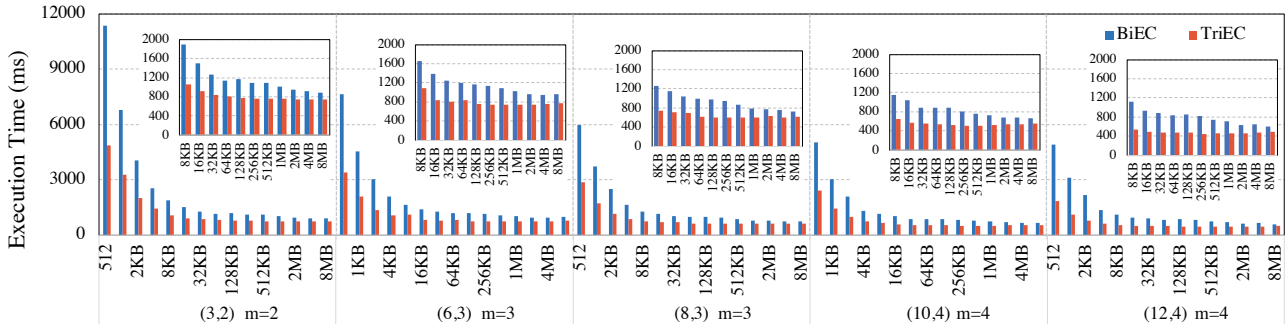


Figure 10: Performance Comparisons for Recovering m Data Chunks with Varied Configurations and Chunk Sizes (OSU RI2 Cluster). The (k, m) (e.g., (3, 2)) refers to an EC configuration of $RS(k, m)$. To recover m data chunks, *TriEC* reduces the overall execution time by up to 57.1%, 55.7%, 53.9%, 53.7%, and 64.6% for RS(3, 2), RS(6, 3), RS(8, 3), RS(10, 4), and RS(12, 4), respectively. $k + m + 1$ nodes (one client, $k + m$ servers).

In our microbenchmarks, there are always overlaps between successive operations for both encoding and decoding. Thus the microbenchmarks reveal the performance potential of *TriEC*. The results qualitatively validate our performance model and Theorem 2.

6.3 Performance in TriEC-Cache

We evaluate latency and throughput performance of TriEC-Cache with the extended YCSB workloads (see Section 6.3.1 for more details). In these experiments, we use $k + m$ EC groups for $RS(k, m)$ to fully utilize server resources. For the experiments of evaluating latency performance, there is only one client issuing requests. While for the experiments of evaluating throughput performance, there are 512 clients running simultaneously.

6.3.1 Workload. We extend YCSB [5] to generate the workloads for our evaluations. Each workload generated by YCSB represents a particular mix of operations (e.g., read, write, etc.), data sizes, request distributions, and so on. In each workload, there is a table of records, each with F fields. Each key in the workloads is generated by concatenating the table name and an identifier (e.g., *usertable-user8295266226408665858*). While the values are compressed objects consisting of multiple fields (named `field0`, `field1`, etc.). Each field is a random string of ASCII characters of length L.

In order to evaluate the performance of read with chunk recoveries, we bring a new operation (i.e., *read_with_erasures*) into YCSB. When TriEC-Cache receives a request of *read_with_erasures*, it will reconstruct data chunks and response the request. During the workload generation, whenever an operation is chosen to be *read*, it will further determine whether the *read* operation should be a *read_with_erasures*. In practice, failures are not always independent. Some prior studies [11, 15, 24, 38] demonstrate that **Weibull** distribution with Weibull shape parameter of 0.7–0.8 provides a much better fit for predicting failures on HPC systems. Hence, our extension in YCSB determines whether a *read* should be a *read_with_erasures* according to Weibull distribution. The study [38] also points out that failure rates vary widely across systems and depend mostly on system size and less on the type of hardware. Therefore, the ratio of *read_with_erasures* operations to *read* operations in our evaluations is fixed to 1%.

For the evaluations in this section, the request distribution of each workload is *Zipfian* [9] distribution, such that some records will be extremely popular while most records will be unpopular. A prior study from Facebook [28] presents that the median value sizes are 4.34KB for `Region`, and 10.7KB for `Cluster`. Therefore, for our evaluations, we choose value sizes of 1KB, 4KB, and 16KB, which are similar and representative to the value sizes in Facebook's `memcached` cluster. There are two kinds of workloads in our evaluation: (1) fixed-size workload, in which the total workload size is fixed to be 2GB for each EC group, and (2) variable-size workload. The fixed-size workload is the one used for evaluations if not explicitly mentioned. The read/write ratios of involved workloads include equal-shares (r:w=50:50), read-mostly (r:w=95:5), and read-only(r:w=100:0). If the workloads are configured with *read_with_erasures* enabled, then 1% of *read* operations in equal-shares, read-mostly, and read-only workloads are *read_with_erasures* operations.

6.3.2 Latency and Throughput. As the `read` requests are handled in the same way by both *BiEC* and *TriEC*, the read performance is very similar. Thus, we do not specially discuss read performance in this section.

Figure 11 shows that, compared with *BiEC*, *TriEC* reduces the average write latency of $RS(3, 2)$ by up to 23.2% and 12.0% for equal-shares (r:w=50:50) and read-mostly (r:w=95:5) workloads, respectively. Meanwhile, the improvement gained by *TriEC* for writes with $RS(6, 3)$ is up to 1.12x and 1.10x for equal-shares and read-mostly workloads, respectively. The speedup achieved by *TriEC* for the normal cases, i.e., there is no recovery occurring during reads, follows the same trend as depicted in Figure 11; thus we do not show them to save space.

On the other hand, the average latency of *read_with_erasures* is prominently improved by *TriEC* as well. For $RS(3, 2)$, the improvement is 1.25 − 1.61x for equal-shares, 1.27 − 1.60x for read-mostly, and 1.20 − 1.56x for read-only (r:w=100:0) workloads. For $RS(6, 3)$, *TriEC* achieves 1.24 − 1.40x, 1.22 − 1.32x, and 1.20 − 1.29x for equal-shares, read-mostly, and read-only workloads, respectively.

Moreover, we evaluate the overall throughput performance of TriEC-Cache for $RS(6, 3)$ on 17 nodes as shown in Figure 13. Compared with *BiEC*, *TriEC* speeds up the overall throughput performance by up to 13.3% for equal-shares, 14.8% for read-mostly, and 13.9% for read-only workloads.

Since the requests from YCSB clients are distributed across all the EC groups, and the percentages of *write* and *read_with_erasures* requests are at most 50% and 1%, there are few overlaps between *write* and *read_with_erasures* in these experiments. Therefore, the performance improvement shown in this section mainly comes from the parallelism of *TriEC*. In the future, we will integrate *TriEC* into distributed file systems, in which there are more chances to benefit from the overlapping capability of *TriEC*.

While prior evaluations are evaluated with the fixed-size workload, we also conduct performance evaluations with the variable-size workload on OSU RI2 Cluster. The throughput numbers are taken with the equal-shares (50:50) benchmark in YCSB for $RS(6, 3)$, in which the value size is fixed to $4KB$, and 1% of reads are with three erasures. The total number of keys are $9M$, $18M$, and $36M$ (i.e., total workload sizes are $36GB$, $72GB$, and $144GB$). In our experiments, both *BiEC* and *TriEC* perform better if more keys are stored in each EC group. Since the request distribution is *Zipfian*, the clients have more chance to request different servers in parallel if there are more keys; thus, the throughputs go up. Compared with *BiEC*, *TriEC* shows up 11% to 14% performance improvement.

6.3.3 Full-node Recovery. We evaluate the full-node recovery performance of TriEC-Cache for $RS(3, 2)$ using $16KB$ value size with the variable-size workload. As shown in Figure 14, *TriEC* reduces the execution time by up to 28% to fully recover one node, and is demonstrated to be scalable with respect to the number of keys to be reconstructed.

6.3.4 Calculator Cache. Calculator cache is an important optimization in TriEC-Cache to alleviate the expensive cost in initializing EC calculators with Mellanox's EC offload APIs. We conduct experiments to evaluate the average latencies of *read_with_erasures* with different calculator cache technologies. The performance numbers are taken with the read-only benchmark in YCSB for $RS(3, 2)$, in which the value size is fixed to $1KB$, and the total number of operations is $300K$. As revealed in Figure 15, the uses of static cache and dynamic cache reduce the average latency of *read_with_erasures* for $RS(3, 2)$ by 93.1% and 86.5%, respectively.

7 RELATED WORK

Erasure Coding for Storage Systems Erasure coding has been extensively adopted in many widely-used storage systems [1, 2, 12, 14, 20, 27, 30] to take advantage of the capability of delivering higher data reliability and durability with prominent lower storage overhead. Towards making EC viable for large scale storage systems, multiple works along different directions have been proposed. One direction is to reduce the recovery overhead by the assistances of *local parities*, such as Local Reconstruction Codes in [17] and Locally Repairable Codes in [37]. Meanwhile, several approaches to reduce the repair bandwidth (e.g., [8, 16, 21, 26, 34]) have been proposed. On the other hand, after several work of applying EC for Big Data and Cloud storage systems (e.g., [3, 4, 11, 13, 19, 23, 33, 46]), EC is also being employed to design resilient key-value store systems, including, Cocytus [48], EC-Cache [32], Hybris [44], BCSore [22], and [39, 43]. This increased focus on EC for storage resilience serves as a motivation for this paper.

Erasure Coding Offload With the emergence of the next generation hardware which is capable of offloading computation, several erasure coders have been proposed to leverage the next-generation hardware to accelerate EC calculations fully. For instance, Gibraltar [6] is a famous GPU-based erasure coder which takes advantage of GPU's massively multicore architecture. After comprehensive characterizing erasure coders on next-generation hardware [40], we found several critical limitations of current-generation EC NIC offload schemes on modern SmartNICs, such as Mellanox ConnectX-4 (and later) [25]. Motivated by the insights, we propose *TriEC* to leverage the capabilities of high-performance network adapters effectively.

8 CONCLUSION AND FUTURE WORK

In this paper, we discussed three identified limitations of current-generation EC NIC offload schemes on modern SmartNICs (e.g., Mellanox ConnectX-5), such as bipartite-based EC encoding and decoding (*BiEC*) which underutilize the networked computing resources, missing the support of receive-and-decode primitive, and semantic mismatch between application requirements and supported primitives. To address these limitations, this paper proposes a new EC NIC offload paradigm based on the tripartite graph model, namely *TriEC*. *TriEC* decomposes a full EC calculation pipeline into three stages and each stage only executes a subset of EC tasks in parallel. *TriEC* supports both encode-and-send and receive-and-decode operations efficiently. We prove that *TriEC* is correct and has the potential to deliver better performance than current-generation *BiEC* based NIC offload schemes. Experiments show that *TriEC* outperforms *BiEC* by up to 1.82x and 2.33x for encoding and recovering, respectively. With extended YCSB workloads, *TriEC* reduces the average write latency by up to 23.2% and the recovery time by up to 37.8% even with only 1% failure occurrences under the Weibull distribution. Moreover, *TriEC* outperforms *BiEC* by 1.32x for a full-node recovery with 8 million records. The performance evaluations qualitatively validate our performance model.

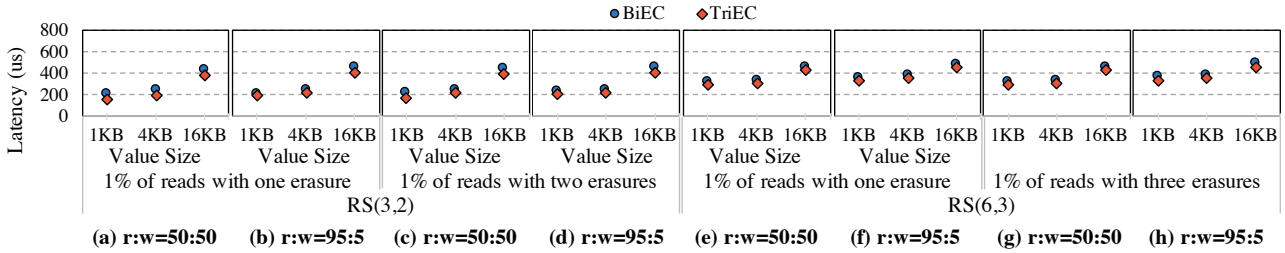


Figure 11: Write Latencies with Varied Configurations and Value Sizes (OSU RI2 Cluster). $k + m + 1$ nodes (one client, $k + m$ EC groups) for $RS(k, m)$.

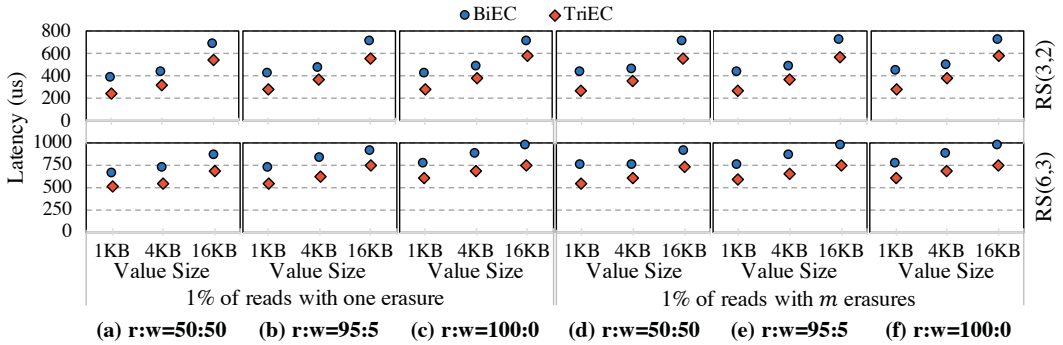


Figure 12: Latencies of `read_with_erasures` with Varied Configurations and Value Sizes (OSU RI2 Cluster). $k + m + 1$ nodes (one client, $k + m$ EC groups) for $RS(k, m)$.

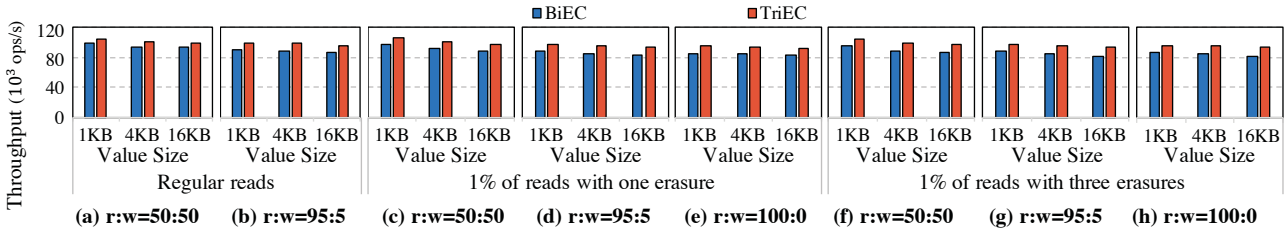


Figure 13: Throughput Comparisons for $RS(6, 3)$ (OSC Pitzer Cluster). 17 nodes (512 clients on eight nodes, nine EC groups).

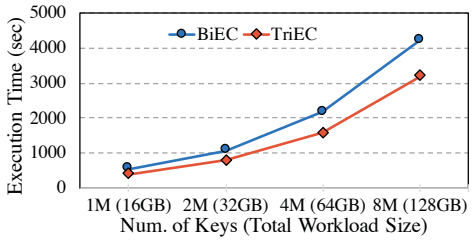


Figure 14: Execution Time to Recover One Node with Varied Number of Keys (OSC Pitzer Cluster). Six nodes (one client, five EC groups).

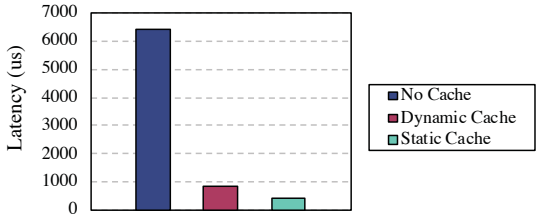


Figure 15: Performance Impact of Calculator Cache (OSU RI2 Cluster). Six nodes (one client, five EC groups).

In the future, we plan to extend *TriEC* along two directions: (1) applying *TriEC* on other hardware devices (e.g., CPUs and GPUs), or even combining *TriEC* with our previous work Multi-Rail EC [41, 42] to take advantage of multiple EC-capable devices in parallel, and (2) co-designing *TriEC* with other types of storage systems, such as distributed file systems and cloud storage systems.

9 ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their precious feedback, and Ohio Supercomputer Center (OSC) for providing the cluster access. This work is supported in part by National Science Foundation grant #CCF-1822987.

REFERENCES

[1] Backblaze. 2015. Backblaze Reed-Solomon. https://www.backblaze.com/open-source-reed-solomon.html.

[2] Ceph. 2016. Ceph Erasure Coding. http://docs.ceph.com/docs/master/rados/operations/erasure-code/.

[3] Jeremy C. W. Chan, Qian Ding, Patrick P. C. Lee, and Helen H. W. Chan. 2014. Parity Logging with Reserved Space: Towards Efficient Updates and Recovery in Erasure-coded Clustered Storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST'14)*. USENIX Association, Berkeley, CA, USA, 163–176.

[4] Byung-Gon Chun, Frank Dabek, Andreas Haeberlen, Emil Sit, Hakim Weatherspoon, M. Frans Kaashoek, John Kubiatowicz, and Robert Morris. 2006. Efficient Replica Maintenance for Distributed Storage Systems. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3 (NSDI'06)*. USENIX Association, Berkeley, CA, USA, 4–4.

[5] Cooper, Brian F. and Silberstein, Adam and Tam, Erwin and Ramakrishnan, Raghu and Sears, Russell. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*. ACM, New York, NY, USA, 143–154.

[6] Matthew L. Curry, Anthony Skjellum, H. Lee Ward, and Ron Brightwell. 2011. Gibraltar: A Reed-Solomon Coding Library for Storage Applications on Programmable Graphics Processors. *Concurr. Comput. : Pract. Exper.* 23, 18 (Dec. 2011), 2477–2495.

[7] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56, 2 (Feb. 2013), 74–80.

[8] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran. 2010. Network Coding for Distributed Storage Systems. *IEEE Transactions on Information Theory* 56, 9 (Sep. 2010), 4539–4551.

[9] L. Egghe. 2005. Zipfian and Lotkaian Continuous Concentration Theory: Research Articles. *J. Am. Soc. Inf. Sci. Technol.* 56, 9 (July 2005), 935–945.

[10] Facebook. 2010. Facebook's Erasure Coded Hadoop Distributed File System (HDFS-RAID). https://github.com/facebookarchive/hadoop-20.

[11] Daniel Ford, François Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. 2010. Availability in Globally Distributed Storage Systems. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, Berkeley, CA, USA, 61–74.

[12] Google. 2012. Colossus: Successor to the Google File System (GFS). https://www.systutorials.com/3202/colossus-successor-to-google-file-system-gfs/.

[13] Kevin M Greenan, Xiaozhou Li, and Jay J Wylie. 2010. Flat XOR-based erasure codes in storage systems: Constructions, efficient recovery, and tradeoffs. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. 1–14.

[14] Apache Hadoop. 2017. Apache Hadoop 3.0.0. http://hadoop.apache.org/docs/r3.0.0/.

[15] Taliver Heath, Richard P. Martin, and Thu D. Nguyen. 2002. Improving Cluster Availability Using Workstation Validation. In *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '02)*. ACM, New York, NY, USA, 217–227.

[16] Yuchong Hu, Henry C. H. Chen, Patrick P. C. Lee, and Yang Tang. 2012. NCCloud: Applying Network Coding for the Storage Repair in a Cloud-of-clouds. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*. USENIX Association, Berkeley, CA, USA, 21–21.

[17] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. 2012. Erasure Coding in Windows Azure Storage. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (USENIX ATC'12)*. USENIX Association, Berkeley, CA, USA, 2–2.

[18] Intel. 2016. Intel Intelligent Storage Acceleration Library (Intel ISA-L). https://software.intel.com/en-us/storage/ISA-L.

[19] Osama Khan, Randal Burns, James Plank, William Pierce, and Cheng Huang. 2012. Rethinking Erasure Codes for Cloud File Systems: Minimizing I/O for Recovery and Degraded Reads. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*. USENIX Association, Berkeley, CA, USA, 20–20.

[20] Chunbo Lai, Song Jiang, Liqiong Yang, Shiding Lin, Guangyu Sun, Zhenyu Hou, Can Cui, and Jason Cong. 2015. Atlas: Baidu's Key-value Storage System for Cloud Data. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 1–14.

[21] Runhui Li, Xiaolu Li, Patrick P. C. Lee, and Qun Huang. 2017. Repair Pipelining for Erasure-coded Storage. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '17)*. USENIX Association, Berkeley, CA, USA, 567–579.

[22] Shenglong Li, Quanlu Zhang, Zhi Yang, and Yafei Dai. 2017. BCStore: Bandwidth-Efficient In-memory KV-store with Batch Coding. In *International Conference on Massive Storage Systems and Technology (MSST)*.

[23] Xiaolu Li, Runhui Li, Patrick PC Lee, and Yuchong Hu. 2019. OpenEC: Toward Unified and Configurable Erasure Coding Management in Distributed Storage Systems. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. 331–344.

[24] T-TY Lin and Daniel P Siewiorek. 1990. Error Log Analysis: Statistical Modeling and Heuristic Trend Analysis. *IEEE Transactions on Reliability* 39, 4 (1990), 419–432.

[25] Mellanox. 2016. Understanding Erasure Coding Offload. https://community.mellanox.com/docs/DOC-2414.

[26] Subrata Mitra, Rajesh Panta, Moo-Ryong Ra, and Saurabh Bagchi. 2016. Partial-Parallel-Repair (PPR): A Distributed Technique for Repairing Erasure Coded Storage. In *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 30.

[27] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, et al. 2014. f4: Facebook's Warm BLOB Storage System. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, 383–398.

[28] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. 2013. Scaling Memcache at Facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. 385–398.

[29] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 69–84.

[30] Michael Ovsiannikov, Silvius Rus, Damian Reeves, Paul Sutter, Sriram Rao, and Jim Kelly. 2013. The Quantcast File System. *Proceedings of the VLDB Endowment* 11 (2013), 1092–1101.

[31] James S Plank, Scott Simmerman, and Catherine D Schuman. 2008. Jerasure: A Library in C/C++ Facilitating Erasure Coding for Storage Applications. (2008).

[32] KV Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. 2016. EC-Cache: Load-Balanced, Low-Latency Cluster Caching with Online Erasure Coding. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association.

[33] K.V. Rashmi, Nihar B. Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. 2014. A "Hitchhiker's" Guide to Fast and Efficient Data Reconstruction in Erasure-coded Data Centers. *Proceedings of the 2014 ACM Conference on SIGCOMM* 44, 4 (Aug. 2014), 331–342.

[34] K. V. Rashmi, Preetum Nakkiran, Jingyan Wang, Nihar B. Shah, and Kannan Ramchandran. 2015. Having Your Cake and Eating It Too: Jointly Optimal Erasure Codes for I/O, Storage and Network-bandwidth. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. USENIX Association, Berkeley, CA, USA, 81–94.

[35] K. V. Rashmi, Nihar B. Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. 2013. A Solution to the Network Challenges of Data Recovery in Erasure-coded Distributed Storage Systems: A Study on the Facebook Warehouse Cluster. In *Proceedings of the 5th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage'13)*. USENIX Association, Berkeley, CA, USA, 8–8.

[36] Irving S Reed and Gustave Solomon. 1960. Polynomial Codes Over Certain Finite Fields. *J. Soc. Indust. Appl. Math.* 8, 2 (1960), 300–304.

[37] Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros G. Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. 2013. XORing Elephants: Novel Erasure Codes for Big Data. *Proceedings of the VLDB Endowment* 6, 5 (March 2013), 325–336.

[38] Bianca Schroeder and Garth Gibson. 2010. A large-scale study of failures in high-performance computing systems. *IEEE Transactions on Dependable and Secure Computing* 7, 4 (2010), 337–350.

[39] Dipti Shankar, Xiaoyi Lu, and D. K. Panda. 2017. High-Performance and Resilient Key-Value Store with Online Erasure Coding for Big Data Workloads. In *Proceedings of the 37th IEEE International Conference on Distributed Computing Systems (ICDCS)*.

[40] Haiyang Shi, Xiaoyi Lu, and Dhabaleswar K. (DK) Panda. 2018. EC-Bench: Benchmarking Onload and Offload Erasure Coders on Modern Hardware Architectures. In *International Symposium on Benchmarking, Measuring and Optimizing (Bench'18)*. Springer.

[41] Haiyang Shi, Xiaoyi Lu, Dipti Shankar, and Dhabaleswar K. Panda. 2019. UMR-EC: A Unified and Multi-Rail Erasure Coding Library for High-Performance Distributed Storage Systems. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '19)*. ACM, New York, NY, USA, 219–230.

[42] Haiyang Shi, Xiaoyi Lu, Dipti Shankar, and Dhabaleswar K. (DK) Panda. 2018. High-Performance Multi-Rail Erasure Coding Library over Modern Data Center Architectures: Early Experiences. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC '18)*. ACM, New York, NY, USA, 530–531.

[43] Konstantin Taranov, Gustavo Alonso, and Torsten Hoefler. 2018. Fast and Strongly-consistent Per-item Resilience in Key-value Stores. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*. ACM, New York, NY, USA, Article 39, 14 pages.

[44] Paolo Viotti, Dan Dobre, and Marko Vukolić. 2017. Hybris: Robust Hybrid Cloud Storage. *ACM Trans. Storage* 13, 3, Article 27 (Sept. 2017), 32 pages.

[45] Hakim Weatherspoon and John D Kubiatowicz. 2002. Erasure Coding vs. Replication: A Quantitative Comparison. In *International Workshop on Peer-to-Peer Systems*. Springer, 328–337.

[46] Mingyuan Xia, Mohit Saxena, Mario Blaum, and David A. Pease. 2015. A Tale of Two Erasure Codes in HDFS. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*. USENIX Association, Santa Clara, CA, 213–226.

[47] Yahoo. 2019. Yahoo Cloud Object Store. https://yahooeng.tumblr.com/post/116391291701/yahoo-cloud-object-store-object-storage-at.

[48] Heng Zhang, Mingkai Dong, and Haibo Chen. 2016. Efficient and Available In-memory KV-Store with Hybrid Erasure Coding and Replication. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX Association, Santa Clara, CA, 167–180.