INEC: Fast and Coherent In-Network Erasure Coding

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) is a promising fault tolerance scheme that has been applied to many well-known distributed storage systems. The capability of Coherent EC Calculation and Networking on modern SmartNICs has demonstrated that EC will be an essential feature of in-network computing. In this paper, we propose a set of coherent in-network EC primitives, named INEC. Our analyses based on the proposed α-β performance model demonstrate that INEC primitives can enable different kinds of EC schemes to fully leverage the EC offload capability on modern SmartNICs. We implement INEC on commodity RDMA NICs and integrate it into five state-of-the-art EC schemes. Our experiments show that INEC primitives significantly reduce 50^{th}, 95^{th}, and 99^{th} percentile latencies, and accelerate the end-to-end throughput, write, and degraded read performance of the key-value store co-designed with INEC by up to 99.57%, 47.30%, and 49.55%, respectively.**

***Index Terms*—Next Generation Networking, In-Network Computing, Fault Tolerance, Erasure Coding**

I. Introduction

Data-intensive applications have increasing demands for high-performance and reliable storage systems on modern HPC clusters. Many existing HPC storage systems adopt the n-way replication technique to ensure data reliability and availability with high storage cost [1]–[3]. On the other hand, Erasure Coding (EC) has become a prevalent alternative to data replication in fault-tolerant distributed storage systems for years. One of the most popular erasure codes is called *Reed-Solomon (RS)* code [4], which has been widely adopted in many distributed storage systems and parallel file systems [5]–[15]. The use of EC in storage systems can significantly reduce the storage cost; however, it usually causes performance degradation as the trade-off.

To alleviate the performance overhead of using EC in storage systems, many advanced EC schemes [11], [16]–[22] have been proposed in the community to accelerate EC. Among these studies, we have surveyed five different kinds of EC schemes, including conventional RS Code [4], Local Reconstruction Code (LRC) [11], Partial Parallel Repair (PPR) Code [16], Repair Pipelining (ECPipe) Code [17], and Tripartite Graph based EC (TriEC) [18], which can represent the state-of-the-art EC designs.

Figure 1 summarizes these schemes and their required functional primitives, while more details about these EC schemes can be found in Section II-B. From this figure, we can clearly see the fact that all these EC schemes have their encoding and decoding calculations tightly coupled with communication tasks for data distribution (represented by edges in the figure) in distributed storage systems. Take PPR for example (see the description in the caption of Figure 1 for more details), the required primitive set of Layer-2 for constructing a PPR protocol is {wait (for receive completion), EC, wait (for EC completion), send}. It is obvious that the first *wait* and *send* are networking operations, and EC calculation is tightly interleaved in.

We can see similar functional requirements of other EC schemes as well in Figure 1. This common pattern exposes that the performance of EC schemes employed in distributed storage systems is heavily relying on the performance of underlying networking systems. Such a demand on high-performance networks draws the attentions of some major network interface card (NIC) vendors to provide support for EC.

A. Motivation and Challenges

The high-speed interconnect community has proposed several commodity SmartNICs that support EC on the NIC. For instance, Mellanox ConnectX-5 delivers advanced EC offload capability [22]. However, the current-generation SmartNICs' APIs expose EC and networking functionality disjointedly, without awareness of the fact that EC calculation is in close conjunction with networking. To take full advantage of SmartNICs' capability and reduce CPU involvement for further shortening the latency, one desired design approach is Coherent EC Calculation and Networking, or **Coherent In-Network EC**. Instead of exposing separate APIs for performing EC and networking operations incoherently, coherent in-network EC provides joint APIs, such as *recv-ec-send* to perform EC calculation and networking coherently. These coherent APIs can offload a task set onto SmartNICs, which will take care of waiting for receive completion, performing EC calculation, waiting for EC completion, and sending out results. By this means, *recv-ec-send* is a primitive that is required by PPR as discussed above, yet requires much less CPU involvement. As depicted in Figure 2, with coherent in-network EC, a *recv-ec-send* primitive offloads a task list to a SmartNIC, and the completion of each task automatically activates the execution of the subsequent task without CPU involvement.

Even though the benefits of coherent in-network EC are clearly shown in Figure 2, there are still many challenges to be resolved to provide complete and efficient coherent in-network EC functionality. First of all, as we see in Figure 1, different EC protocols require different communication graphs and steps for finishing EC tasks. To support all of them effectively and

This work is supported in part by NSF research grant CCF-1822987.

SC20, November 9-19, 2020, Is Everywhere We Are
978-1-7281-9998-6/20/$31.00

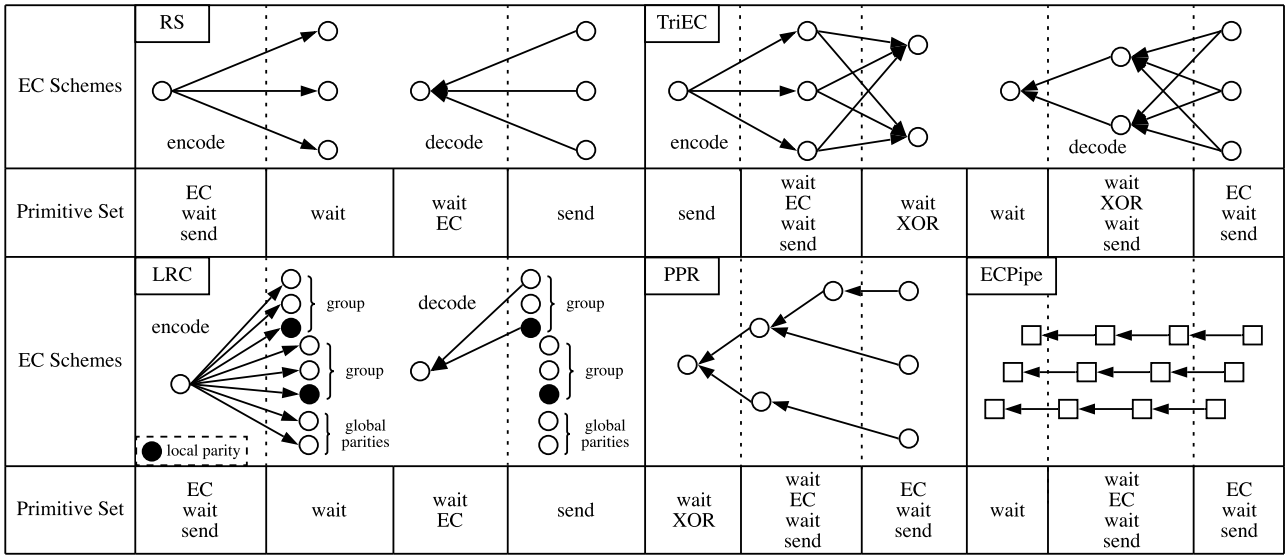


Fig. 1: State-of-the-Art EC Schemes and Primitive Sets for Layers. Take PPR as an example, since PPR is a reduction tree structure, all nodes can be categorized into root node, internal nodes, and leaf nodes. Let *Layer-1* denote root node, *Layer-2* denote internal nodes, and *Layer-3* denote leaf nodes. To reconstruct chunk $D^* = \sum_i^k \{S_i \cdot g_i'\}$, where S_i indicates a survived chunk and g_i' indicates the corresponding coefficient for reconstruction, nodes on Layer-3 calculate $R_{\text{Layer}-3} = S_i \cdot g_i'$ and send the results to Layer-2. Such that the primitive set of Layer-3 is {EC (i.e., calculate $R_{\text{Layer}-3}$), wait (for EC completion), send}. In the meantime, nodes on Layer-2 wait for $R_{\text{Layer}-3}$ from children, calculate $R_{\text{Layer}-2} = S_i \cdot g_i' + \sum_{\text{children}} R_{\text{Layer}-3}$, and send $R_{\text{Layer}-2}$ to Layer-1. In a nutshell, Layer-2's primitive set is {wait (for $R_{\text{Layer}-3}$), EC (i.e., calculate $R_{\text{Layer}-2}$), wait (for EC completion), send}. Finally, Layer-1 XORs on all the received results from Layer-2 to get the repaired chunk (i.e., $D^* = \sum_{\text{children}} R_{\text{Layer}-2}$), so its primitive set is {wait (for $R_{\text{Layer}-2}$), XOR}.

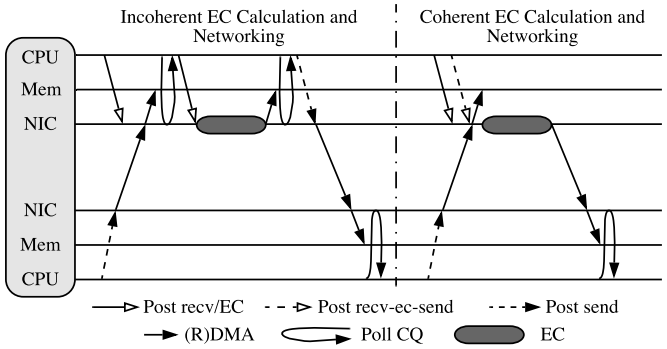


Fig. 2: Incoherent vs. Coherent. Note that DMAs issued by NICs in performing EC and RDMA SEND/WRITE_WITH_IMM are not shown for simplicity and clarity.

efficiently, what should be the common primitives that the underlying network needs to provide?

In addition, designing coherent in-network EC schemes is challenging because we need to find a fast way to connect many subsequent tasks together to construct a distributed EC pipeline and trigger pre-posted tasks on the NIC without CPU involvement. Can these be doable for all possibly defined EC primitives and the EC protocols mentioned above?

Moreover, from the bottom (hardware driver) up (application), how can we analyze and verify the effectiveness and efficiency of the proposed EC primitive and protocol designs?

B. Contribution

To overcome these challenges, some prior studies [18], [22] shed light on the opportunities to design efficient coherent in-network EC for conventional or a particular enhanced RS code. However, these designs can either only support one type of code or just partially offload some steps to the network, which makes them not fully address the challenges we have identified above and results in suboptimal performance. On the other hand, recent studies [22], [23] also point out that modern Remote Direct Memory Access (RDMA) capable NICs (RNICs) provide some low-level mechanisms such as *RDMA WAIT*, which can trigger events between two communication channels on the same RNIC.

With these opportunities, this paper proposes a holistic approach to design a set of fast and coherent in-network EC primitives, named *INEC*. Our approach makes the following contributions: 1) We successfully identify that to fully support all five different state-of-the-art EC protocols, we only need three types of essential coherent in-network EC primitives (i.e., *ec/xor-send*, *recv-ec/xor-send*, and *recv-ec/xor*). 2) To fully offload these EC primitives and protocols to the network, we propose efficient designs with *RDMA WAIT* to deliver fast EC and networking capability, and have implemented INEC primitives within Mellanox OFED driver. To the best of our knowledge, this is the first design of proposing a complete set of coherent in-network EC primitives with *RDMA WAIT* on RNICs to support multiple types of EC protocols. 3) We come up with an α-β performance model to analyze the performance gains of INEC primitives for five state-of-the-art EC schemes. 4) Through benchmarking and co-designing with a key-value store system, the proposed INEC primitives with five state-of-the-art EC protocols are validated from the bottom up. Our microbenchmarks show that INEC primitives accelerate state-

of-the-art EC schemes' encoding and decoding bandwidth by up to 5.87× and 2.94×, respectively. Evaluations on YCSB workloads illustrate that the co-designed key-value store with INEC attains up to 99.57% improvement for end-to-end throughput, and reduces up to 47.30% and 49.55% for write and degraded read latency, respectively.

II. Background

A. Erasure Coding Basics

A *Reed-Solomon (RS)* code [4] can be specified as *RS(k, m)*, which means the encoder takes k data chunks and generates m parity chunks to form a stripe (i.e., $k+m$ coded chunks), and the decoder is able to reconstruct the original data chunks from any k chunks out of the stripe. A typical encoding procedure is represented by $\boldsymbol{C} \times \boldsymbol{D} = \boldsymbol{E}$, in which $\boldsymbol{D}$ is a vector of k data chunks, $\boldsymbol{C}$ is a matrix of $k+m$ rows and k columns represented by $\boldsymbol{C} = \begin{bmatrix} \boldsymbol{I} \\ \hline \boldsymbol{G} \end{bmatrix}$, and $\boldsymbol{E}$ is a vector of $k+m$ encoded chunks. $\boldsymbol{I}$ is a $k \times k$ identity matrix, such that the first k chunks after encoding are identical with the data chunks, and we name the other m chunks as parity chunks. This property is named as *systematic*, which enables applications to read original content without extra computation if all data chunks (i.e., the first k chunks in the stripe) survive. The $m \times k$ matrix $\boldsymbol{G}$ is a generator matrix yielding the *Maximum Distance Separable (MDS)* property, which guarantees that any k chunks are sufficient to reconstruct the original data chunks.

The decoding procedure is based on the fundamental property of matrix $\boldsymbol{C}$, which is such that any $k \times k$ submatrix is invertible. Let $\{S_1, S_2, \cdots, S_k\}$ denote any k survived chunks in a stripe and $\boldsymbol{C}_i$ denote the corresponding row to S_i in matrix $\boldsymbol{C}$. Therefore, data chunks can be reconstructed by multiplying survived chunks with the inverse of the submatrix of $\boldsymbol{C}$, which is composed of $\boldsymbol{C}_1, \boldsymbol{C}_2, \cdots, \boldsymbol{C}_k$.

B. State-of-the-Art EC Schemes

Several state-of-the-art EC schemes have been proposed in the community to accelerate EC calculation. In addition to the conventional RS code, we have mainly studied the following enhanced codes in this paper.

Local Reconstruction Code (LRC) [11] is an EC scheme used in Microsoft Azure, which splits k data chunks in each stripe into l groups, and generates one local parity chunk for each group during encoding. If a group loses one data chunk, the missing chunk is able to be recovered by other data chunks in the group plus the local parity chunk for this group (see LRC in Figure 1). In this case, the coder only needs to decode on k/l chunks, which reduces the network overhead as well as the time to perform data reconstruction. On the other hand, if there are multiple lost chunks in a group, LRC will fall back to RS scheme, such that the coder fetches any k healthy chunks from data chunks in multiple groups as well as global parity chunks to proceed with the reconstruction.

Partial Parallel Repair (PPR) [16] divides a reconstruction operation into sub-operations and schedules them on multiple servers for better overlapping and parallelism. The involved servers are arranged as a reduction tree (see PPR in Figure 1), thus the reconstruction takes logarithmic time steps to complete. The tree-like structure fully utilizes available network resources and significantly reduces repair time.

Repair Pipelining (ECPipe) [17] separates a chunk into a set of slices and pipelines the reconstruction across storage nodes in units of slices. As we can see that ECPipe in Figure 1 repairs a chunk by dividing each involved chunk into three slices and conducting pipelined repair in units of slices in parallel. Therefore, it balances repair traffic, appropriately makes use of network bandwidth, and accelerates chunk repair.

Tripartite Graph-Based EC (TriEC) [18] decomposes encoding and decoding calculation into sub-operations, and assigns them to involved storage nodes. These nodes are put into different layers in a tripartite graph according to the category of operations assigned to each node (see TriEC in Figure 1). The tripartite graph structure takes advantage of EC offload capability on SmartNICs to bring more parallelism and overlapping, fully use network and compute resources, and reduce encoding and decoding time.

C. RDMA WAIT on Modern High-Speed Networks

RDMA WAIT is a network-level operation that enables work request (WR) triggering across two communication channels. Although it is not defined in RDMA specification [24], modern commodity RNICs have supported it (e.g., Mellanox CORE-Direct [25] and Cross Channel [26]). As illustrated in Figure 3, a WAIT WR holds execution of subsequent pre-posted WRs (i.e., Op) in the same send queue (SQ) until a pre-specified number of completions in a completion queue (CQ) are met. Note that the *trigger* and *activate* operations are conducted by RNIC without CPU involvement. Since WAIT WRs can only be triggered by work completions, RDMA WAIT has to cooperate with two-sided RDMA operations such as SEND and WRITE_WITH_IMM.

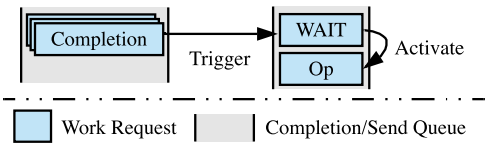


Fig. 3: Overview of RDMA WAIT

III. INEC Design

This section illustrates the design of INEC, including explanations of proposed primitives and implementation details.

A. Primitive Design

In a thorough exploration, we abstract three categories of coherent in-network EC primitives, which, together with networking functionality (e.g., RDMA SEND and RECV), are sufficient to express all the state-of-the-art EC schemes in Figure 1. Listing 1 represents the APIs of INEC primitives.

ec/xor-send: In almost every state-of-the-art EC scheme, there are nodes being responsible for performing EC/XOR calculation and sending the output to other nodes in the cluster (e.g., Layer-1 in the encoding structures of RS and LRC in Figure 1).

We abstract this process as *ec/xor-send*, which enables upper-layer applications to offload this process to SmartNICs and benefit from coherent in-network EC. As shown in Listing 1, *inec_ec_send* is the function of the *ec/xor-send* primitive. Parameter `calc` represents a data structure which describes calculator type (i.e., EC or XOR calculator), generator matrix for EC calculation, etc. The memory layout is described by the `mem` parameter, while `stripe` represents the communication channels for sending the EC stripe.

Listing 1: INEC Primitives

```
int inec_ec_send(struct inec_calc_desc *calc, struct
    inec_mem_desc *mem, struct inec_stripe_desc *stripe);
int inec_recv_ec_send(struct inec_wait_desc *wait, struct
    inec_calc_desc *calc, struct inec_mem_desc *mem,
    struct inec_stripe_desc *stripe);
int inec_recv_ec(struct inec_wait_desc *wait, struct
    inec_calc_desc *calc, struct inec_mem_desc *mem);
```

recv-ec/xor-send: Many EC schemes in the community decompose encoding and/or decoding calculation into subproblems and schedule them to different nodes involved in the computation to gain better parallelism, overlapping, and full use of available resources. These EC schemes, therefore, have complex and well-defined structures. Our exploration shows that the internal nodes of these structures share the same process pattern. As demonstrated in Figure 1, Layer-2 (i.e., internal nodes) of PPR, ECPipe, and TriEC, waits for the result(s) from upstream(s) to arrive, computes on both local chunk(s) and received result(s), and sends the output(s) to downstream(s). The process is abstracted as *recv-ec/xor-send*.

Primitive *recv-ec/xor-send* requires one more parameter (i.e., the `wait` parameter of *inec_recv_ec_send* in Listing 1), which indicates the completion queues (CQs) and the number of completions of each CQ to wait for.

recv-ec/xor: Another common process pattern is depicted by the decoding process of RS in Figure 1. Layer-1 of RS receives chunks from remote peers and performs EC calculation on the received chunks to reconstruct data chunks. The same process also appears in LRC, PPR, ECPipe, and TriEC. We use the *recv-ec/xor* primitive to abstract this process.

The function of the *recv-ec/xor* primitive is *inec_recv_ec* in Listing 1. Parameter `wait` specifies CQs and the number of receive completions of each CQ to wait for. The calculator type and generator matrix for EC computation are defined in `calc`. The memory to place received and generated chunks is identified by the `mem` parameter.

The key technique required to implement INEC on SmartNICs is an event-driven mechanism, such that the subsequent sub-tasks in a primitive can be triggered by the completion of previous sub-tasks. In Section III-C, we will elaborate on how to efficiently implement INEC on commodity RNICs.

B. Primitive Analysis

In this section, we extend the methodology proposed in [27] to study the performance factors of INEC primitives.

We analyze INEC along three substantial dimensions:

Operations: The number of operations must be performed to accomplish an in-network EC request. The EC basics in Section II-A illustrate that EC computation is a kind of matrix multiplication, so we can get the time complexities of EC calculations by counting the operations occurred in matrix multiplications. Recall that the time complexity of multiplying an $x \times y$ matrix and a $y \times z$ matrix is $O(x \cdot y \cdot z)$. On the other hand, the time complexity of computing XOR of n chunks is $O(n \cdot c)$, where c is the chunk size. Based on these time complexities, we conclude the *Operations* metric for each INEC primitive within each EC scheme in Table I. Take *ec_send* of RS(k, m) encode for example, it multiplies an $m \times k$ generator matrix with a $k \times c$ matrix consisting of k data chunks. Therefore, the number of operations taken place in the *ec_send* primitive is $O(k \cdot m \cdot c)$.

Communication: We use $T = L + n \cdot c/B$ to model the communication, where T is communication time, L is network latency, n is the number of chunks to be transmitted simultaneously, c is chunk size, and B is network bandwidth. For instance, the *recv_ec_send* primitive of 'TriEC Encode' receives one chunk and outputs m intermediate chunks, such that its *Communication* metric is $O(L + m \cdot c/B)$.

Gains: To estimate the potential benefits of coherent in-network primitives, we introduce the *Gains* metric, which includes three sub metrics: 1) Metric *States* indicates the number of extra states needs to be maintained for each coherent request with respect to its incoherent counterpart. As aforementioned in Section III-A, INEC primitives have to track extra states to determine when to activate subsequent tasks. For instance, the use of *recv_ec_send* in PPR(k, m) needs to maintain $O(in)$ states (in is the number of input chunks), such that the receive completions of in input chunks can trigger the subsequent EC computation. In the meantime, it also maintains one extra state to make the subsequent send operation after EC computation to be triggerable. Hence, the *States* metric of *recv_ec_send* within PPR(k, m) is $-O(in)$. Note that we add a minus sign to indicate that the *States* metric is a kind of overhead. 2) Metric *CPU Involvement* is the amount of CPU involvement reduced by employing INEC. As illustrated in Figure 2, *recv_ec_send* saves $O(1)$ CPU involvement for each input/output chunk. We summarize the number of saved CPU involvement for all primitives in the table. 3) Metric *DMA* is the amount of DMA traffic saved by coherent in-network primitives. In practice, if there is only one output chunk to be sent, INEC can use the same QP for both EC computation and communication. Such that, SmartNICs can directly send the output chunk after EC operation completes, rather than putting the output chunk in host memory and reading the chunk again from host memory for sending. This optimization is able to reduce $O(c)$ DMA traffic for each output chunk. Further reducing DMA traffic for other cases will be our future work.

From Table I, we can see that the performance of all INEC primitives can be modeled as $\alpha \cdot c + \beta$, referred to as **α-β model**, where α is a function of EC configurations, network bandwidth, and DMA traffic, and β is a function of network latency, in-network states, and CPU involvement. If chunk size c becomes small, the total cost is dominated by network

TABLE I: Primitive Analysis. Note that the `decode` functionality in the table refers to recovering a single erased chunk. c: chunk size, in: number of received chunks, L: network latency, B: network bandwidth.

EC Scheme	Func.	Primitive	Operations	Communication	Gains		
					States	CPU Inv.	DMA
RS(k,m)	encode	ec_send	$O(k \cdot m \cdot c)$	$O(L + (k+m) \cdot c/B)$	$-O(m)$	$O(m)$	0
	decode	recv_ec	$O(k^2 \cdot c)$	$O(L + k \cdot c/B)$	$-O(k)$	$O(k)$	0
LRC(k,l,r)	encode	ec_send	$O(k \cdot (l+r) \cdot c)$	$O(L + (k+l+r) \cdot c/B)$	$-O(l+r)$	$O(l+r)$	0
	decode	recv_ec	$O((k/l)^2 \cdot c)$	$O(L + k \cdot c/(B \cdot l))$	$-O(k/l)$	$O(k/l)$	0
PPR(k,m)	decode	ec_send	$O(c)$	$O(L + c/B)$	$-O(1)$	**O(1)**	$O(c)$
		recv_ec_send	$O(in \cdot c)$	$O(L + in \cdot c/B)$	$-O(in)$	$O(in)$	$O(c)$
		recv_xor	$O(in \cdot c)$	$O(L + in \cdot c/B)$	$-O(in)$	$O(in)$	0
ECPipe(k,m)	decode	ec_send	$O(c)$	$O(L + c/B)$	$-O(1)$	**O(1)**	$O(c)$
		recv_ec_send	$O(c)$	$O(L + c/B)$	$-O(1)$	$O(1)$	$O(c)$
TriEC(k,m)	encode	recv_ec_send	$O(m \cdot c)$	$O(L + m \cdot c/B)$	$-O(m)$	$O(m)$	$O(m \cdot c)$
		recv_xor	$O(k \cdot c)$	$O(L + k \cdot c/B)$	$-O(k)$	$O(k)$	0
	decode	ec_send	$O(c)$	$O(L + c/B)$	$-O(1)$	$O(1)$	$O(c)$
		recv_xor_send	$O(k \cdot c)$	$O(L + k \cdot c/B)$	$-O(k)$	$O(k)$	$O(c)$

latency, number of in-network states, and amount of CPU involvement. On the other hand, if chunk size c is large, both EC computation and network communication dominate the entire procedure. In this scenario, reducing DMA traffic could gain considerable performance improvement.

For large scale EC configurations (i.e., $k+m$ for RS, PPR, ECPipe, and TriEC, and $k+l+r$ for LRC are large), INEC needs to maintain a large number of states on NICs, which possibly influences the on-NIC resource management and thus cuts down the performance of EC computation and/or network communication. However, we find that most widely-used EC configurations [5], [8], [11], [12] are not in very large scale, and our evaluations on these configurations (see Section IV) validate that INEC is able to deliver better performance with commonly-adopted configurations.

C. Implementation and Optimization on RNICs

As aforementioned in Section II-C, RDMA WAIT is an event-driven mechanism on RNICs to support work request (WR) triggering. In this section, we explain the details about implementing INEC primitives with RDMA WAIT.

As clarified in Figure 4a, posting an *ec/xor-send* actually posts an EC/XOR work request (WR) to the calculator's send queue (SQ), and a WAIT WR followed by a SEND WR to each SQ for sending chunks. Multiple WRs to the same SQ are posted by one posting, which reduces PCIe transactions and CPU use [28]. Upon completion of the posted EC/XOR WR, the WAIT WRs in the front of SQs are triggered, and the subsequent SEND WRs are therefore activated and executed by the RNIC without involving the host CPU.

Figure 4b illustrates the detail of *recv-ec/xor-send*. Different from *ec/xor-send*, a post of *recv-ec/xor-send* also posts a number of WAIT WRs to the calculator's SQ, and the number of WAIT WRs is equal to the number of CQs specified by the caller. These WAIT WRs will finally be triggered one by one by receive completions and the last WAIT WR will activate the EC/XOR calculation. The completion of EC/XOR WR will trigger the WAIT WRs in the front of all SQs, thus SEND WRs are activated, and results are sent out. In our implementation, WAIT WRs and the EC/XOR WR to the calculator's SQ is posted by the same PCIe transaction, which further reduces CPU involvement and latency.

For the *recv-ec/xor* primitive, as shown in Figure 4c, several WAIT WRs and an EC/XOR WR are posted into the calculator's SQ by one post and these WAIT WRs will finally be triggered by receive completions of the CQs being waited for. In the end, the EC/XOR WR is activated by the last WAIT WR in front of it.

We implemented INEC primitives within Mellanox OFED driver 4.7-3.2.9.0. Several optimizations are applied to overcome the existing limitations of Mellanox OFED driver. 1) Modifying underneath QPs used by EC calculators to make EC computation to be triggerable, 2) Implementing XOR calculators with the Mellanox Vector CALC capability, which, by current revision, only exposes low-level APIs for bitwise XOR calculation, and 3) Applying doorbell batching technique to post as many WRs as possible in one doorbell to reduce CPU-generated memory mapped I/Os (MMIOs).

D. Dynamic EC Graph

Towards flexibly integrating INEC primitives into state-of-the-art EC schemes, we introduce an abstraction to construct EC schemes and automatically apply INEC primitives. The abstraction is a directed acyclic graph, named *Dynamic EC Graph* (DEG), in which, each vertex represents a node of EC schemes and edges denote data transmission. Figure 5 displays DEGs constructed for PPR$(6,3)$ and TriEC$(3,2)$.

Upper-layer applications construct DEGs in runtime with EC scheme specific layout algorithms (e.g., use PPR layout algorithm to construct PPR's DEG). Once a DEG is constructed, data flow along the edges in the graph, and each

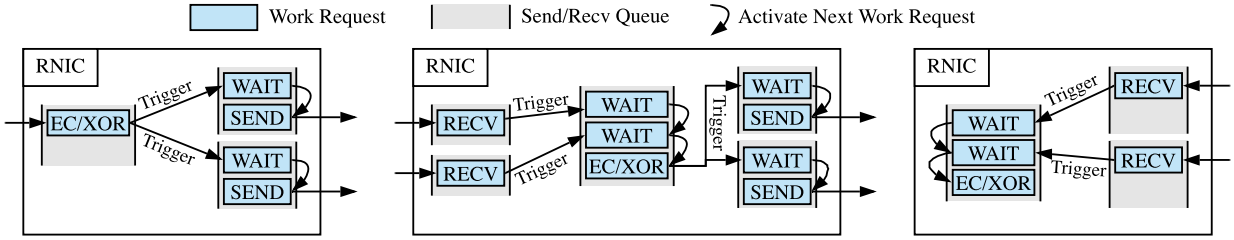


Fig. 4: Design Details of Proposed Primitives

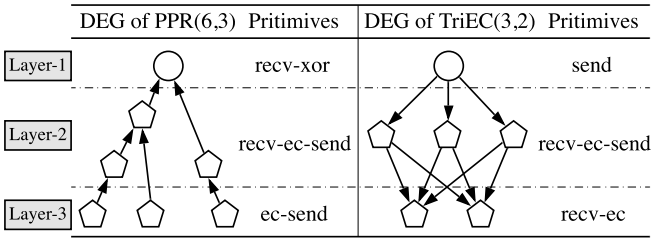


Fig. 5: DEGs of PPR(6,3) and TriEC(3,2)

vertex (i.e., node) makes use of one INEC primitive to join in the computation and data transmission. The basic rules for picking up INEC primitives are: 1) vertices with zero indegree (i.e., Layer-3 of PPR and Layer-1 of TriEC) choose *ec/xor-send* or just *send* if there is no computation, 2) vertices with zero outdegree (i.e., Layer-1 of PPR and Layer-3 of TriEC) leverage *recv-ec/xor*, 3) other vertices (i.e., Layer-2 of PPR and TriEC) utilize *recv-ec/xor-send*, and 4) selecting proper calculator (i.e., EC or XOR calculator) based on EC scheme's requirement. For PPR in Figure 5, the final primitive solution is: *recv-ec/xor* for Layer-1 (root vertex), *recv-ec/xor-send* for Layer-2 (internal vertices), and *ec/xor-send* for Layer-3 (leaf vertices). For the calculator selection, based on different functionality of vertices in PPR's DEG, EC calculators are chosen for primitives *recv-ec/xor-send* and *ec/xor-send* (i.e., for Layer-2 and Layer-3), and XOR calculators are used for the *recv-ec/xor* primitive (i.e., for Layer-1).

E. Co-Design Case Study: INEC-Cache

In this section, we implement a key-value store, named INEC-Cache, based on `memcached` (v1.5.12). The implementation demonstrates the performance implication of integrating state-of-the-art EC schemes and employing INEC primitives on storage systems.

The design of INEC-Cache adopts an interleaved architecture, which is similar to the architectures of Cocytus [29] and TriEC-Cache [18]. There are three kinds of INEC-Cache nodes, i.e., *Agent* node, *Data Cache* node, and *Parity Cache* node. An Agent node and multiple Data Cache and Parity Cache nodes form a *Mem Stripe*, such that each EC stripe is processed, stored, and repaired within a Mem Stripe. Meanwhile, data chunks in an EC stripe are finally stored onto Data Cache nodes, and parity chunks onto Parity Cache nodes. The Agent node in each Mem Stripe processes clients' requests, dispatches control messages and data to Data Cache and Parity Cache nodes in the same Mem Stripe, and responds to the clients. Mem Stripes are arranged in a circularly interleaved manner across the cluster to balance the use of CPU, memory, and network resources.

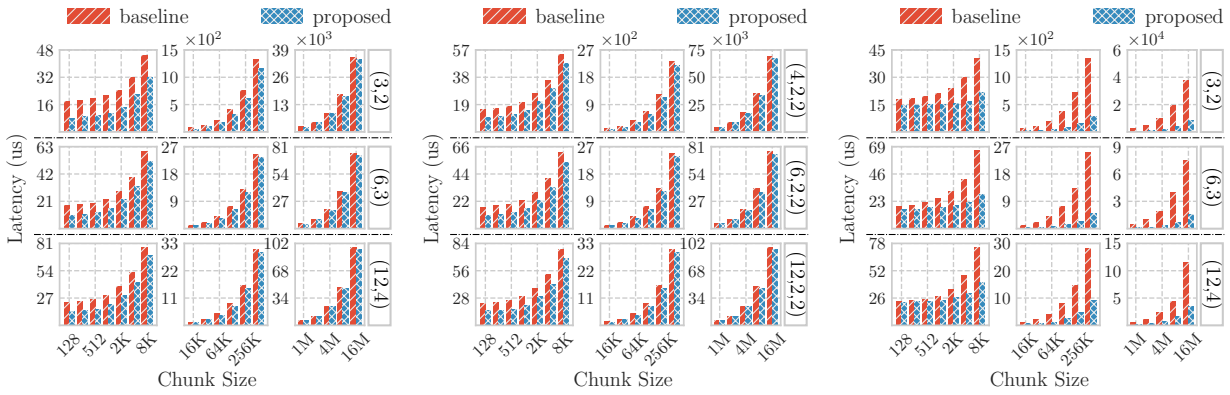
Agent node issues control messages to notify Data Cache and Parity Cache nodes to construct a DEG on demand. Each Data Cache and Parity Cache node decides *whether* and *how* to join in constructing the DEG on 1) local rank, and the rank(s) of the chunk(s) to be reconstructed, 2) DEG topology information and 3) DEG constructing algorithm of currently employed EC scheme. For instance, DEGs for writes and normal reads of all EC schemes in Figure 1 have to be constructed when INEC-Cache is up since the required DEGs for writes and normal reads do not change through all the time. However, DEGs for repairing might be different because the chunk(s) to be recovered would change over time. Moreover, repairing DEGs for various EC schemes are diverse as well, since they adopt dissimilar repair structures.

INEC-Cache maintains metadata (e.g., local rank, topology information) separately from the data with replication scheme for efficient access. When node failures happen, INEC-Cache takes different actions for different situations. When an Agent node fails, the Mem Stripe it belongs to becomes unreachable until a new Agent node joins in and becomes recovered. If Data Cache and/or Parity Cache nodes fail, and the number of failed nodes is no more than the max number tolerated by the employed EC scheme, the Mem Stripe becomes read-only until other nodes join in with reconstructed chunks. The failure recovery process is carried out on the server side, which is transparent to upper-layer applications.

IV. Evaluation

In this section, we evaluate five state-of-the-art EC schemes implemented with the proposed INEC primitives to illustrate performance benefits brought by INEC. We choose a custom benchmark suite [30] and Yahoo Cloud Serving Benchmark (YCSB) [31] to evaluate the performance of INEC primitives and its co-designed key-value store (i.e., INEC-Cache). The baseline of all experiments in this section refers to the same implementation yet with incoherent EC calculation and networking approach, i.e., posting *send*, posting *ec/xor*, and polling for completion separately.

The experiments are conducted on a cluster with up to 17 nodes. Each node is equipped with Intel Broadwell E5-



(a) RS **(b)** LRC **(c)** TriEC

Fig. 6: Encoding Latency. Note different scales on Y axes. (k, m) (e.g., $(3, 2)$) and (k, l, r) (e.g., $(4, 2, 2)$) refer to EC configurations.

2680 v4 (2.4 GHz) CPUs, 128GB memory, and CentOS 7.4. The cluster is interconnected with Mellanox ConnectX-5 IB-EDR (100 Gbps) RNICs; thus, Mellanox OFED driver is also installed on each node, the OFED we use for this paper is modified based on $v4.7-3.2.9.0$ (to support INEC primitives).

For simplicity and clarity, we use the same methodology as we use in Figure 1 to describe EC schemes, i.e., Layer-1 and Layer-3 refer to the leftmost and rightmost layers, respectively, and Layer-2 refers to the nodes in between. We also utilize the α-β performance model in Section III-B to explain observations.

A. Microbenchmark

We evaluate the performance of EC schemes implemented with INEC primitives by a custom benchmark suite [30], which includes a latency benchmark and a bandwidth benchmark. The latency benchmark consists of two parts, 1) encoding benchmark, which encodes a file filled with random strings to generate coded blocks, and 2) decoding benchmark, in which each EC scheme reconstructs the original file from coded blocks while some blocks are erased according to the configuration. Upon completion, the latency benchmark reports the latencies of encoding/decoding operations. On the other hand, the bandwidth benchmark is designed similarly as the traditional window-based messaging benchmarks for network bandwidth measurement. In the bandwidth benchmark, $\text{bandwidth} = (\text{size of generated data})/(\text{elapsed time})$. For instance, the bandwidth of encoding a file with $RS(k, m)$ is $(m \times \text{filesize})/(k \times \text{time})$, while the bandwidth of decoding on survived coded blocks to recover an erased block with $RS(k, m)$ is $\text{filesize}/(k \times \text{time})$. The objective of this bandwidth benchmark is to determine the maximum sustained EC rate that can be achieved.

Three widely-used EC configurations (i.e., $(3, 2)$, $(6, 3)$, and $(12, 4)$) are chosen for EC schemes except for LRC. For the case of LRC, the selected configurations are LRC$(4, 2, 2)$, LRC$(6, 2, 2)$ and LRC$(12, 2, 2)$, which are used in Microsoft Azure [11]. We split involved chunk sizes into three groups (i.e., 128B to 8KB, 16KB to 512KB, and 1MB to 16MB), which are mapped to small, medium, and large chunk sizes, separately, for illustrating performance characteristics.

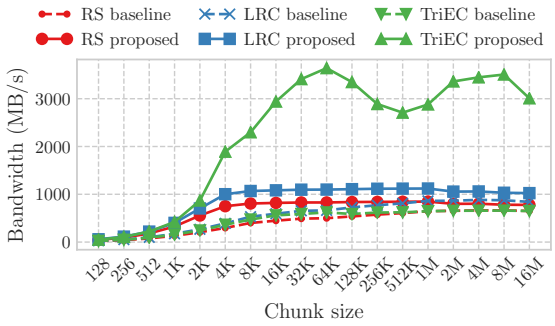


Fig. 7: Encoding Bandwidth of RS(6,3), LRC(6,2,2), and TriEC(6,3)

1) Encoding Latency and Bandwidth: There are two typical structures in organizing involved nodes for performing encoding in the state-of-the-art EC schemes. One structure adopted by RS and LRC arranges nodes to form a bipartite graph, such that the structure can be split into two layers, i.e., Layer-1 and Layer-2 of RS and LRC in Figure 1. Layer-1 encodes on data chunks and then sends chunks to Layer-2, while peers on Layer-2 receive chunks from Layer-1. Our evaluations, as shown in Figures 6a and 6b, reveal that the integration of coherent *ec-send* significantly reduces the encoding latencies of EC schemes RS and LRC for small and medium chunk sizes, but does not help much for large chunk sizes. The observation can be explained by the α-β model in Section III-B. As illustrated in Section III-B, the proposed coherent in-network primitive does not reduce DMA traffic of *ec-send* used in RS; therefore, α_{baseline} and α_{INEC} are theoretically close and similar. As a consequence, latencies of the baseline and INEC are also similar when c is large. The other structure to conduct encoding is a tripartite graph based structure (i.e., the structure employed in TriEC), in which Layer-1 sends out data chunks, Layer-2 encodes on received data chunks and sends intermediate chunks to Layer-3, and Layer-3 XORs on the intermediate chunks from Layer-2 to generate parity chunks. Our integration applies

ec-send to Layer-1, *recv-ec-send* to Layer-2, and *recv-xor* to Layer-3. Figure 6c depicts that INEC primitives deliver significant performance benefits to TriEC for small, medium, and even large chunk sizes, since the primitives utilized in TriEC calculators reduce unnecessary DMA traffic as revealed in Table I. We do not include PPR and ECPipe in this section since they are specifically designed for decoding.

The results in Figures 6 cover three EC configurations of each EC scheme with various chunk sizes. INEC improves RS(3,2) by 29.29%–56.12% for small chunk sizes, 12.65%–28.69% for medium chunk sizes, and performs on par with the baseline for large chunk sizes. LRC(6,2,2) is improved by INEC by up to 36.63% for small chunks, and up to 16.20% for medium chunks. In the meantime, the achieved speedup for TriEC(12,4) is up to 1.81×, 3.36×, and 3.32× for small, medium, and large chunk sizes, respectively. Figure 7 illustrates that INEC is able to speed up the bandwidth of RS(6,3) by up to 2.71×, LRC(6,2,2) by up to 2.63×, and TriEC(6,3) by up to 5.87×. The performance fluctuation of TriEC(6,3) (in Figure 7) in the range of 128KB to 2MB is caused by the policy change of buffer management in our RDMA runtime. Overall, TriEC with INEC primitives achieves the best bandwidth performance, which demonstrates that the tripartite graph structure is more friendly to take advantage of coherent in-network primitives.

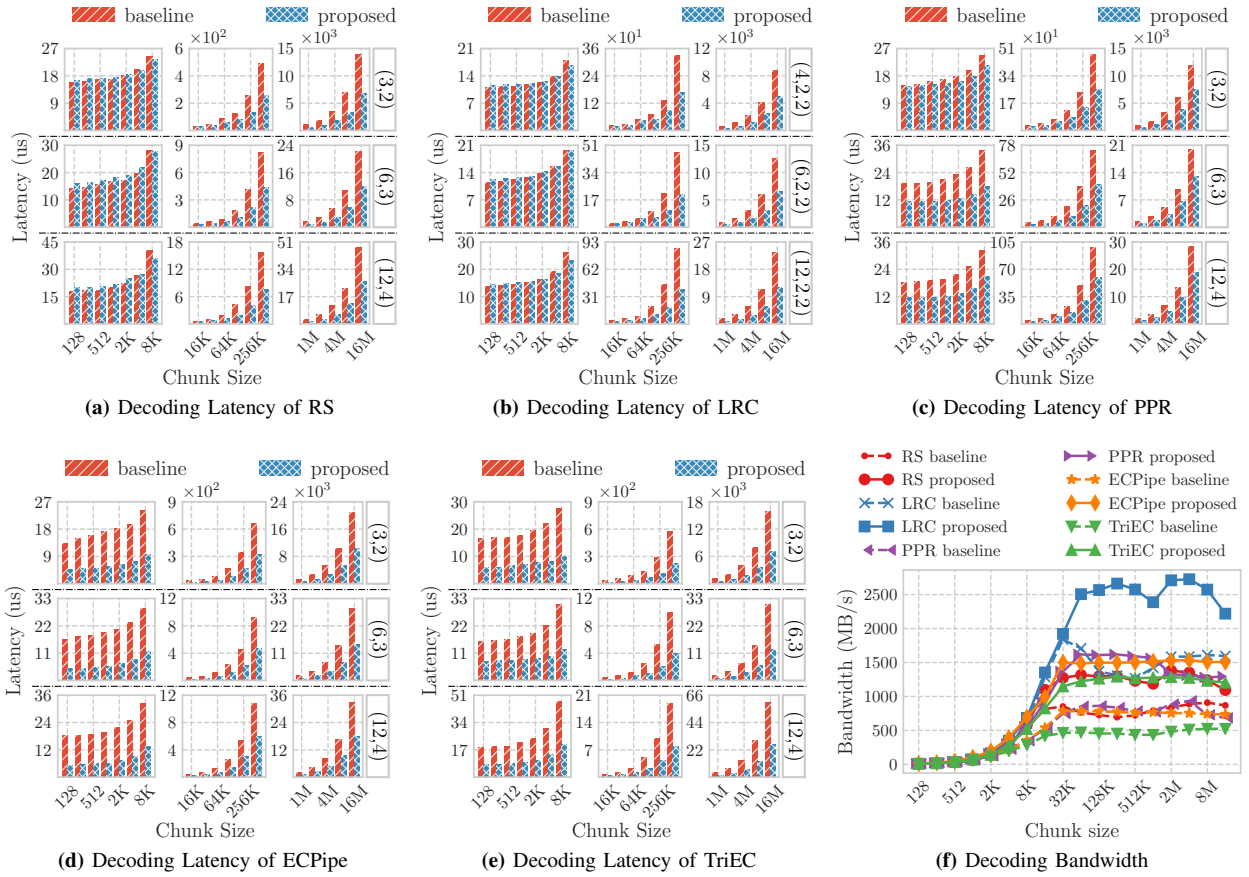
2) Decoding Latency and Bandwidth: Reducing decoding latency is a significant focus in many prior studies [11], [16]–[18]. One direction along this line is to decrease the number of chunks for repairing. LRC introduces a local parity for each EC group for fast recovery in case of single chunk failure in the group. Nevertheless, LRC still maintains the receive and decode pattern which RS holds. Thereby, *recv-ec* is used to accelerate the receive and decode process in both RS and LRC in our experiments. On the contrary, most EC schemes (e.g., PPR, ECPipe, TriEC), which are optimized for repairing, decompose decoding computation into subproblems and schedule them to all involved nodes to parallel computation and balance resource utilization. These EC schemes have deep repair structures, and the involved nodes can typically be categorized into three layers based on their different functionality (i.e., Layer-1, Layer-2, and Layer-3). INEC can be easily integrated into all three layers.

We evaluate the decoding latency for repairing a single chunk. Figures 8a and 8b indicate that *recv-ec* performs significantly better except for small chunk sizes. Performance numbers show that *recv-ec* accelerates RS(12,4) by 25.57%–54.01% for chunk sizes ranging from 16KB to 16MB, and LRC(12,2,2) by 1.34×–2.36× and 1.94×–2.27× for medium and large chunk sizes, respectively. We also use the α-β performance model in Section III-B to explain why *recv-ec* does not perform better for small chunks. If chunk size c is small, β possibly dominates in the performance model. Since β is a function of network latency, number of in-network states, and amount of CPU involvement, these factors, especially in-network states and CPU involvement, play an important role to determine the coherent primitives' performance benefits for small chunk sizes. In the latency benchmark, the CPU usage keeps relatively low in the case of small chunk sizes, and CPU involvement is not a performance bottleneck of the incoherent baseline; therefore, the extra in-network states introduced by *recv-ec* degrades its latency performance. However, in practice, there are other computation and communication workloads in HPC clusters and multi-tenant data centers, which may incur high CPU utilization and many context switches. Based on our analysis, INEC has the potentials to outperform the incoherent baseline for small chunks in these environments.

Figure 8c reveals that INEC accelerates PPR by up to 1.86×, 2.05×, and 1.75× for configurations (3,2), (6,3), and (12,4), respectively. In the evaluation of ECPipe, the internal slice size is fixed to 32KB, such that decoding is carried out in the units of 32KB or actual chunk size for chunk sizes less than 32KB. The results in Figure 8d reveal that the use of INEC speeds up ECPipe for all evaluated chunk sizes. For chunk sizes from 128B to 16MB, INEC obtains 49.82%–67.82% for (3,2), 49.47%–72.73% for (6,3), and 42.40%–72.25% for (12,4), respectively, compared with the baseline approach. Figure 8e illustrates the performance gains obtained by TriEC with the proposed primitives. For (3,2), (6,3), and (12,4), TriEC with INEC reduces latencies by up to 64.29%, 61.84%, and 59.11%, respectively. From Figure 8f, we can see that INEC outperforms the baseline in terms of bandwidth by up to 1.86×, 2.04×, 2.03×, 2.04×, and 2.94× for RS, LRC, PPR, ECPipe, and TriEC, respectively. LRC with INEC gains the best bandwidth performance, because of the fact that LRC, which has the assistance of local parities, requires fewer chunks for reconstructing a single chunk. Thus, less EC computation and communication in LRC incur higher bandwidth. Another fact we would like to point out is that TriEC has the ability to perform in-band recovery [18], thus upper-layer applications could benefit more from TriEC than other EC schemes, since they need to perform extra computation and communication to conduct out-of-band recoveries.

B. Yahoo Cloud Serving Benchmark (YCSB)

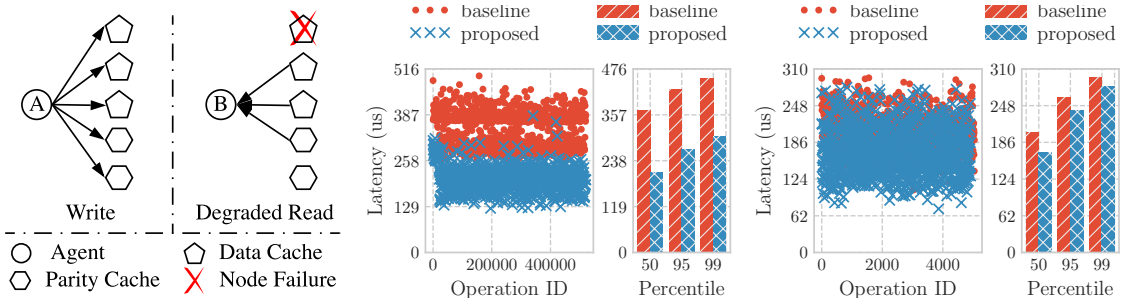
In this section, we evaluate the performance of INEC-Cache with YCSB. Each workload generated by YCSB consists of a well-defined combination of operations (e.g., read, write), data sizes, request distributions, etc. We modify YCSB to support degraded read, i.e., read with data repair. In distributed systems, failures are not always independent. In practice, *Weibull* distribution with shape parameter of 0.7–0.8 provides a close fit for predicting failures on HPC systems [32]–[35]. Therefore, our modified YCSB generates degraded read operations based on the Weibull distribution. The ratio of degraded read to normal read is set to 1% for our evaluations, unless explicitly stated otherwise. The request distribution is *Zipfian* [36] distribution, which simulates the situation that some portion of records is extremely popular for being requested. We choose different workloads (i.e., Workload-A (50% reads, 50% writes), Workload-B (95% reads, 5% writes), and Workload-C (100% reads)), value sizes, and EC configurations for INEC-Cache with varied EC schemes.



(a) Decoding Latency of RS **(b)** Decoding Latency of LRC **(c)** Decoding Latency of PPR

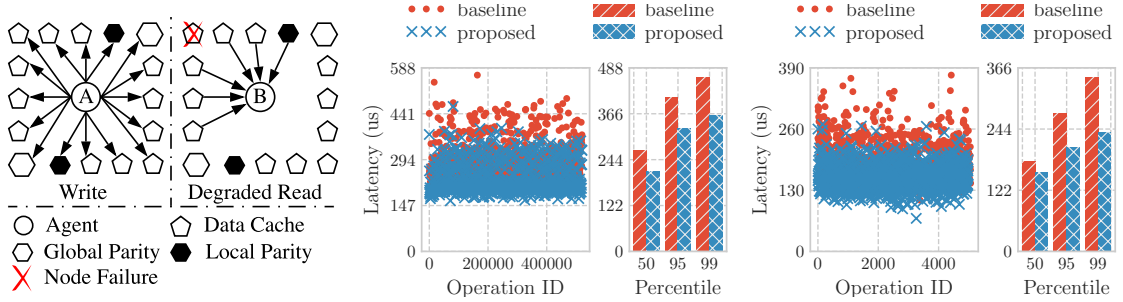
(d) Decoding Latency of ECPipe **(e)** Decoding Latency of TriEC **(f)** Decoding Bandwidth

Fig. 8: Decoding Latency and Bandwidth. Note different scales on Y axes. (k, m) (e.g., $(3, 2)$) and (k, l, r) (e.g., $(4, 2, 2)$) refer to EC configurations. Decoding bandwidth benchmarks are conducted with RS(6,3), LRC(6,2,2), PPR(6,3), ECPipe(6,3), and TriEC(6,3).



(a) RS DEGs **(b)** ec-send of Node A **(c)** recv-ec of Node B

Fig. 9: INEC-Cache with RS(3,2). YCSB Workload-A with 4KB value size



(a) LRC DEGs **(b)** ec-send of Node A **(c)** recv-ec of Node B

Fig. 10: INEC-Cache with LRC(12,2,2). YCSB Workload-A with 32KB value size

1) INEC-Cache with RS(3,2): Figure 9a shows the encoding and decoding DEGs constructed for write and degraded read. Node A in the encoding DEG is applied with the *ec-send* primitive, and node B in the decoding DEG employs the *recv-ec* primitive. The evaluation is conducted with YCSB *Workload-A*, in which value size is fixed to 4KB. Figure 9b reveals *ec-send*'s latency distribution and three major percentile latencies (i.e., 50^{th}, 95^{th}, and 99^{th} percentile latencies). The distribution clearly illustrates that *ec-send* performs better than the baseline, since the use of *ec-send* reduces CPU involvement and context switches, and alleviates compute intensity. As a result, *ec-send* cuts down 50^{th}, 95^{th}, and 99^{th} percentile latencies by 44.00%, 36.82%, and 32.99%, respectively. Figure 9c depicts the improvement gained by *recv-ec*. Since chunk size is relatively small, *recv-ec* does not deliver as much improvement as *ec-send*. In this evaluation, *recv-ec* speeds up 50^{th}, 95^{th}, and 99^{th} percentile latencies by 1.20×, 1.09×, and 1.05×, respectively.

2) INEC-Cache with LRC(12,2,2): The evaluation is performed with YCSB Workload-A with the value size of 32KB. The constructed encoding and decoding DEGs for LRC are shown in Figure 10a. Similar to the integration of RS, *ec-send* is applied to node A, and *recv-ec* is used by node B. The latency distributions in Figures 10b and 10c demonstrate that using coherent primitives can alleviate the performance fluctuation introduced by frequent context switches and thereby provide relatively stable latency performance. Overall, *ec-send* speeds up 50^{th} percentile latency by 1.27×, 95^{th} percentile latency by 1.26×, and 99^{th} percentile latency by 1.28×, while *recv-ec* reduces 50^{th}, 95^{th}, and 99^{th} percentile latencies by 12.19%, 24.64% and 31.48%, respectively.

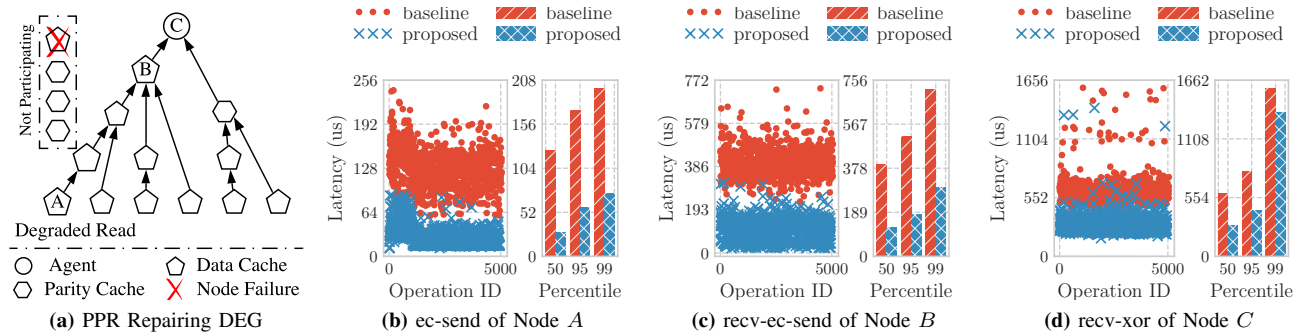
3) INEC-Cache with PPR(12,4): PPR is an advanced EC scheme designed for chunk repairing but without encoding optimizations, so we only evaluate PPR's decoding performance in this benchmark. We evaluate INEC-Cache with PPR(12,4) on YCSB Workload-C, and the value size is fixed to 64KB. As illustrated in Figure 11a, the repairing DEG is a tree-like structure, and we apply *ec-send* to node A, *recv-ec-send* to node B, and *recv-xor* to node C. Figure 11b reveals that, compared with the baseline, *ec-send* delivers better and more stable latency performance. It achieves 77.16%, 66.79%, and 62.40% performance improvement for 50^{th}, 95^{th}, and 99^{th} percentile latencies, respectively. Figure 11c depicts that, even in the most complicated place (i.e., node B), *recv-ec-send* is still able to offer significant speedup. Compared with the baseline approach, 50^{th}, 95^{th}, and 99^{th} percentile latencies are cut down by 69.03%, 65.32%, and 58.73%, respectively. Node C in the decoding DEG finally performs a *recv-xor* to generate the reconstructed chunk. Since it is the end of the entire reduction tree, both the baseline and the proposed approaches encounter some significant outliers and thus incur high 99^{th} percentile latencies as shown in Figure 11d. Our experiments show that *recv-xor* speeds up 50^{th}, 95^{th}, and 99^{th} percentile latencies by 2.05×, 1.86×, and 1.17×, respectively.

4) INEC-Cache with ECPipe(6,3): ECPipe is specifically designed for chunk reconstruction as well, so we use INEC-accelerated RS(6,3) for performing encoding operations and ECPipe(6,3) with INEC for decoding. In this evaluation, the internal slice size of ECPipe is set to 1KB, and we use YCSB Workload-B with 16KB value size. In order to study different primitives involved in ECPipe, we select three nodes along the pipeline, i.e., nodes A, B, and C in Figure 12a. As shown in Figure 12b, the *ec-send* primitive of node A outperforms the baseline by 5.47×, 4.25×, and 3.54× for 50^{th}, 95^{th}, and 99^{th} percentile latencies, respectively. The *recv-ec-send* of node B, as in Figure 12c, gains 48.47% for 50^{th} percentile latency, 50.51% for 95^{th} percentile latency, and 47.86% for 99^{th} percentile latency. Node C is the end node of ECPipe and it only receives slices from the downstream node. Figure 12d demonstrates that node C with coherent in-network primitives outperforms the baseline dramatically as well (i.e., by 68.73%, 69.29%, and 68.45% for 50^{th}, 95^{th}, and 99^{th} percentile latencies, respectively).

5) INEC-Cache with TriEC(3,2): The evaluation of TriEC is carried out with YCSB Workload-A with the value size of 8KB. Note that each degraded read needs to repair two chunks in this experiment. As illustrated in Figure 13a, we pick up three nodes to clarify INEC's benefits in accelerating encoding and decoding processes. Figure 13b shows that node A, as one of the major computing nodes in encoding process, obtains low and stable latencies by the apply of *recv-ec-send*. The results reveal that *recv-ec-send* performs 5.87×, 4.21×, and 3.60× better than the baseline for 50^{th}, 95^{th}, and 99^{th} percentile latencies, respectively. Node B in the encoding structure receives intermediate chunks from Layer-2 and XORs on these intermediate chunks to construct parity chunks. As illustrated in Figure 13c, the employment of *recv-xor* gains 39.59%, 31.96%, and 31.62% performance improvement for 50^{th}, 95^{th}, and 99^{th} percentile latencies, respectively. Figure 13d represents that, by leveraging the *recv-xor-send* primitive, node C (i.e., the node to be reconstructed) reduces 50^{th} percentile latency by 81.27%, 95^{th} percentile latency by 50.17%, and 99^{th} percentile latency by 47.19%. As we can see, the proposed coherent in-network primitives significantly accelerate reconstructions.

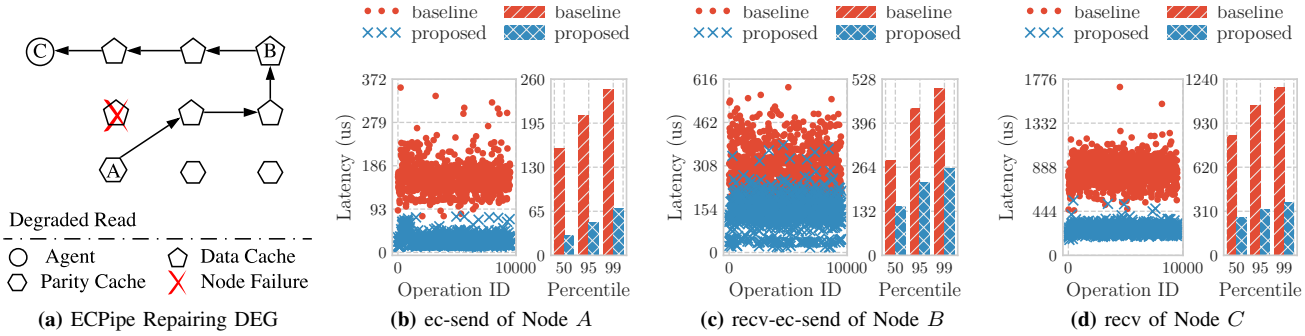
6) Overall Performance Improvement of INEC-Cache: In this section, we evaluate the overall performance of INEC-Cache with comparable EC configurations, i.e., RS(6,3), LRC(6,2,2), PPR(6,3), ECPipe(6,3), and TriEC(6,3). The chosen YCSB workloads are Workloads A, B, and C, in which read requests are all degraded reads (since normal reads do not use INEC primitives), and value size is fixed to about 8KB, which is a representative value size in Facebook's memcached cluster [37]. All the experiments are conducted with 144 YCSB clients and 9 INEC-Cache Mem Stripes for RS, PPR, ECPipe, and TriEC, and 160 YCSB clients and 10 INEC-Cache Mem Stripes for LRC. Since PPR and ECPipe only support decoding, we use RS encoding together with PPR and ECPipe to complete the experiments in this section. Figure 14 presents the improvement ratios with respect to the baseline.

For RS(6,3) and LRC(6,2,2), since the chunk size in this evaluation falls into the range of small chunk sizes, INEC



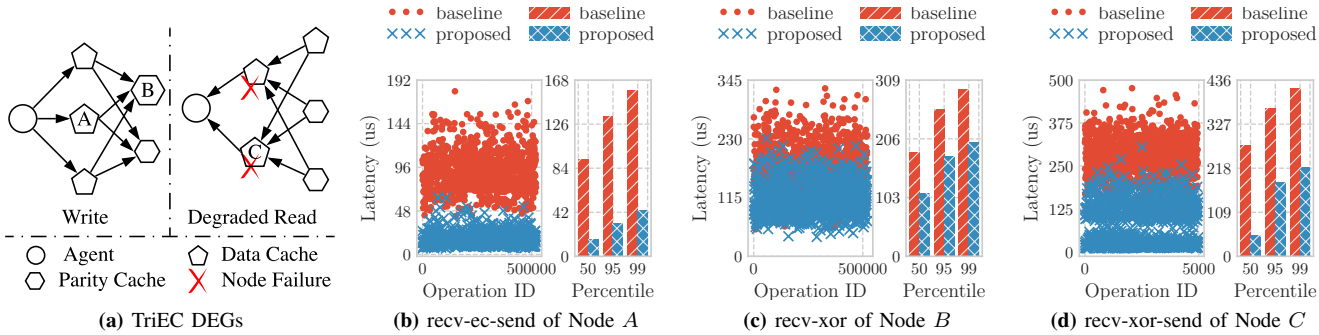
(a) PPR Repairing DEG **(b)** ec-send of Node A **(c)** recv-ec-send of Node B **(d)** recv-xor of Node C

Fig. 11: INEC-Cache with PPR(12,4). YCSB Workload-C with 64KB value size



(a) ECPipe Repairing DEG **(b)** ec-send of Node A **(c)** recv-ec-send of Node B **(d)** recv of Node C

Fig. 12: INEC-Cache with ECPipe(6,3). YCSB Workload-B with 16KB value size



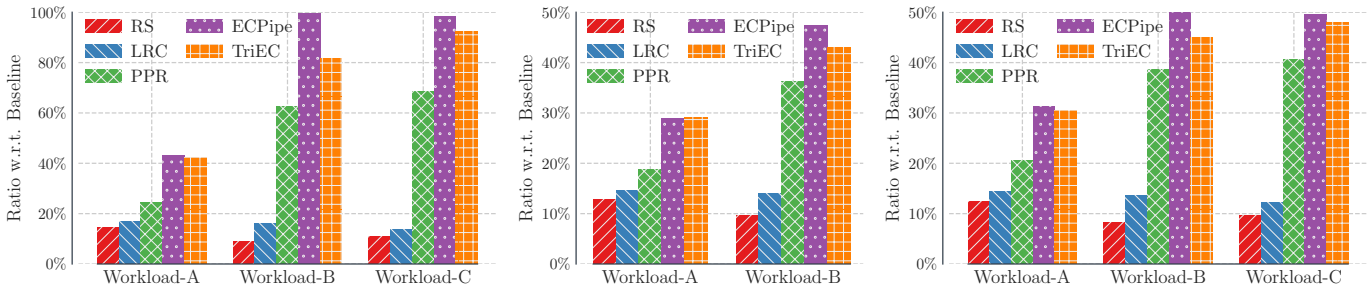
(a) TriEC DEGs **(b)** recv-ec-send of Node A **(c)** recv-xor of Node B **(d)** recv-xor-send of Node C

Fig. 13: INEC-Cache with TriEC(3,2). YCSB Workload-A with 8KB value size

accelerates encoding operations but does not help much in decoding operations. Therefore, among Workloads A, B, and C, RS and LRC with INEC achieve the most improvement with Workload-A, which is a write-heavy workload compared with the other two. As shown in Figure 14, RS with INEC outperforms the baseline by 14.36%, 12.85%, and 12.48% for throughput, write latency, and degraded read latency, respectively, with Workload-A. On the other hand, LRC with INEC on Workload-A improves overall throughput by 16.99%, write latency by 14.68%, and degraded read latency by 14.40%.

Our microbenchmarks in Sections IV-A1 and IV-A2 illustrate that the decoding performance gains of INEC-based PPR, ECPipe, and TriEC are larger than their encoding performance gains for small chunks. Therefore, INEC-based PPR, ECPipe, and TriEC perform better with read-dominated workloads (i.e., Workloads B and C) than with write-heavy workloads such as Workload-A. As we can see from Figure 14, INEC-based PPR obtains improvement by up to 68.33% for throughput, 36.33% for write latency, and 40.60% for degraded read latency. ECPipe with INEC speeds up throughput performance by up to 99.57%, write latency by up to 47.30%, and degraded read latency by up to 49.92%. INEC-accelerated TriEC improves throughput, write latency, and degrade read latency by up to 92.26%, 42.91%, and 48.04%, respectively.

To summary, compared with the incoherent baseline, INEC-Cache with varied INEC-based EC schemes presents an average performance boost of up to 57% for throughput with read-dominated workloads and an average speedup of 28% for throughput with write-heavy workloads. The evaluation results demonstrate that the proposed coherent in-network primitives (i.e., INEC), are able to deliver high-performance EC computation to real-world applications.



(a) YCSB Throughput **(b)** YCSB Write Latency **(c)** YCSB Degraded Read Latency

Fig. 14: Overall Performance Improvement of INEC-Cache.

C. Discussion

Several insights from these experiments are worthwhile to discuss further. For small and extremely large chunk sizes in some experiments, offloading computation and communication in the same post (i.e., INEC) performs on par or a little worse than offloading them separately (i.e., baseline), such that a hybrid design utilizing both INEC and incoherent primitives may deliver the optimal performance by two reasons: 1) INEC primitives are compatible with the conventional incoherent primitives, so a hybrid design is feasible, and 2) INEC needs to put some RDMA WAITs on RNICs to construct coherent in-network primitives, while maintaining a large amount of RDMA WAITs may adversely impact on-NIC resource management and thus incur performance degradation. A hybrid design can reduce the number of states to be maintained on SmartNICs and make proper use of available resources.

Some experiments also reveal that EC calculation on extremely large chunk sizes might exhaust the computing power of RNICs, which would incur performance loss in computation and communication for both INEC and incoherent primitives. This insight indicates that we would better choose a refined chunk size for EC offload based storage systems to fully take advantage of the EC offload capability on SmartNICs. Overall, our experiment results clearly show that the benefits of INEC can be attained in most cases. These insights observed in the experiments validate our analysis in Section III-B.

V. Related Work

The capability of delivering higher data reliability with lower storage overhead [38] makes EC be a promising alternative to replication. Many existing storage systems [5]–[8], [10], [12], [13] have adopted EC in their implementations to replace or to be a complement to replication. A rich body of work focuses on accelerating EC to make it viable for large scale storage systems. Local Reconstruction Codes [11] and Locally Repairable Codes [39] facilitate more efficient recovery scenarios by storing additional *local parities*, which can reduce the number of chunks to be read for data reconstruction. PPR [16], ECPipe [17], and TriEC [18] decompose EC calculation into sub calculation and schedule them to multiple nodes to achieve better parallelism, overlapping, and balanced resource utilization. OpenEC [40] and Fast Erasure Coding [41] have proposed frameworks to manage efficiently and optimize EC systematically. On the other hand, the increased focus on EC for storage resilience has motivated several works to apply EC for Big Data and Cloud storage systems as well as key-value store systems (e.g., [29], [32], [42]–[53]).

The emergence of next-generation HPC or data center hardware platforms has inspired several prior studies to design network-based resilient schemes. HyperLoop [23] offloads replication transactions to RDMA NICs to dramatically reduce the 99^{th} percentile latency of a multi-tenant non-volatile memory based storage systems. TriEC [18] has proposed a new EC NIC offload paradigm which overcomes some major limitations of current-generation EC NIC offload schemes on modern SmartNICs. This emerging focus on network-based resilient schemes serves as a motivation for this paper.

VI. Conclusion and Future Work

In this paper, we proposed a set of coherent in-network EC primitives called INEC, which can leverage the event-driven work request triggering mechanism available on modern SmartNICs to make EC calculation and networking functionality to be coherent. In the meantime, we also proposed a performance model (i.e., α-β model) to analyze the performance gains of INEC primitives in multiple state-of-the-art EC schemes. Our analyses reveal that INEC enables state-of-the-art EC schemes to take full advantage of the EC offload capability on modern SmartNICs. The performance gains of INEC are validated with five state-of-the-art EC schemes. YCSB-based evaluations on the key-value store co-designed with INEC (i.e., INEC-Cache) show that INEC significantly reduces percentile latencies and improves the end-to-end throughput, write, and degraded read performance by up to 99.57%, 47.30%, and 49.55%, respectively. The evaluations validate that the integration of INEC primitives into state-of-the-art EC schemes is feasible and delivers optimized performance. Cross comparisons based on the evaluation results illustrate that INEC-accelerated TriEC significantly outperforms other EC schemes in terms of encoding bandwidth, while LRC with INEC attains the best decoding bandwidth performance. In the future, we plan to propose designs for INEC primitives on more types of commodity SmartNICs and examine the benefits of INEC with more applications.

REFERENCES

[1] F. Herold, S. Breuner, and J. Heichler, "An introduction to BeeGFS," https://www.beegfs.io/docs/whitepapers/Introduction_to_BeeGFS_by_ThinkParQ.pdf, 2014 (Accessed on 2020-08-31).

[2] "GlusterFS," https://www.gluster.org/, 2020 (Accessed on 2020-08-31).

[3] "OrangeFS," http://www.orangefs.org/, 2020 (Accessed on 2020-08-31).

[4] I. S. Reed and G. Solomon, "Polynomial Codes Over Certain Finite Fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[5] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly, "The Quantcast File System," *Proceedings of the VLDB Endowment*, no. 11, pp. 1092–1101, 2013.

[6] Ceph, "Erasure code," https://docs.ceph.com/docs/master/rados/operations/erasure-code/, 2016 (Accessed on 2020-04-17).

[7] C. Lai, S. Jiang, L. Yang, S. Lin, G. Sun, Z. Hou, C. Cui, and J. Cong, "Atlas: Baidu's key-value storage system for cloud data," in *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, 2015, pp. 1–14.

[8] A. Hadoop, "Apache Hadoop 3.0.0," http://hadoop.apache.org/docs/r3.0.0/, 2017 (Accessed on 2020-04-17).

[9] Facebook, "Facebook's Erasure Coded Hadoop Distributed File System (HDFS-RAID)," https://github.com/facebookarchive/hadoop-20, 2010 (Accessed on 2020-04-17).

[10] Backblaze, "Backblaze Reed-Solomon," https://www.backblaze.com/open-source-reed-solomon.html, 2015 (Accessed on 2020-04-17).

[11] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, S. Yekhanin *et al.*, "Erasure Coding in Windows Azure Storage," in *Usenix Annual Technical Conference*. Boston, MA, 2012, pp. 15–26.

[12] Google, "Colossus: Successor to the Google File System (GFS)," https://www.systutorials.com/3202/colossus-successor-to-google-file-system-gfs/, 2012 (Accessed on 2020-04-17).

[13] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang *et al.*, "f4: Facebook's Warm BLOB Storage System," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, 2014, pp. 383–398.

[14] Lustre, "File Level Redundancy Solution Architecture," http://wiki.lustre.org/File_Level_Redundancy_Solution_Architecture, 2019 (Accessed on 2020-08-28).

[15] "Introduction to IBM Spectrum Scale Erasure Code Edition," https://www.ibm.com/support/knowledgecenter/STXKQY_ECE_5.0.5/com.ibm.spectrum.scale.ece.v5r05.doc/b1lece_intro.htm, 2020 (Accessed on 2020-08-31).

[16] S. Mitra, R. Panta, M.-R. Ra, and S. Bagchi, "Partial-Parallel-Repair (PPR): A Distributed Technique for Repairing Erasure Coded Storage," in *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 2016, p. 30.

[17] R. Li, X. Li, P. P. Lee, and Q. Huang, "Repair Pipelining for Erasure-coded Storage," in *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC'17)*, 2017, pp. 567–579.

[18] H. Shi and X. Lu, "TriEC: Tripartite Graph Based Erasure Coding NIC Offload," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019.

[19] Intel, "Intel Intelligent Storage Acceleration Library (Intel ISA-L)," https://software.intel.com/en-us/storage/ISA-L, 2016 (Accessed on 2020-04-17).

[20] M. Curry, A. Skjellum, H. Lee Ward, and R. Brightwell, "Gibraltar: A Reed-Solomon Coding Library for Storage Applications on Programmable Graphics Processors," in *Concurrency and Computation: Practice and Experience*, vol. 23, 12 2011, pp. 2477–2495.

[21] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, "A Solution to the Network Challenges of Data Recovery in Erasure-coded Distributed Storage Systems: A Study on the Facebook Warehouse Cluster." in *HotStorage*, 2013.

[22] Mellanox, "Understanding Erasure Coding Offload," https://community.mellanox.com/docs/DOC-2414, 2018 (Accessed on 2020-04-17).

[23] D. Kim, A. Memaripour, A. Badam, Y. Zhu, H. H. Liu, J. Padhye, S. Raindel, S. Swanson, V. Sekar, and S. Seshan, "Hyperloop: Group-Based NIC-Offloading to Accelerate Replicated Transactions in Multi-Tenant Storage Systems," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 297–312.

[24] R. Recio, B. Metzler, P. Culley, J. Hilland, and D. Garcia, "A Remote Direct Memory Access Protocol Specification," RFC 5040, October, Tech. Rep., 2007.

[25] Mellanox, "CORE-Direct," https://www.mellanox.com/related-docs/whitepapers/TB_CORE-Direct.pdf, 2010 (Accessed on 2020-04-17).

[26] Mellanox, "Cross Channel," https://docs.mellanox.com/display/rdmacore50/Cross+Channel, 2010 (Accessed on 2020-04-17).

[27] D. R. K. Ports and J. Nelson, "When Should The Network Be The Computer?" in *Proceedings of the Workshop on Hot Topics in Operating Systems*, ser. HotOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 209–215.

[28] A. Kalia, M. Kaminsky, and D. G. Andersen, "Design Guidelines for High Performance RDMA Systems," in *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '16. USA: USENIX Association, 2016, p. 437–450.

[29] H. Zhang, M. Dong, and H. Chen, "Efficient and Available In-memory KV-Store with Hybrid Erasure Coding and Replication," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*. Santa Clara, CA: USENIX Association, Feb. 2016, pp. 167–180.

[30] H. Shi, X. Lu, and D. K. Panda, "EC-Bench: Benchmarking Onload and Offload Erasure Coders on Modern Hardware Architectures," in *International Symposium on Benchmarking, Measuring and Optimization*. Springer, 2018, pp. 215–230.

[31] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 143–154.

[32] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan, "Availability in Globally Distributed Storage Systems," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10. USA: USENIX Association, 2010, p. 61–74.

[33] T. Heath, R. P. Martin, and T. D. Nguyen, "Improving Cluster Availability Using Workstation Validation," in *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 217–227.

[34] T.-T. Lin and D. P. Siewiorek, "Error Log Analysis: Statistical Modeling and Heuristic Trend Analysis," *IEEE Transactions on Reliability*, vol. 39, no. 4, pp. 419–432, 1990.

[35] B. Schroeder and G. A. Gibson, "A Large-Scale Study of Failures in High-Performance Computing Systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 4, pp. 337–350, 2010.

[36] L. Egghe, "Zipfian and Lotkaian Continuous Concentration Theory," *Journal of the American Society for Information Science and Technology*, vol. 56, no. 9, pp. 935–945, 2005.

[37] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab *et al.*, "Scaling Memcache at Facebook," in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013, pp. 385–398.

[38] H. Weatherspoon and J. D. Kubiatowicz, "Erasure Coding vs. Replication: A Quantitative Comparison," in *International Workshop on Peer-to-Peer Systems*. Springer, 2002, pp. 328–337.

[39] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur, "XORing Elephants: Novel Erasure Codes for Big Data," *Proceedings of the VLDB Endowment*, vol. 6, no. 5, pp. 325–336, Mar. 2013.

[40] X. Li, R. Li, P. P. Lee, and Y. Hu, "OpenEC: Toward Unified and Configurable Erasure Coding Management in Distributed Storage Systems," in *17th USENIX Conference on File and Storage Technologies (FAST 19)*, 2019, pp. 331–344.

[41] T. Zhou and C. Tian, "Fast Erasure Coding for Data Storage: A Comprehensive Study of the Acceleration Techniques," in *17th USENIX Conference on File and Storage Technologies (FAST 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 317–329.

[42] M. Xia, M. Saxena, M. Blaum, and D. A. Pease, "A Tale of Two Erasure Codes in HDFS," in *13th USENIX Conference on File and Storage Technologies (FAST 15)*. Santa Clara, CA: USENIX Association, 2015, pp. 213–226.

[43] J. C. Chan, Q. Ding, P. P. Lee, and H. H. Chan, "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)*, 2014, pp. 163–176.

[44] B.-G. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, M. F. Kaashoek, J. Kubiatowicz, and R. Morris, "Efficient Replica Maintenance for Distributed Storage Systems," in *Proceedings of the 3rd Conference on Networked Systems Design and Implementation - Volume 3*, ser. NSDI'06. USA: USENIX Association, 2006, p. 4.

[45] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, "A "Hitchhiker's" Guide to Fast and Efficient Data Reconstruction in Erasure-coded Data Centers," *Proceedings of the 2014 ACM Conference on SIGCOMM*, vol. 44, no. 4, pp. 331–342, Aug. 2014.

[46] K. M. Greenan, X. Li, and J. J. Wylie, "Flat XOR-based Erasure Codes in Storage Systems: Constructions, Efficient Recovery, and Tradeoffs," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*. IEEE, 2010, pp. 1–14.

[47] O. Khan, R. Burns, J. Plank, W. Pierce, and C. Huang, "Rethinking Erasure Codes for Cloud File Systems: Minimizing I/O for Recovery and Degraded Reads," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, ser. FAST'12. USA: USENIX Association, 2012, p. 20.

[48] K. Rashmi, M. Chowdhury, J. Kosaian, I. Stoica, and K. Ramchandran, "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, 2016.

[49] P. Viotti, D. Dobre, and M. Vukolić, "Hybris: Robust hybrid cloud storage," *ACM Trans. Storage*, vol. 13, no. 3, Sep. 2017.

[50] S. Li, Q. Zhang, Z. Yang, and Y. Dai, "BCStore: Bandwidth-Efficient In-memory KV-store with Batch Coding," in *International Conference on Massive Storage Systems and Technology*, 2017.

[51] K. Taranov, G. Alonso, and T. Hoefler, "Fast and strongly-consistent per-item resilience in key-value stores," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. New York, NY, USA: Association for Computing Machinery, 2018.

[52] D. Shankar, X. Lu, and D. K. Panda, "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)*, June 2017.

[53] H. Shi, X. Lu, D. Shankar, and D. K. Panda, "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 2019, Phoenix, AZ, USA, June 22-29, 2019*. ACM, 2019, pp. 219–230.