NetEC: Accelerating Erasure Coding Reconstruction With In-Network Aggregation

Yi Qiao, Menghao Zhang, *Graduate Student Member, IEEE*, Yu Zhou, *Member, IEEE*, Xiao Kong, Han Zhang, *Member, IEEE*, Mingwei Xu, *Senior Member, IEEE*, Jun Bi, and Jilong Wang

Abstract—In distributed storage systems, Erasure Coding (EC) is a crucial technology to enable high data availability. By downloading parity data from survived machines, EC can reconstruct lost data with much lower storage overheads than data replication. However, this reduction in storage cost comes at the expense of extra performance problems: *low reconstruction rate*, *high degraded read latency*, and *high host CPU utilization*. Our analysis shows that these performance problems are deeply rooted in the *host-based* EC processing. To resolve these problems, we present NetEC, an in-network accelerating framework that fully offloads EC to the new generation programmable switching ASICs. We propose Explicit Buffer Size Notification (EBSN) to constrain decoding buffer usage, and design an on-switch one-to-many TCP proxy to integrate EBSN with TCP. We also design two parallel Galois Field (GF) offloading methods—table lookup and bitmatrix methods—to maximize parsable bytes. We implement NetEC on programmable switches and integrate it with HDFS. Extensive evaluations show that NetEC improves the reconstruction rate by 2.7x-6.8x, reduces the degraded read latency significantly, and removes the host CPU overhead completely. We also emulate multi-rack scenarios and show that NetEC is able to support ∼GB/s reconstruction rate and tens of concurrent tasks.

Index Terms—Erasure coding, distributed storage sytems, programmable switch, software-defined networks

1 Introduction

Distributed storage systems are an important building block of modern data centers. As the businesses expand and services develop, the storage systems are scaling rapidly where unexpected failures happen frequently [1], [2], [3]. Traditionally, these systems maintain multiple replications across machines and racks to ensure data availability. Although disk storage seems inexpensive, replicating the entire data footprint is infeasible as the data centers scale to petabytes [4]. As a result, many large-scale distributed storage systems are transitioning to the use of erasure coding [1], [3], [5], which offers high availability with much lower storage overheads compared to data replication.

The most popular erasure coding scheme is the family of Reed-Solomon codes [6]. An RS code is associated with two parameters, k and r. A RS(k, r) code encodes k units of data into r units of parities. The original k units can be reconstructed using any k out of $(k + r)$ units of data. It thus allows for tolerating any r failures of $k + r$ units. A common choice of RS code is RS (10,4), in which any 4 failures can be tolerated, with 1.4x storage overheads. In contrast, the storage overheads becomes 3x to achieve similar degree of availability in replication-based systems.

As revealed by many previous literature [3], [4], [7], erasure coding trades storage cost with extra performance overheads. Among these overheads, the most discussed problems are long reconstruction time, high degraded-read latency[1] and heavy resource usage. These problems affect not only the system Mean Down Time (MDT), but also the performance of high-level services like database and key-value store. These problems have received great attention from academia and industry [3], [4], [7], [8]. However, according to a recent work [7], there is no more than 50 MB/s reconstruction rate (several hours to reconstruct a 1TB HDD), which can be disastrous even for latency-insensitive or off-line services.

Through extensive analysis and case studies (Section 2.2), we identify that *host-based* EC processing is the fundamental issue behind these three problems. In particular, as shown in

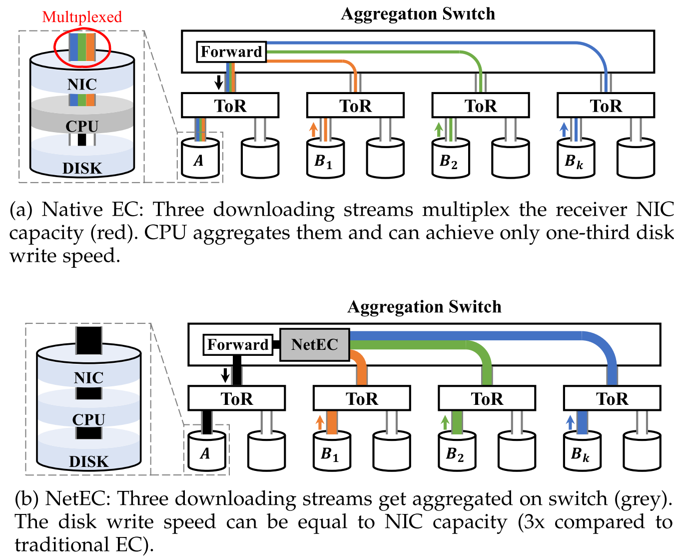
1. Read operations to missing data in EC are "degraded" due to higher latency and lower throughput compared with replication-based systems. This is because in EC these operations are served with on-the-fly reconstruction, while in replication they are served directly from one of the backups.

• *Yi Qiao, Menghao Zhang, Xiao Kong, Han Zhang, and Jun Bi are with the Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing 100084, China, and also with the Beijing National Research Center for Information Science and Technology (BNRist), Beijing 100084, China. E-mail: {qiaoy21, zhangmh16, kx19}@mails.tsinghua.edu.cn, {zhhan, junbi}@tsinghua.edu.cn.*
• *Yu Zhou is with Alibaba Inc., Hangzhou 311121, China. E-mail: yuzhou.zy@alibaba-inc.com.*
• *Mingwei Xu is with the Institute for Network Sciences and Cyberspace, and the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China, and with the Beijing National Research Center for Information Science and Technology (BNRist), Beijing 100084, China, and also with Quan Cheng Laboratory, Jinan 250103, China. E-mail: xumw@tsinghua.edu.cn.*
• *Jilong Wang is with the Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing 100084, China, and with the Beijing National Research Center for Information Science and Technology (BNRist), Beijing 100084, China, and also with Peng Cheng Laboratory, Shenzhen, Guangdong 518066, P. R. China. E-mail: wjl@cernet.edu.cn.*

Manuscript received 1 Nov. 2020; revised 1 June 2021; accepted 17 Jan. 2022. Date of publication 25 Jan. 2022; date of current version 4 Apr. 2022.
This work was supported in part by Joint Research on IPv6 Network Governance: Research, Development and Demonstration under Grant 2020YFE0200500, and in part by the Natural Science Foundation of China under Grant 62002009.
(Corresponding author: Han Zhang.)
Recommended for acceptance by S. K Prasad.
Digital Object Identifier no. 10.1109/TPDS.2022.3145836

1045-9219 © 2022 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: Tsinghua University. Downloaded on April 06,2022 at 15:20:15 UTC from IEEE Xplore. Restrictions apply.



(a) Native EC: Three downloading streams multiplex the receiver NIC capacity (red). CPU aggregates them and can achieve only one-third disk write speed.

(b) NetEC: Three downloading streams get aggregated on switch (grey). The disk write speed can be equal to NIC capacity (3x compared to traditional EC).

Fig. 1. NetEC overview and comparisons with Native EC. Colored lines represent downloading stream, and black lines represent reconstructed stream. The line width represents throughput.

Fig. 1a, end host network I/O (typically NIC capacity) is *multiplexed* by k downloading data streams, so that the effective reconstruction *goodput* is bounded by $1/k$ of the available network I/O, which leads to low reconstruction rate. Besides, the downloading streams induce severe incast [9] in the outbound switch interface connected to the receiver (server A in Fig. 1a), resulting in high degraded read latency. Moreover, *every single* byte from all downloading streams has to be inspected by CPUs to conduct encoding or decoding operations, leading to high CPU utilization on end hosts. Although numerous works [3], [4], [7], [8] have been devoted to these performance problems, as long as processing is conducted on end hosts, these problems cannot be resolved completely. One might think of using a dedicated server or other hardwares like FPGA and making it stand as as middlebox to perform erasure coding. However, they inevitably need NICs to connect to the networks, and still face low goodput when the bounded NIC bandwidth is shared by multiple data streams.

The recent in-network computation paradigm [10], [11], [12], [13], [14], [15] provides an unprecedented opportunity to address these problems. The emergent programmable switching ASICs have ultra-high backplane capacity and line-rate processing capabilities. By exploiting the on-switch memory and stateful processing, researchers extend the usage of programmable switches to carrier or accelerator of various applications such as key-value caches [12], load balancing [16], coordination services [14] and network monitoring [17].

We argue that offloading EC to programmble switching ASICs can solve the above-mentioned three problems. First, on a programmable switch [18], data streams arrive at *different* interfaces, get aggregated and forwarded to yet *another* interface. As a result, there is no sharing of bandwidth. As illustrated in Fig. 1b, where computation (gray box) is moved from end hosts to the switch, the reconstructed data (black) is able to make full use of the entire bandwidth available on the host NIC, so that we can achieve higher disk write speed. The wider black line compared to Fig. 1a indicates the improvement of reconstruction rate. Second, incast is avoided because only one data stream of reconstructed data is sent via the outbound interface, resembling the case in data replication. Finally, in-network computation completely removes extra CPU usage for RS decoding. Storage nodes receive *already reconstructed* data, which can be directly written to disks.

Therefore, we present NetEC, an in-network acceleration framework that fully offloads EC to programmable switching ASICs. RS decoding is modelled with Galois Field (GF) vector dot-products, which can be further decomposed to GF multiplication and partial XOR sum updates. The programmable switch performs GF multiplication on parity data contained in arriving packets and updates the buffered partial XOR sums. When reconstruction completes, the last arrived packet is sent with the decoding result encapsulated in its payload. Storage nodes receive reconstructed data ready to be written to disks. Although the idea of NetEC seems simple conceptually, we address two key challenges:

Performance. Can NetEC achieve ~GB/s reconstruction rate to support next-generation storage medium? Programmable switches can only parse and process limited number of bytes(<500) due to parser and stateful ALU capabilities [17]. Compare to traditional 1500B MTU (Maximum trasmission unit), small packet size puts large burdens on network and end hosts, and downgrades end-to-end throughput. To support larger packet size, NetEC uses two methods for GF multiplication offloading in parallel: table lookup method and bitmatrix method. They consume different parts of switch resources (SRAM and ALU respectively), so that more bytes can be processed given limited switch resources. NetEC further increases packet size with packet recirculation, i.e., let packets go through the processing pipeline multiple times.

Scalability. Can NetEC support tens of concurrent reconstruction tasks to tolerate multiple failures or rack-level failures? This is necessary when several machines simultaneously fail in large storage clusters. NetEC needs to temporarily buffer partial decoding results, and if downloading streams have non-synchronized transmission rates, the on-switch SRAM consumption would increase drastically. Therefore, we need to constrain the decoding buffer size for each task to run multiple concurrent reconstruction tasks with limited SRAM resources (~100MB). To this end, we design the Explicit Buffer Size Notification (EBSN) mechanism to synchronize the downloading rates and constrain buffer usage. The programmable switch explicitly informs senders of the remaining allocated buffer size piggybacked in ACK packets, so that the senders automatically keep in pace with each other and do not send more data than the buffer can hold. We take extra efforts to augment Transmission Control Protocol (TCP) with EBSN by designing a one-to-many TCP proxy to enable easier integration with existing systems.

NetEC is designed to be incrementally deployable. Adopting NetEC in HDFS only requires configuring a customized EC policy and replacing programmable switches in the cluster. NetEC does not require every switch to be programmable, and the programmable switches can also run other network applications because NetEC takes up only a portion of switch resources. Besides, NetEC is orthogonal to other EC solutions. They can evolve separately and cooperate to build a better system. NetEC can potentially support

Authorized licensed use limited to: Tsinghua University. Downloaded on April 06,2022 at 15:20:15 UTC from IEEE Xplore. Restrictions apply.

(a) Reconstruction rate.

(b) Degraded read latency.

(c) CPU Utilization.

Fig. 2. Case studying of current EC in HDFS indicates considerable performance overheads.

other RS-based codes, like LRC codes [3], and can be integrated to other EC systems as a basic coding scheme.

In summary, we make the following contributions:

- We propose NetEC that offloads EC to switching ASICs to solve the three major EC problems (Section 3).
- We design Explicit Buffer Size Notification (EBSN) to constrain buffer usage for concurrent tasks (Section 4).
- We design two parallel GF offloading methods to support larger packet size and better throughput (Section 5).
- We implement NetEC on hardware programmable switches and integrate it with HDFS (Section 6).
- We conduct extensive evaluations and show that NetEC improves the reconstruction rate by 2.7x-6.8x, reduces the degraded read latency significantly, and removes the host CPU overhead completely. We emulate multi-rack scenarios showing ~GB/s reconstruction rate and tens of concurrent tasks (Section 7).

In addition, we show related works in Section 9, and conclude the paper in Section 10.

2 Background and Motivation

2.1 Reed-Solomon Code

We introduce Reed-Solomon (RS) code in this part and show that encoding and decoding can both be modelled with vector dot-products on Galois Fields (GF). GF(2^w) refers to the set containing all w-bit binary numbers. For simplicity, the readers may comprehend GF arithmetics as usual integer arithmetics. We refer to the smallest granularity of data as *symbol*, which we choose to be a 16-bit word.

Encoding. We describe the RS(k,r) coding system using matrix-vector products[19]. As shown in Equation (1), the RS(k,r) code encodes k symbols into r parity symbols with a $(k+r)\times k$ *generating matrix* R, composed of an *identity matrix* and a *redundancy matrix*. We use column vector D to denote a set of original data symbols $d_1, d_2, \ldots, d_k$, and C to denote data and parity symbols $d_1, \ldots, d_k, p_1, \ldots, p_r$. Multiplication of the generating matrix R and D yields C. More specifically, each parity symbol can be computed with $p_i = \sum_{j=1}^{k} a_{i,j} d_i$, where $a_{i,j}$ are elements of the redundancy matrix

$$RD = \begin{bmatrix} 1 & & \\ & \ddots & \\ & & 1 \\ a_{1,1} & \cdots & a_{1,k} \\ \vdots & \ddots & \vdots \\ a_{r,1} & \cdots & a_{r,k} \end{bmatrix} \cdot \begin{bmatrix} d_1 \\ \vdots \\ d_k \end{bmatrix} = \begin{bmatrix} d_1 \\ \vdots \\ d_k \\ p_1 \\ \vdots \\ p_r \end{bmatrix} = C. \quad (1)$$

Decoding. RS(k,r) tolerates at most r lost data words through reconstruction using (any) k of the survived symbols. To reconstruct, we build a $k\times k$ *reconstruction matrix* R' by deleting r rows corresponding to lost data symbols from generating matrix R. Applying the same row deletions on C, we obtain C', which contains k survived symbols. $R'D = C'$, and with matrix inversion, we get $D = (R')^{-1}C'$. This means that we can retrieve the original D by multiplying $(R')^{-1}$ with the survived symbols C'. More specifically, each data symbol can be computed with $d_i = \sum_{j=1}^{k} r_{i,j} c'_i$, where $r_{i,j}$ are elements of the inverted reconstruction matrix.

To conclude, the encoding and decoding process can both be modelled with *GF vector dot-products*. In later sections, we will discuss how it can be transformed to operations that are offloadable to switching ASICs.

2.2 Case Study in HDFS-EC

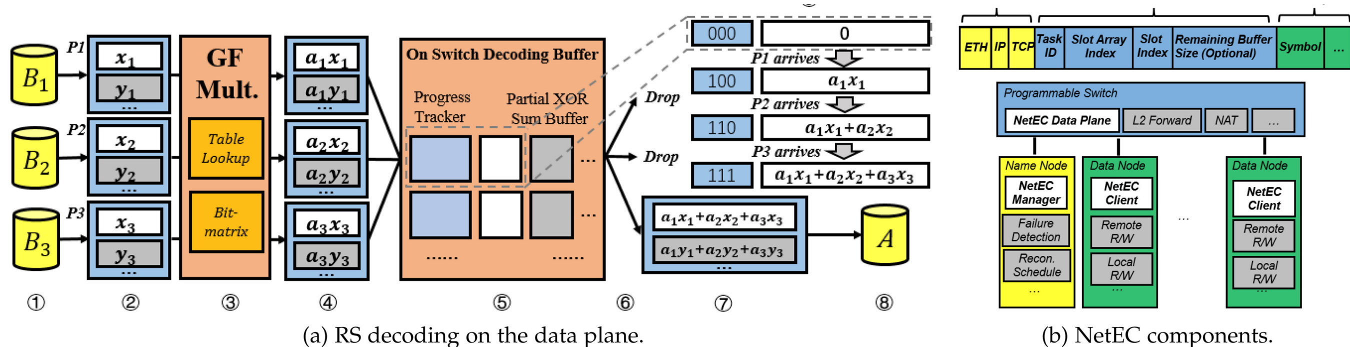
To verify arguments made in Section 1 about EC performance, we measure the reconstruction rate, degraded read latency, and host CPU utilization in HDFS-EC. HDFS [5] starts to natively support EC since version Hadoop-3. We use the built-in `RS-3-2-1024k`, `RS-6-3-1024k` and `REPLICATION`[2] HDFS-EC policies [20]. The cluster setup is shown in Section 7.

First of all, reconstruction rate in RS(k,r) is no more than $1/k$ of replication, resulting in at least k times the reconstruction time. As shown in Fig. 2a, during reconstruction, we increase the allowed NIC bandwidth using Linux `tc`. We observe that the reconstruction rate is always approximately one-third (or one-sixth) of that of replication before it is bounded by HDD write I/O.

Second, degraded reads are already slow because of low construction rate. Fig. 2b shows degraded read latency reflected by single block reconstruction time. Degraded read latency of RS(k,r) is roughly k times of replication. Besides, persistent downloading streams induce severe incast on the switch egress buffers, causing large queues, and thus extra queuing delay for read requests. The long-lived downloading streams greedily grasp all available bandwidth, resulting in unfairly low throughput of on-the-fly reconstruction. This unfairness between large and short flows has been extensively studied [9], [21], and is also observed in our case study. In Fig. 2b, if read requests are made to the *under-reconstruction* node, the degraded read latency is drastically increased. The reconstruction traffic of

2. `REPLICATION` is a special EC policy that users can choose in the EC policy XML file, forcing certain directories to use 3-way replication. `REPLICATION` should be differentiated with the HDFS replication policy, used in `hdfs-site.xml`

Authorized licensed use limited to: Tsinghua University. Downloaded on April 06,2022 at 15:20:15 UTC from IEEE Xplore. Restrictions apply.



(a) RS decoding on the data plane. (b) NetEC components.

Fig. 3. NetEC overview.

the requested block fails to compete with heavy incast background traffic, leading to unfairly low throughput and longer reconstruction time.

Last but not least, CPU utilization on end hosts is high since CPUs have to inspect *every* single byte from *all* k downloading streams, taking away valuable computational resources for other applications. Fig. 2c shows the host CPU utilization with respect to reconstruction rate in an SSD setting. With Intel ISA-L[22] support, the CPU processing throughput manages to keep up with I/O at the cost of high utilization. We observe that the CPU utilization is almost linear to both reconstruction rate and the number of parity nodes (k as in RS(k, r)).

2.3 Towards an In-Network Solution

The above mentioned problems are hard to eliminate as long as EC processing is conducted on end hosts. Scaling up by upgrading NICs or dedicating more CPU cores seems feasible, but can potentially lead to low utilization of these hardware because reconstruction does not happen frequently. Therefore, we seek to move erasure coding from end hosts to the network.

Programmable Switching ASICs. Programmable switching ASICs [18], [23] are designed to implement match-action forwarding applications at line rates that are orders of magnitude higher than CPU, NPU and FPGAs. Programmable ASICs have very specialized architecture that exploits pipelining and instruction-level parallelism[18] to support terabit-rate data plane functionalities. Current commodity products include Barefoot Tofino [24] and Cavium XPliant [25].

Switch ASICs differ from other programmable devices, typically FPGAs in the following two perspectives. On one hand, Switch ASICs has orders of magnitude higher packet processing capability than FPGAs and NPUs, and more importantly, performance is *non-degradable*, i.e., computation complexity and resource usage of the installed program does *not* affect the processing delay ($<1\mu$s), jitter and throughput ($\sim$Tb/s) of the switch pipeline [17], even under extreme throughput load. Packets go through a fixed number of stages on switch, with a constant traversal time. A programmable switch can install multiple programs which use separate resources (SRAM and ALUs) in an exclusive manner. These programs will not affect each other's performance. On the other hand, the switch ASIC programming model is less expressive than other hardwares in trade of performance. It is initially designed to ease network protocol customization, so it can only support applications that can be mapped to match-action table pipelines.

3 NetEC Overview

3.1 Terms

We first define terms used in following sections. We refer to the node on which a data *symbol* m is reconstructed as the *reconstructing node* (termed A, ⑧ in Fig. 3a). m is reconstructed from the *symbol-stripe* $x_1, x_2, \ldots, x_k$ (or $y_1, y_2, \ldots, y_k$) downloaded from *survived nodes* $B_1, B_2, \ldots, B_k$ (①). To simplify, we term all survived nodes as Bs. The switch running NetEC is termed S. The set of packets that contains the same set of symbol-stripes are termed the *packet-stripe*. A packet-stripe contains multiple symbol-stripes (②). For example, in Fig. 3a, $P1$, $P2$, $P3$ form a packet stripe, and contain symbol-stripes x_i, y_i, etc. A *reconstruction task* refers to the reconstruction process of one block of data.

3.2 RS Decoding on the Data Plane

We illustrate how RS decoding is performed on switch data plane in Fig. 3a. For every incoming packet from Bs (②), NetEC extracts symbols x_i, y_i, etc., and performs GF multiplication on them (③). We design two parallel GF multiplication method: the table lookup method and the bitmatrix method (Section 5). By "parallel", we mean these two methods apply to different symbols in a packet simultaneously. For example, we can calculate $a_i x_i$ with the table lookup method and $a_i y_i$ with the bitmatrix method. In this way, the total number of symbols in a packet is the sum of the number of symbols supported by *both* methods. The results are written back in the extracted header fields (④).

The switch then picks a slot (detailed in Section 3.3) in the partial XOR sum buffers implemented with registers, and XORs the symbols with buffer contents. It also updates the progress tracker, whose ith bit is flipped if the packet comes from the ith survived node. As other packets of the packet-stripe arrive, the buffer content changes as shown in Fig. 3a (⑨), where "+" means bit-wise XOR. The switch drops all packets except the last arrived packet in a packet-stripe (⑥). When the progress tracker becomes all ones, the switch knows the whole packet stripe is received. It overwrites the payload of the last arrived packet with the finished XOR sums (⑦), and forwarded it to A (⑧). We can see that decoding buffer is needed since the arrival of first packet until the departure of the last packet. A receives packets containing reconstructed data, which can be directly written to disk.

Authorized licensed use limited to: Tsinghua University. Downloaded on April 06,2022 at 15:20:15 UTC from IEEE Xplore. Restrictions apply.

Buffer Slot Indexing. In every packet, Bs explicitly include a slot index, indicating which slot to hold partial decoding result for the packet-stripe. Packets of a packet-stripe share the slot index, so that the extracted symbols are XOR-ed altogether in the same buffer. The allocated SRAM is used as a ring buffer. Suppose the total slot size is M (total allocated SRAM size divided by packet size), then the ith packet sent by Bs has the slot index i (mod M). We adopt measures discussed later to guarantee that the buffer will never be full, so that there will not be collisions of slot usage.

3.3 NetEC Components

NetEC consists of the following three system components and NetEC Packet Format (Fig. 3b).

NetEC Data Plane. NetEC Data Plane is the switching ASIC program responsible for RS decoding. It performs GF multiplication on symbols and aggregates symbols from the same packet-stripe. The detailed workflow has been shown earlier (Section 3.2). It also implements Explicit Buffer Size Notification (EBSN, Section 4.1), preventing buffer overflow by explicitly informing NetEC Clients of the remaining buffer size. The one-to-many TCP proxy is deployed in our HDFS implementation.

NetEC Manager. NetEC Manager manages the life cycle of NetEC reconstruction process. In our HDFS implementation, NetEC Manager co-locates with the HDFS namenode with access to all RS decoding matrices (Section 2.1). When a block reconstruction is scheduled, NetEC Manager takes over and manages the reconstruction with its own logics. Whether data is reconstructed with NetEC or native EC is transparent to other modules of the file system. NetEC Manager also keeps track of ongoing tasks and the switch slot arrays they use.

NetEC Client. NetEC Client is responsible for data transmission. NetEC Clients use sliding windows for rate control. The window size is updated with the remaining buffer size advertised by the NetEC Data Plane. In our HDFS implementation, NetEC Clients co-locate with datanodes and use traditional TCP, enabled by the one-to-many TCP proxy (Section 4.3).

NetEC Packet Format. NetEC packets are formatted as shown in Fig. 3b. A NetEC packet includes a 4-bit `task ID` field, an 8-bit `slot array index` field, a 16-bit `slot index` field and a 16-bit `remaining buffer size` field. A reconstruction task uses one slot array, which is a continuous piece of switch SRAM. Slot index is used as shown in Section 3.2. The remaining buffer size field is used by EBSN mechanism.

3.4 NetEC Reconstruction Life Cycle

When a block reconstruction request is made, NetEC Manager selects an unoccupied slot array and a reconstruction task ID. NetEC Manager sends the task ID, the slot array index and the decoding matrices to involved NetEC Clients on A and Bs, and the NetEC Data Plane on S.

Bs transmit parity data with the NetEC Packet Format. The packets from Bs are aggregated as previously shown in Section 3.2. When A receives the reconstructed packet, it will reply with an acknowledgement (ACK) packet. S piggybacks the remaining buffer size field and multicast ACK packets to Bs. On receipt of these ACK packet, Bs adjust their sending rates accordingly. After the whole block is reconstructed, NetEC Clients inform the NetEC Manager to finish the task and release the allocated slot array.

Fallback Mechanisms. When some downloading streams are broken or the receiver detects corrupted blocks (e.g., wrong checksums), the NetEC manager will terminate the reconstruction task and attempt a retry. Also, if the reconstruction task still fails (e.g., when there are insufficient resources), the system will gracefully degenerate into native EC described in Fig. 1a, where switches only deliver traffic without getting involved in computation. Other ongoing reconstruction tasks are not affected. Basic network forwarding functionality still works normally, even when there are faults.

4 Constraining Decoding Buffer Usage

In this section, we propose Explicit Buffer Size Notification (EBSN) to constrain decoding buffer usage per task so that NetEC can support multiple concurrent reconstruction tasks. Then we introduce one-to-many TCP proxy designed for our HDFS integration, and how to combine EBSN to it.

4.1 Explicit Buffer Size Notification (EBSN)

To avoid buffer overflow, our basic idea is to explicitly inform senders of the remaining buffer size on switch. We call this method *Explicit Buffer Size Notification (EBSN)*. We assume that the transmission protocol has an ACK (acknowledgement) mechanism, which is generally needed for reliability and rate control, and that the ACK packets pass our NetEC switch. We piggyback the remaining buffer size value in ACK packets. If the ACK packets already have similar fields, like TCP window size, we overwrite this field if the remaining buffer size is smaller than the field value. Senders maintain a sliding window to keep track of sent and unacknowledged packets, and update the window size with the remaining buffer size on receipt of ACK packets.

To show the effectiveness of EBSN, we analogue it with the TCP rate control mechanism. TCP receivers maintain a receive buffer to hold received data not yet consumed by upper layer applications. It constantly informs its peer of the remaining receive buffer size in the `window size` field of TCP header, so that the peer will not send more data than the receiver can receive. Similarly, the NetEC explicitly inform senders of the remaining decoding buffer size, so that the senders will not send data more than the decoding buffer can hold. In some way, EBSN can be viewed as "offloading" the entire receive buffer to the switch data plane.

4.2 Choice of Buffer Size

The choice of buffer size allocated for each task is important because it affects maximum reconstruction rate. The decoding buffer should be able to hold all in-flight packets, bounded by BDP (bandwidth-delay product) of the link. Suppose we use NetEC to reconstruct an SSD at its full speed 1GB/s. In data centers, RTT is typically around 250 μs [21]. In this case, the in-flight data size I is estimated to be 250KB.

However, we cannot simply allocate buffer equal to the in-flight data size because the remaining buffer information cannot be advertised to senders *timely*. First, most NICs adopt

Authorized licensed use limited to: Tsinghua University. Downloaded on April 06,2022 at 15:20:15 UTC from IEEE Xplore. Restrictions apply.

TCP segmentation offloading, which reduces the number of ACK packets. Multiple packets are aggregated to a larger packet by NICs before being passed to higher network stacks, and the network stack may respond only one ACK for the aggregated packet. This means the change of the remaining buffer size will likely be out-of-date because of the "diluted" ACK packets. Second, there also exist a delay before the sender is aware of the remaining buffer size piggybacked in ACK packets. Before senders are notified the remaining buffer size, the buffer may still be accumulating data. These two problems potentially lead to buffer overflow.

To resolve these problems, we allocate buffer size to be $2I$, *twice* as the estimated maximum in-flight packet size I, but only "expose" I to senders. The other half of I serves as a headroom for potential overflow packets due to reasons discussed above. Specifically, suppose the current buffer consumption is C and the remaining buffer size we will advertise to senders is R. If $C < I$, we let $R = I - C$; otherwise, we consider that buffer overflow already happens and let $R = 0$, even if C has not reached $2I$. Before this information ($R = 0$) reaches the senders, they might still be transmitting packets, but the size of these in-flight packets will be no more than the value of last notified R. Therefore, these data can be held safely in the other half of the allocated buffer, because R is always no more than I. Allocating $2I = 500$KB with the above method, we still allow at most $I = 250$KB in-flight data for senders, so that the transmission rate can still saturate SSD writes (1GB/s) in our setting.

Doubling buffer size usage seems wasteful, but because of the limited throughput bounded by SSD write speed and small RTTs in data centers, we only use 500KB per task, which is manageable for on-chip memory (0.5% in a switch with 100MB total SRAMs).

4.3 One-to-Many TCP Proxy With EBSN

While it is easy to implement a simple NetEC version with EBSN (as shown in Section 6), we take extra efforts to augment TCP with EBSN by designing the one-to-many TCP proxy. The reasons are two-fold. First, TCP already has the `window size` header field that we can easily parse and modify on switch to support EBSN. Second, it will be easier to integrate NetEC to existing systems (like HDFS in our implementation) by augmenting TCP than designing a clean-slate protocol, because TCP is still a predominant transport layer protocol for distributed file systems.

4.3.1 Overwriting the TCP Window Size

TCP headers include a field called `window size` which represents the number of bytes the receive buffer can hold. The sending side updates its sliding window size upon receiving every ACK to avoid overflowing the receive buffer of its peer. In NetEC, we simply overwrite the TCP window size W with the remaining buffer size B if $B < W$. In this way, the senders will never transmit more data than the receive buffer can hold.

4.3.2 One-to-Many TCP Proxy

We build a one-to-many TCP proxy to manage downloading streams of parity data sent by survived nodes Bs and reconstructed data produced by the switch S. From Bs' point of view, they all connect to S, sending parity data. From A's point of view, it connects only to S, receiving already reconstructed data.

Handshake. We first assign a VIP (virtual IP) address to the switch S, and let A connects to this VIP. S modifies the source IP of the SYN packet from A to the VIP assigned to S, and multicasts the SYN packet to Bs (with corresponding destination IPs). Therefore, from Bs' point of view, they receive SYN packets from S and will reply S with SYN-ACK packets. After receiving all SYN-ACK packets from Bs, S sends one SYN-ACK packet to A. From A's point of view, it successfully connects to S, and will reply with an ACK packet. S multicasts this ACK packet to Bs after similar modifications it does to the SYN packet. Then, S successfully connects to all Bs.

4.3.3 Rate Control and Packet Loss Recovery

NetEC prevents senders from overwhelming the allocated decoding buffer and maintains synchronicity of sending rates. All Bs will automatically have synchronized rates that are almost always equal to the rate of the slowest sender. If any sender becomes faster than others, the remaining buffer size will reduce. The reduced value will be quickly advertised to all senders through EBSN, and the faster sender will reduce its window size, thus its transmission rate. The completion time of EC reconstruction is dominated by the "straggler" of the senders, so reducing the rate of other senders to the straggler will not impact the completion time.

Next we discuss how NetEC deals with two types of packet loss. We mainly focus on lost packets from Bs to S because those from S to A follows similar analysis, and those from A to S and S to Bs (pure ACKs) have negligible effects as long as any following ACK packet is successfully transmitted. The first type of packet loss is RTO, or re-transmission timeout, where no packets are received on S for a period of time from one of Bs, saying B_1. During this time period, no aggregation can be performed on switch, and therefore A receives no packets, and sends no ACK packets to Bs. When Bs receive no ACK packet for a period of time, they detect RTO and start re-transmission. The second type of packet loss is duplicate ACKs. Suppose one packet is lost on the path from B_1 to S, but following packets are successfully transmitted. Also, no packet loss happen from other Bs. Then from A's point of view, it receives the following packets and detects the missing packet. A then sends ACK packet requesting the missing packet anytime it receives a following packet, and these ACK packets will be multicast to all Bs. After Bs receives three duplicate ACKs requesting the same packet, it confirms that this packet is truly lost, and re-transmits this packet.

To properly handle re-transmitted packets and ensure correct decoding, we design the progress tracker (Fig. 3a, ⑨). In the above example for duplicate ACKs, only one packet in the packet stripe (from B_1) is lost, while packets from other Bs reach S and contribute to the partial sum. However, all Bs detects duplicate ACKs and re-transmits the required packet. The switch has to identify the real lost packet (from B_1), and disregard already received packets (from other Bs). The NetEC progress tracker serves this purpose by maintaining tracker bits. In this example, the tracker

Authorized licensed use limited to: Tsinghua University. Downloaded on April 06,2022 at 15:20:15 UTC from IEEE Xplore. Restrictions apply.

bits is 0 for B_1, and 1 for others before re-transmission. Then, upon receiving the re-transmitted packets from all Bs, the switch checks the tracker bits and only processes the packet from B_1. S uses the data contained in this packet to finish partial sum calculation, and sends the reconstructed packet to A.

5 Offloading GF Multiplication

In this section, we aim to find the maximum number of bytes B_{max} on which to perform GF multiplication, given limited switch resources. We introduce and analyze two methods running in parallel. We also let packet recirculate once to double packet size: $pkt_size = 2B_{max}$.

5.1 Table Lookup Method

Table lookup method is widely used for accelerating GF operations [26], [27]. Suppose a data symbol m is to be reconstructed as follows: $m = \sum_{i=1}^{k} m_i = \sum_{i=1}^{k} a_i x_i$, where x_i are data symbols from survived nodes Bs and a_i are pre-computed coefficients. To multiply x_i with a_i, we look up $logx_i$ in a pre-installed logarithm table and add it with the pre-calculated $loga_i$: $logx_i + loga_i = logx_i a_i = logm_i$. Note that the addition here is integer addition, which is supported by the current programmable switching ASICs. Then we get m_i by looking up $logm_i$ in an exponent table.

The logarithm table and exponent table store mappings between an element and its logarithm or exponent in Galois field. The total size of the logarithm table is $w \times 2^w$ bit, because there are 2^w entries and each entry value has w-bit width. In NetEC, we choose $w = 16$ to be the symbol size, so the logarithm table takes up 131072 bytes. The exponent table takes up twice (262144 bytes) as much as logarithm table because the integer sum of two 16-bit integers ranges from 0 to 131070. However, exponent tables can be optimized to 131072 bytes, because the exponent table has repeated contents [19].

5.2 Bitmatrix Method

Another widely-used method is bitmatrix method [27], [28], [29]. Galois field operations can be conducted with *only binary XOR* with the underlying bit vectors and matrices [29]. Each element e in GF(2^w) (We let $w = 16$) can be alternatively represented as a column vector $V(e)$ or a matrix $M(e)$. $V(e)$ is equal to the binary representation of e, and the jth row of $M(e)$ is equal to $V(e^{2^{j-1}})$. $M(a)$ can be computed from a once in advance, and installed to the switch. This representation ensures the following important equation: $V(ax) = M(a)V(x)$, where a and x are elements in GF(2^w). Therefore, to calculate the multiplication of a and x ($V(ax)$), we convert a to its bitmatrix form ($M(a)$) and multiply it to the bit representation of x ($V(x)$). Specifically, to get the jth bit of $V(ax)$, we calculate the bit-wise dot-product of the jth row of $M(a)$ and $V(x)$ [29]. To do this, we first bitwise AND the jth row of $M(a)$ and x (Algorithm 1, Line 3). Then we need to XOR all 16 bits of AND-ed result altogether. However, if we do this one bit by one bit, we need up to 16 operations, which consume too much switch ALU resource. We adopt a method shown in Line 4-8 (Algorithm 1) to calculate the XOR sum of all 16 bits of res in $log_2(16) = 4$ times instead of 16 times.

TABLE 1
Switch Resources and Constraints

Variable	Value	Description
M	2 MB	Amount of SRAM per stage
A	128	Number of actions per stage
W	16	VLIW operation width
P	512 B	Maximum parsable bytes
w	16 bit	Symbol width
S_{tl}	260 KB	Lookup table size per symbol

Index	Constraints
C1	$B_{tl} \leq w \cdot N \cdot \frac{M}{S_{tl}}$
C2	$B_{bm} \leq w \cdot N \cdot W \cdot \frac{A}{80}$
C3	$B_{max} = B_{bl} + B_{bm} \leq P$

N refers to number of allocated stages.

Algorithm 1. Calculate the jth Bit of GF Multplication ax With Bitmatrix Representations

1: **procedure** GETBIT(a, x, j)
2: row = the jth row of bitmatrix $M(a)$
3: res = x AND row
4: res = res XOR ($res >> 8$)
5: res = res XOR ($res >> 4$)
6: res = res XOR ($res >> 2$)
7: res = res XOR ($res >> 1$)
8: **return** $res[0]$

5.3 Resource Usage Analysis

5.3.1 Switch Resources

In current switch architectures, a pipeline contains a fixed number of stages [18], [23]. Each stage has dedicated resources that can only be used *within itself* to achieve ultra-high line rate. In this part, we view the stage as the minimum allocation unit. We denote the number of allocated stages for NetEC as N. In Table 1, we summary the resource types and their typical values in our Barefoot Tofino hardware switch.

5.3.2 Resource Usage Analysis of GF Multiplication Methods

We analyze the resource usage of above two offloading methods, and find that the table lookup method mainly uses SRAMs, and the bitmatrix method only uses actions (ALU).

Table Lookup Method. In C1 (Table 1), we denote the number of bytes processed with this method B_{tl}. In current switching ASICs, tables can only be accessed in the same stage *once* per packet. NetEC requires table reads for every symbol in a packet, so we have to install multiple copies of lookup tables to the data plane.[3] According to Section 5.1, both tables need roughly 130KB SRAM. For each 16-bit symbol, we need one logarithm table and one exponent table, therefore, a total of $S_{tl} = 2 \times 130 = 260$ KB SRAM. Then, $\frac{M}{S_{tl}}$

3. Some recent works discuss how to relieve the memory access constraint. Without this constraint, we may only need one copy of lookup tables. dRMT [30] builds a memory pool accessible through a crossbar, and GEM [31] proposes RDMA-based external memory.

Authorized licensed use limited to: Tsinghua University. Downloaded on April 06,2022 at 15:20:15 UTC from IEEE Xplore. Restrictions apply.

refers to the number of lookup tables that can be installed per stage, equal to the maximum processed symbols per stage.

Bitmatrix Method. We denote the number of bytes processed with this method B_{bm}. Five steps (Algorithm 1, Line 3-7) are needed to calculate one bit, so a total of 80 actions are needed to calculate the 16-bit symbol. $\frac{A}{80}$ in C2 (Table 1) refers to the number of symbols processed per stage. The bitmatrix method leverages the VLIW support of programmable ASICs. Within each action, AND and XOR operations are simultaneously performed on W fields.

Hardware Metrics. Using the resource values shown in Table 1, we get $B_{tl} \leq w \cdot N \cdot \frac{M}{S_{tl}} = 15.38N$ bytes, and $B_{bm} \leq w \cdot N \cdot W \cdot \frac{A}{80} = 51.2N$ bytes. If we allocate one pipeline ($N = 12$), we choose $B_{tl} = 160B$ and $B_{bm} = 140B$ (Both meet constraints in Table 1), and $B_{max} = B_{tl} + B_{bm} = 300B$. Using packet recirculation discussed later, we get $pkt_size = 600B$. As shown in Section 7.3, this choice of packet size already achieves 80% throughput compared to the standard 1460B Maximum Transmission Unit (MTU). Future hardware may contain more stages, so that we can allocate more stages to NetEC. Also, each stage contains more resources, so that we can achieve larger B_{max} with the same N.

5.4 Packet Recirculation

In the previous part, we have analyzed the maximum number of bytes B_{max} processed by NetEC *in one pass* of the switch pipeline. We further leverage packet recirculation to let packets traverse the switch pipeline again to double the number of processed bytes. We configure end host MTU to be $2m + H$, where H stands for network header size, so that each packet has $2m$ size payloads.

For each incoming packet, we process the first m bytes of packets and truncate these bytes by not emitting them. Then the packet is loop-backed to the ingress pipeline again to process the next m bytes. After two passes, we can inspect the whole payload of the packet. Most incoming packets will be dropped except the last packet in a packet-stripe. The last packet, however, has to go through the pipeline again to retrieve and emit finished XOR sums from corresponding slots. The packet is assembled and leaves the switch.

Recirculating too many times could reduce switch overall throughput. Our choice of packet size parameter (i.e., pkt_size) and recirculating packets *once* can achieve a good balance between overall and end-to-end throughput: On the one hand, NetEC achieves near-optimal end-to-end throughput with well-chosen packet size (Section 5.3.2). On the other hand, recirculating once does *not* greatly impact switch throughput [11], [32]. Suppose the end-to-end throughput is 1GB/s, which is typical for SSD sequential write, the throughput consumption is 16Gb/s, taking up only a small portion of the switch overall throughput (0.48% for 3.3Tb/s capacity).

6 Implementation

We implement a prototype of NetEC on commodity hardware (codes available on [33]). Data plane components are implemented with P4 [23] (457 LoC) and compiled with Barefoot Capilano software suite. The controller of the data plane program is written with python (269 LoC). On server side, we implemented a set of HDFS-EC policies with Java

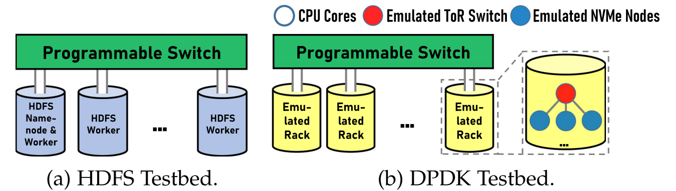


Fig. 4. Testbeds.

(2587 LoC). We also use DPDK (Data Plane Development Kit) [34] to implement NetEC simple version in C (652 LOC).

NetEC Data Plane. Our data plane implementation consists of RS decoding module and the control module. The RS decoding module is a direct mapping of Fig. 3a. The control module is responsible for overwriting the TCP window size and the translation of TCP SEQ/ACK and IP ID, described in Section 4.3. It also recalculates checksums, converts MAC addresses and performs simple L2 forwarding.

NetEC Manager. In our HDFS implementation, we overwrite `ErasureCodingPolicyManager` on HDFS namenodes and implement our own the request handling logics described in Section 3.4. We also add our NetEC policies to `SystemErasureCodingPolicies` so that we can configure them with `hdfs-core.xml` configuration file.

NetEC Client. In our HDFS implementation, we modify the socket creation logic in `BlockReaderRemote` on datanodes. Instead of establishing connections with all k survived nodes, the reconstructing node now connects to only the VIP address (Section 4.3) we assigned for this particular block reconstruction task. Also, we modify the data serialization and transmission logic. We simplify the original logic and use class `NetECPacketHeader` to restructure format before writing data to Java sockets.

NetEC Simple Version. NetEC simple version is used for evaluation of performance and scalability. It is not integrated with existing systems to eliminate framework overheads, and is implemented with DPDK to eliminate kernel network stack overheads. The data plane now only includes the RS decoding module. We implement a NetEC Client with sliding window and acknowledgement mechanism with DPDK.

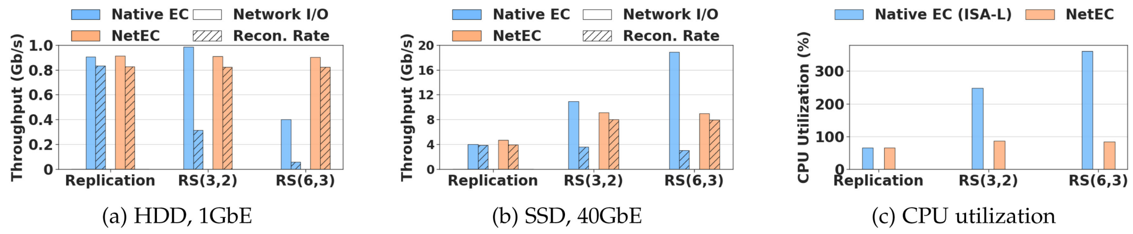
7 Evaluation

7.1 Experimental Setup

Topology. We build a cluster using 9 servers, each with two 12-core Xeon E5-2650v3 CPUs, 64GB of memory, 1TB HDD and 400GB Intel P3500 SSD. The servers are directly connected to a 3.3 Tb/s Barefoot Tofino programmable switch [24] via 1GbE and 40GbE NICs.

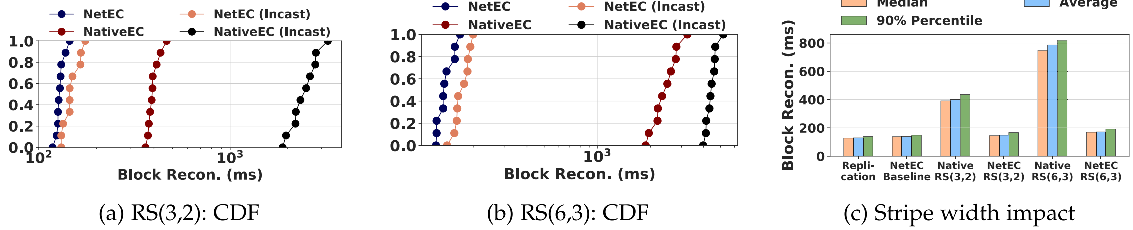
HDFS Testbed. To show effectiveness of NetEC, we build a Hadoop cluster as shown in Fig. 4a, deployed with one-to-many TCP proxy with EBSN (Section 4.3). Servers are directly connected to switch, each running Hadoop with our customized EC policies. To demonstrate the benefits of NetEC, we compare the following mechanisms: Replication, Native EC and NetEC. Replication and Native EC are HDFS default implementations. We use the default 128 MB block size and 3-way replication. The reason to compare NetEC with replication and Native EC is to show that NetEC significantly outperforms Native EC (up to k times in terms of

Authorized licensed use limited to: Tsinghua University. Downloaded on April 06,2022 at 15:20:15 UTC from IEEE Xplore. Restrictions apply.



(a) HDD, 1GbE (b) SSD, 40GbE (c) CPU utilization

Fig. 5. Reconstruction speed and CPU utilization.



(a) RS(3,2): CDF (b) RS(6,3): CDF (c) Stripe width impact

Fig. 6. Degrade read latency.

throughput) and even matches replication in various performance metrics.

DPDK Testbed. We use DPDK to emulate multi-rack NVMe (Non-Volatile Memory Express) scenarios. DPDK achieves ultra-high packet I/O with various technologies [34], and supports core affinity so that we can explicitly assign each core its own task. As shown in Fig. 4b, we assign one of the cores to run a virtual switch, which emulates the ToR switch, and assign other cores to perform read/write directly on DRAM, emulating an NVMe storage node. The emulated storage nodes run NetEC simple version (Section 6). The Tofino switch serves as an aggregate-level switch, which runs NetEC and forwards all cross-rack traffic. We aim to show that NetEC can scale to ~GB/s reconstruction rate and tens of concurrent tasks.

7.2 Effectiveness

7.2.1 Reconstruction Rate and Total Network Usage

NetEC significantly increases reconstruction rate and decreases network usage. In Fig. 5, each bar indicates the total network throughput or reconstruction rate (disk write speed). We manually kill one node to trigger replication or EC reconstruction. We use b/s (bits per second) as the unit for both reconstruction rate and network throughput.

Baseline. We first run NetEC in the replication scenario. All traffic transverses the entire NetEC processing pipeline. The left bars in Figs. 5a and 5b show that the processing of NetEC incurs negligible throughput overheads. Note that the reconstruction rate is slightly lower than network throughput because of network and NetEC header overheads.

HDD (1GbE). Fig. 5a shows results in an HDD and 1GbE setting. In RS(3,2) (middle bars), Native EC consumes 1.0 Gb/s network throughput but only achieves 0.31 Gb/s reconstruction rate, because the NIC capacity is multiplexed by three downloading streams. NetEC consumes 0.9 Gb/s network throughput and achieves 0.8 Gb/s reconstruction rate, which improves over Native EC by 2.7x. In RS(6,3), the improvement becomes even more (6.8x) because the NIC is divided by 6 downloading streams.

SSD (40GbE). Fig. 5b shows results in an SSD and 40GbE setting, where Java socket I/O becomes an extra bottleneck. A single Java socket can not exceed 4 Gb/s throughput, while multiple sockets can achieve larger aggregate throughput. Native EC downloads from 3 or 6 servers and achieves 4 Gb/s reconstruction throughput, bounded by the Java socket limit from the sending side. On the other hand, to show that NetEC is capable of reaching 8 Gb/s reconstruction rate (middle and right bars in Fig. 5b), saturating the write speed of our SSD (1 GB/s), we let NetEC uses two concurrent tasks (two Java Sockets), with a network usage of 9.3 Gb/s. Meanwhile, Native EC consumes 11.3 Gb/s and 18.6 Gb/s network throughput, significantly higher than NetEC. Although Native EC implementation uses one socket by default and we let NetEC use two sockets, we still demonstrate the advantages of NetEC because NetEC achieves better reconstruction speed with less network usage than Native EC.

7.2.2 CPU Utilization

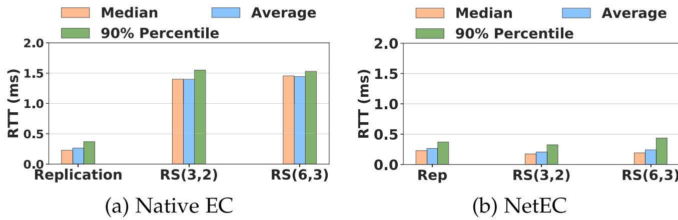
We enable Intel ISA-L [22] in HDFS to achieve higher reconstruction rate and do not constrain CPU core usage of HDFS, so it grasps as many cores as possible, leading to utilization higher than 100%. Fig. 5c shows CPU utilization during reconstruction on an HDD at its top writing speed (170 MB/s). CPU utilization can be as high as 350%. Note that replication also incurs moderate CPU overheads for transactional operations [35], but the additional computational overheads of EC are eliminated by NetEC.

7.2.3 Degraded Read Latency

In Fig. 6, we measure the reconstruction time of a single block to reflect degraded read latency, as in [36]. In Figs. 6a and 6b, we show that NetEC achieves significantly shorter single block reconstruction time compared to Native EC, with or without incast. In Fig. 6c, we show that NetEC single block reconstruction time is always roughly the same with replication, no matter in RS(3,2) or RS(6,3).

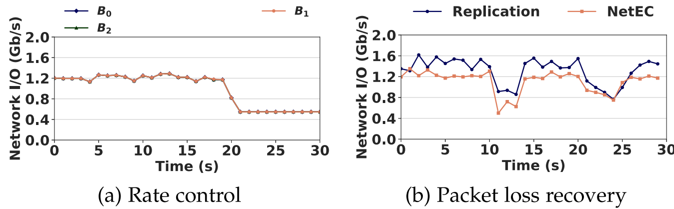
Incast. We measure the RTT with `ping` between the reconstructing node and other nodes to reflect incast level in Fig. 7. We vary the stripe width and observe that the RTT

Authorized licensed use limited to: Tsinghua University. Downloaded on April 06,2022 at 15:20:15 UTC from IEEE Xplore. Restrictions apply.



(a) Native EC (b) NetEC

Fig. 7. Incast level reflected by RTT.



(a) Rate control (b) Packet loss recovery

Fig. 8. TCP EBSN effects.

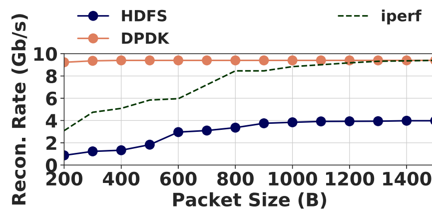


Fig. 9. Packet size impact.

increases with the stripe width in Native EC. For NetEC, however, the RTT remains the same.

7.2.4 TCP EBSN Effects

We measure the reconstruction rate in face of link capacity changes and packet losses to show that EBSN introduced in Section 4.3 is effective.

Rate Control. To measure rate control effects, we use Linux `tc` to adjust available output bandwidth of one of the survived nodes. The evaluation is conducted under HDD setting with 10GbE NICs. Fig. 8a shows that when one node undergoes rate-limiting, the other nodes respond almost instantly. The sending rates of all nodes remain equal after rate limiting that happens at around 19 second.

Packet Loss. For packet loss, we use replication as baseline and compare NetEC to it with the same end host TCP setting. We manually introduce random packet losses with Linux `tc` for three seconds twice with different loss percentage. As shown in Fig. 8b, we see that NetEC behaves similarly with replication in face of packet losses. The throughput is lowered because of the constant packet loss rate, but the flow is not disrupted.

7.3 Performance and Scalability

Recon. Rate versus Packet Size. We demonstrate that NetEC can achieve high reconstruction rate by supporting large packet size. As shown in Fig. 9, the packet size does not affect throughput in DPDK Testbed because of high packet I/O performance. In HDFS Testbed, we use `ifconfig` to adjust the MTU of the interface to control the packet size. The measured reconstruction rate increases with the packet

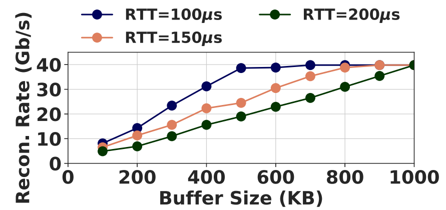


Fig. 10. Buffer size impact.

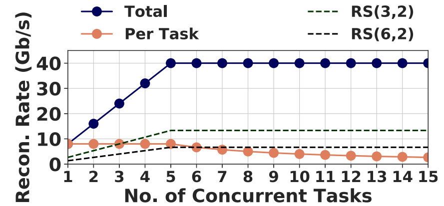


Fig. 11. Concurrent recon. tasks.

size, because small packets bring large overheads to kernel stacks and Java socket I/O. Nevertheless, we note that even in HDFS Testbed, when we use packet size larger than 600B, the throughput already reaches 80% of the maximum throughput of Java socket. The dashed line is the end-to-end TCP throughput measured with a single `iperf` TCP stream. It shows that transmission rate with kernel TCP stack is also affected by the packet size, but the throughput drop is less severe than Java Socket. This is probably because Java I/O libraries have another layer of bottleneck on top of kernel stacks.

Recon. Rate versus Allocated Buffer Size. We demonstrate that NetEC constrains buffer usage while achieving high reconstruction rate with NetEC simple version in DPDK testbed. Fig. 10 shows that before reaching maximum rate (40 Gb/s), the reconstruction rate R and the allocated buffer size B are roughly proportional, i.e., $\frac{R}{B} \approx RTT$. The raw RTT measured with `ping` in our cluster is 100 μs. We let NetEC clients intentionally delay all ACK packets by 50μs and 100μs to emulate 150 μs and 200 μs RTTs.

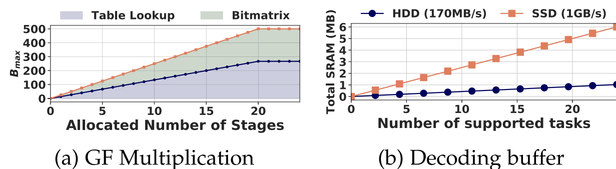
Impact of Number of Concurrent Tasks. We demonstrate that NetEC supports multiple concurrent tasks. We emulate a rack-level failure recovery process by starting N concurrent reconstruction tasks to one machine from other machines. We rate-limit each flow to be 1 GB/s, emulating SSD sequential write. As shown in Fig. 11, if $N \leq 5$, the total reconstruction rate grows linearly with N. If $N > 5$, the output NIC bandwidth (40 Gb/s) becomes the bottleneck. Therefore, the reconstruction rate per task is roughly $\frac{40}{N}$ Gb/s. The total reconstruction rate remains 40 Gb/s as N increases. The dashed lines show the total reconstruction rate of $RS(6,3)$ and $RS(3,2)$, which is roughly $\frac{1}{6}$ and $\frac{1}{3}$ of NetEC.

7.4 Switch SRAM Consumption

NetEC SRAM usage consists of two parts as shown in Fig. 12. The GF multiplication part occupies only part of the available stages to support an acceptable B_{max}, and the decoding buffer part requires less than 10MB to support tens of reconstruction tasks. NetEC can thus co-locate with other essential network functionalities on the same switch.

GF Multiplication Offloading. We show in Fig. 12a that the number of bytes on which to perform GF multplication is

Authorized licensed use limited to: Tsinghua University. Downloaded on April 06,2022 at 15:20:15 UTC from IEEE Xplore. Restrictions apply.



(a) GF Multiplication (b) Decoding buffer

Fig. 12. Switch SRAM usage.

linear to the number of allocated stages, according to the constraints described in Section 5.3.2.

Decoding Buffers. In Fig. 12b, the total decoding buffer size is linear to the number of supported tasks. As shown in Section 4.1, we allocated 500KB for each SSD reconstruction task (1GB/s). Similarly, HDD (170MB/s) requires 86KB per task.

8 Discussion

Comparison With Other Hardware Platforms. Other hardware platforms (e.g., FPGA, programmable NIC) can also be used for acceleration of intensive applications. NetEC prefers programmable switching ASICs for the following two reasons. First, programmable switching ASIC performs processing on switches rather than hosts. Most platforms perform processing on hosts and the three performance problems (long reconstruction time, high degraded read latency and heavy resource usage) remain. Some works augment switches with FPGA modules to avoid the performance problems, but goodput is limited by the inbound I/O of the FPGA chips. Second, other hardware might encounter performance degradation when throughput is large. For switch ASICs, all computation tasks are conducted at line-rate and throughput is not affected. The processing delay is bounded by the constant total pipeline traversal time around hundreds of nanoseconds [16].

Multi-Switch Schemes. In this paper, we only focus on single-switch design, where all downloading streams are aggregated on a single switch. However, NetEC can potentially be extended to leverage multiple switches. We design a tree-pattern in-network reconstruction scheme similar to PPR[37], where low-level switches aggregate a subset of streams and high-level switches combine the results generated by the low-level switches. In this way, NetEC can make full use of the processing power of rack-level programmable switches, and reduce the bandwidth usage between different levels of switches. Also, multiple switch schemes can handle scenarios where some nodes are co-located under a same rack. We are currently working in progress on the multiple switch schemes.

9 Related Work

Accelerating Erasure Coding Reconstruction. Some approaches propose new coding mechanisms, e.g., PM-RBT [38] and PM-MSR [2], while others aim at building more responsive and flexible systems, such as HACFS [7] and RAFI [39]. OpenEC [40] and SelectiveEC [41] propose frameworks with better management and scheduling. Some works, such as D3 [42] and PDL [43] propose more efficient data placement to reduce network usage. Repair Pipelining [36] achieves reconstruction rates close to NetEC through pipelining the repair of failed data in small-size units across storage nodes. PPR [37] decompose EC calculations into sub-calculations that can be executed in parallel on multiple nodes. TriEC [44] and INEC [45] leverage smartNIC offloading to accelerate EC calculations. TriEC [44] proposes a new EC offload paradigm and INEC proposes a set of coherent in-network EC primitives on smartNICs.

In-Network Computation and Aggregation. Among numerous works that offloads application to programmable switches [12], [15], [16], [17], [46], some works have focused on in-network aggregation. XORInc [47] has similar motivation with this paper, performing XOR on software switches and has simulation-based experiments, while NetEC also considers GF multiplication and targets at real hardwares. SwitchML [15] and ATP [48] aim to accelerate deep learning applications with in-network aggregation.

10 Conclusion

We present NetEC, an in-network accelerating framework that fully offloads EC to programmable switching ASICs. We propose Explicit Buffer Size Notification (EBSN) to constrain decoding buffer usage, and design an on-switch one-to-many TCP proxy to integrate EBSN with TCP. We also design two parallel Galois Field (GF) offloading methods to maximize parsable bytes. Extensive evaluations show that NetEC is effective and is able to support ~GB/s reconstruction rate and tens of concurrent tasks.

References

[1] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," *ACM SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 29–43, 2003.
[2] K. V. Rashmi, N. B. Shah, and P. V. Kumar, "Optimal exact-regenerating codes for the MSR and MBR points via a product-matrix construction," *IEEE Trans. Inf. Theory*, vol. 57, no. 8, pp. 5227–5239, Aug. 2011.
[3] C. Huang *et al.*, "Erasure coding in windows azure storage," in *Proc. USENIX Annu. Tech. Conf.*, 2012, pp. 15–26.
[4] M. Sathiamoorthy *et al.*, "XORing elephants: Novel erasure codes for big data," *Proc. VLDB Endowment*, vol. 6, no. 5, pp. 325–336, 2013.
[5] Apache hadoop, 2018. [Online]. Available: https://hadoop.apache.org/
[6] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *J. Soc. Ind. Appl. Math.*, vol. 8, no. 2, pp. 300–304, 1960.
[7] M. Xia, M. Saxena, M. Blaum, and D. Pease, "A tale of two erasure codes in HDFS," in *Proc. USENIX Conf. File Storage Technol.*, 2015, pp. 213–226.
[8] 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," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, pp. 331–342, 2015.
[9] M. Alizadeh *et al.*, "CONGA: Distributed congestion-aware load balancing for datacenters," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, pp. 503–514, 2014.
[10] Keynote in NetCompute18, 2018. [Online]. Available: http://conferences.sigcomm.org/sigcomm/2018/files/slides/netcompute/2018–08-20-sigcomm.pdf
[11] D. Wu, A. Chen, T. E. Ng, G. Wang, and H. Wang, "Accelerated service chaining on a single switch ASIC," in *Proc. 18th ACM Workshop Hot Top. Netw.*, 2019, pp. 141–149.
[12] X. Jin *et al.*, "NetCache: Balancing key-value stores with fast in-network caching," in *Proc. 26th Symp. Oper. Syst. Princ.*, 2017, pp. 121–136.
[13] N. Gebara *et al.*, "PANAMA: Network architecture for machine learning workloads in the cloud," Tech. Rep., 2020. [Online]. Available: https://people.csail.mit.edu/ghobadi/papers/panama.pdf
[14] X. Jin *et al.*, "Netchain: Scale-free sub-RTT coordination," in *Proc. 15th USENIX Symp. Netw. Syst. Des. Implementation*, 2018, pp. 35–49.

Authorized licensed use limited to: Tsinghua University. Downloaded on April 06,2022 at 15:20:15 UTC from IEEE Xplore. Restrictions apply.

[15] A. Sapio *et al.*, "Scaling distributed machine learning with in-network aggregation," 2019, *arXiv:1903.06701*.
[16] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "SilkRoad: Making stateful layer-4 load balancing fast and cheap using switching ASICs," in *Proc. Conf. ACM Special Int. Group Data Commun.*, 2017, pp. 15–28.
[17] J. Sonchack, O. Michel, A. J. Aviv, E. Keller, and J. M. Smith, "Scaling hardware accelerated network monitoring to concurrent and dynamic queries with* flow," in *Proc. USENIX Conf. USENIX Annu. Tech. Conf.*, 2018, pp. 823–835.
[18] P. Bosshart *et al.*, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 43, pp. 99–110, 2013.
[19] J. S. Plank, "A tutorial on reed–solomon coding for fault-tolerance in raid-like systems," *Softw.: Pract. Experience*, vol. 27, no. 9, pp. 995–1012, 1997.
[20] HDFS erasure coding, 2018. [Online]. Available: https://hadoop.apache.org/docs/r3.1.1/hadoop-project-dist/hadoop-hdfs/HDFSErasureCoding.html
[21] M. Alizadeh *et al.*, "Data center TCP (DCTCP)," in *Proc. ACM SIGCOMM Conf.*, 2010, pp. 63–74.
[22] Intel Intelligent Storage Acceleration Library (Intel ISA-L), 2018. [Online]. Available: https://software.intel.com/en-us/storage/ISA-L
[23] P. Bosshart *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, pp. 87–95, 2014.
[24] Barefoot tofino, 2018. [Online]. Available: https://www.barefootnetworks.com/technology/#tofino
[25] Cavium xpliant, 2018. [Online]. Available: https://www.marvell.com/documents/netpxrx94dcdhk8sksbp/?x=2,2015
[26] H. Giesen *et al.*, "In-network computing to the rescue of faulty links," in *Proc. Morning Workshop In-Netw. Comput.*, 2018, pp. 1–6.
[27] J. S. Plank, S. Simmerman, and C. D. Schuman, "Jerasure: A library in C/C++ facilitating erasure coding for storage applications," Univ. Tennessee, Knoxville, TN, USA, Version 1.2. Tech. Rep. CS-08–627, 2008.
[28] T. Zhou and C. Tian, "Fast erasure coding for data storage: A comprehensive study of the acceleration techniques," in *Proc. USENIX Conf. File Storage Technol.*, 2019, pp. 317–329.
[29] J. Bloemer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman, "An XOR-based erasure-resilient coding scheme," 1995.
[30] S. Chole *et al.*, "dRMT: Disaggregated programmable switching," in *Proc. Conf. ACM Special Int. Group Data Commun.*, 2017, pp. 1–14.
[31] A. Sivaraman *et al.*, "Packet transactions: High-level programming for line-rate switches," in *Proc. ACM SIGCOMM Conf.*, 2016, pp. 15–28.
[32] T. Jepsen *et al.*, "Fast string searching on PISA," in *Proc. ACM Symp. SDN Res.*, 2019, pp. 21–28.
[33] Source Code, 2020. [Online]. Available: https://github.com/netec-2020/netec
[34] Data Plane Development Kit, 2018. [Online]. Available: https://www.dpdk.org/
[35] D. Kim *et al.*, "Hyperloop: Group-based NIC-offloading to accelerate replicated transactions in multi-tenant storage systems," in *Proc. Conf. ACM Special Int. Group Data Commun.*, 2018, pp. 297–312.
[36] R. Li, X. Li, P. P. Lee, and Q. Huang, "Repair pipelining for erasure-coded storage," in *Proc. USENIX Annu. Tech. Conf.*, 2017, pp. 567–579.
[37] S. Mitra, R. Panta, M.-R. Ra, and S. Bagchi, "Partial-parallel-repair (PPR) a distributed technique for repairing erasure coded storage," in *Proc. 11th Eur. Conf. Comput. Syst.*, 2016, pp. 1–16.
[38] K. Rashmi, P. Nakkiran, J. Wang, N. B. Shah, and K. Ramchandran, "Having your cake and eating it too: Jointly optimal erasure codes for I/O, storage, and network-bandwidth," in *Proc. USENIX Conf. File Storage Technol.*, 2015, pp. 81–94.
[39] J. Fang, S. Wan, and X. He, "RAFI: Risk-aware failure identification to improve the RAS in erasure-coded data centers," in *Proc. USENIX Annu. Tech. Conf.*, 2018, pp. 495–506.
[40] X. Li, R. Li, P. P. Lee, and Y. Hu, "OpenEC: Toward unified and configurable erasure coding management in distributed storage systems," in *Proc. 17th USENIX Conf. File Storage Technol.*, 2019, pp. 331–344.
[41] L. Xu, M. Lyu, Q. Li, L. Xie, and Y. Xu, "SelectiveEC: Selective reconstruction in erasure-coded storage systems," in *Proc. 12th USENIX Workshop Hot Top. Storage File Syst.*, 2020, Art. no. 8.
[42] Z. Li, M. Lv, Y. Xu, Y. Li, and L. Xu, "D3: Deterministic data distribution for efficient data reconstruction in erasure-coded distributed storage systems," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2019, pp. 545–556.
[43] L. Xu, M. Lv, Z. Li, C. Li, and Y. Xu, "PDL: A data layout towards fast failure recovery for erasure-coded distributed storage systems," in *Proc. IEEE Conf. Comput. Commun.*, 2020, pp. 736–745.
[44] H. Shi and X. Lu, "TriEC: Tripartite graph based erasure coding NIC offload," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2019, pp. 1–34.
[45] H. Shi and X. Lu, "INEC: Fast and coherent in-network erasure coding," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2020, pp. 1–17.
[46] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One sketch to rule them all: Rethinking network flow monitoring with UnivMon," in *Proc. ACM SIGCOMM Conf.*, 2016, pp. 101–114.
[47] F. Wang, Y. Tang, Y. Xie, and X. Tang, "XORInc: Optimizing data repair and update for erasure-coded systems with XOR-based in-network computation," in *Proc. 35th Symp. Mass Storage Syst. Technol.*, 2019, pp. 244–256.
[48] C. Lao *et al.*, "ATP: In-network aggregation for multitenant learning," in *Proc. 18th USENIX Symp. Netw. Syst. Des. Implementation*, 2021, pp. 741–761.

Yi Qiao received the BS degree in computer science and technology from Tsinghua University, Beijing, China. He is currently working toward the PhD degree in the Institute of Network Science and Cyberspace, Tsinghua University, Beijing, China. His research focuses on software-defined networking, network function virtualization, and cyber security.

Menghao Zhang (Graduate Student Member, IEEE) received the BS and PhD degrees in computer science from Tsinghua University, Beijing, China, in 2016 and 2021, respectively. He is now a joint postdoctoral with Tsinghua University and Kuaishou Technology. His research interests include programmable network, high-performance network, and network security.

Yu Zhou (Member, IEEE) received the BS degree from the School of Information and Communication Engineering, Beijing University of Posts and Telecommunications, Beijing, China, in 2016. He is currently working toward the PhD degree in the Institute for Network Sciences and Cyberspace, Tsinghua University, Beijing, China. His research interests include software-defined networking and programmable data planes.

Xiao Kong received the BS degree in computer science and technology from Nankai University, Tianjin, China. He is currently working toward the MS degree in the Institute for Network Science and Cyberspace, Tsinghua University, Beijing, China. His research focuses on software-defined networking, network function virtualization, and cyber security.

Authorized licensed use limited to: Tsinghua University. Downloaded on April 06,2022 at 15:20:15 UTC from IEEE Xplore. Restrictions apply.

Han Zhang (Member, IEEE) received the BS degree in computer science and technology from Jilin University, Changchun, China, and the PhD degree from Tsinghua University, Beijing, China. He is now working with the Institute for Network Sciences and Cyberspace, Tsinghua University. His research concerns computer network systems and network security.

Mingwei Xu (Senior Member, IEEE) received the BS and PhD degrees from Tsinghua University, Beijing, China. He is currently a full professor with the Department of Computer Science and Technology, Tsinghua University. His research interests include computer network architecture, high-speed router architecture, and network security.

Jun Bi received the BS, CS, and PhD degrees from the Department of Computer Science, Tsinghua University, Beijing, China. He is currently a Changjiang Scholar distinguished professor with Tsinghua University and the director with the Network Architecture Research Division, Institute for Network Sciences and Cyberspace, Tsinghua University. His current research interests include Internet architecture, SDN/NFV, and network security.

Jilong Wang received the PhD degree from the Department of Computer Science and Technology, Tsinghua University, Beijing, China, in 2000. He is currently a professor with Tsinghua University. His research focuses on network measurement, location-oriented network, SDN systems, and network security.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.

Authorized licensed use limited to: Tsinghua University. Downloaded on April 06,2022 at 15:20:15 UTC from IEEE Xplore. Restrictions apply.