Towards In-network Acceleration of Erasure Coding

Yi Qiao[★°], Xiao Kong[★°], Menghao Zhang[★†°], Yu Zhou[★†°], Mingwei Xu[★†°], Jun Bi[★°]
[★]Institute for Network Sciences and Cyberspace, Tsinghua University
[†]Department of Computer Science and Technology, Tsinghua University
[°]Beijing National Research Center for Information Science and Technology (BNRist)

ABSTRACT

In distributed storage systems, erasure coding (EC) is a crucial technology to enable high fault tolerance with lower storage overheads than data replication. EC can reconstruct missing data by downloading parity data from survived machines. However, downloading streams of EC multiplex the available network I/O on the receiving end, leading to a substantially low data reconstruction speed. In this paper, we present NetEC, a novel in-network accelerating system that fully offloads EC to programmable switching ASICs. NetEC prevents multiplexing network I/O through on-switch downloading stream aggregation, thus significantly improving reconstruction speed. NetEC addresses three key challenges: computation offloading of complex EC operations, rate synchronization of multiple downloading streams, and deep payload inspection/assembly. We implement NetEC on hardware programmable switches. Evaluation shows that compared to HDFS-EC, NetEC significantly improves reconstruction rate by 2.7x-9.0x and eliminates CPU overheads, with low switch memory usage.

CCS CONCEPTS

• **Networks → In-network processing**;

KEYWORDS

Programmable switches; Erasure coding

ACM Reference Format:
Yi Qiao, Xiao Kong, Menghao Zhang, Yu Zhou, Mingwei Xu, Jun Bi. 2020. Towards In-network Acceleration of Erasure Coding. In *Symposium on SDN Research (SOSR '20), March 3, 2020, San Jose, CA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3373360.3380833

1 INTRODUCTION

Many large-scale distributed storage systems are transitioning to *erasure coding (EC)* [1, 10, 11] to provide high availability with much lower storage overheads than data replication. Reed-Solomon (RS) code [23] is one of the most popular choices of EC. An RS(k, r) code encodes k units of data into r units of parities. RS(k, r) reconstructs the original k units from any k out of (k + r) units of data, thus tolerating any r failures. For example, RS(10,4) can tolerate any 4 failures with 1.4x storage cost, while replication-based systems requires 3x storage cost to achieve the similar degree of availability. However, as revealed by many previous literature [11, 24, 30], EC trades storage cost with extra performance overheads. In particular, *low reconstruction rate* is one of the most significant problems, receiving great attention from both academia and industry [11, 21, 24, 30].

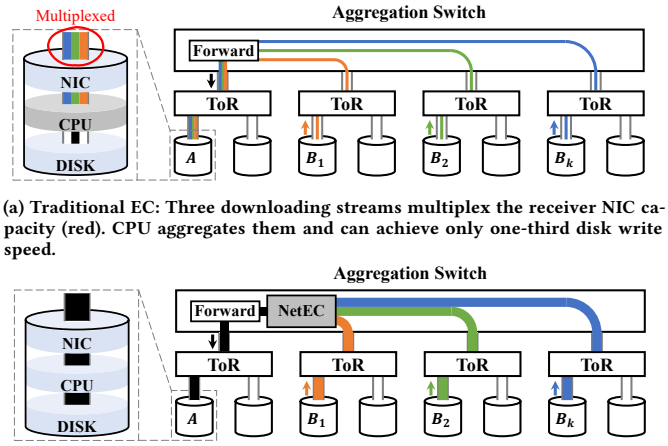


Figure 1: NetEC Overview and Comparisons with Traditional EC. (Colored lines represent downloading stream, and black lines represent reconstructed stream. The line width represents throughput.)

The fundamental problem behind low reconstruction rate is *the proportionate goodput*, which is unavoidable in all current systems based on end-hosts. Reconstructing one block of data in RS(k, r) requires downloading k blocks from other nodes. The available network I/O (typically the NIC capacity) on the receiver side is *multiplexed* by k downloading data streams so that the effective reconstruction *goodput* is no larger than $1/k$ of the available network I/O. As shown in Figure 1a, the three colored data streams share the available NIC bandwidth, and the actual reconstruction goodput (disk write speed) is only *one-third* of the network I/O. The proportionate goodput problem cannot be completely resolved as long as processing is done on end-hosts, including FPGAs or SmartNICs, because they need NICs to connect to networks.

Therefore, we resort to the in-network computation paradigm to resolve the proportionate goodput problem. On a programmable switch, data streams arrive at *different* interfaces, get aggregated and forwarded to yet *another* interface. There is no sharing of bandwidth. In Figure 1b, where computation (gray box) is moved

SOSR '20, March 3, 2020, San Jose, CA, USA

ACM ISBN 978-1-4503-7101-8/20/03…$15.00
https://doi.org/10.1145/3373360.3380833

from CPU to the network, the reconstructed data (black) is able to make full use of the entire bandwidth available on NIC, so that we can acheive higher disk write rate. The wider black line compared to Figure 1a indicates the improvement of reconstruction rate.

Another benefit of in-network computation is to relieve heavy CPU utilization of EC. During reconstruction, CPU has to inspect *every* single byte from *all k* downloading streams. Storage devices waste many CPU cores for reconstruction, leading to high capital expenditure [12]. In-network computation completely removes extra CPU usage (see §6.3).

In this paper, we present NetEC, an in-network acceleration framework that fully offloads EC to programmable switching ASICs. Nevertheless, offloading EC is non-trivial, and NetEC should address three new challenges:

Computation offloading. RS code relies on Galois Field (GF) arithmetics (see §2 and §4.1). To fully offload GF computation to less expressive switching ASICs, we design an *RS codec engine* that transforms GF vector dot-product to table lookups and partial XOR sum updates. All arriving packets perform GF multiplication on extracted payloads and update the buffered partial XOR sums. When reconstruction completes, the last arrived packet is sent with the decoding result encapsulated as its payload.

Rate synchronization. If packets from different sources arrive at different rates, the switch memory consumption would increase drastically, because NetEC needs to buffer partial decoding results temporarily. We build a *one-to-many TCP proxy* to synchronize arriving rates of different TCP downloading streams. NetEC only needs to buffer partial XOR sums whose size is equal to in-flight packets, which is upper bounded by TCP receive window size (see §4.2).

Deep payload inspection/assembly. Limited number of bytes can be parsed and processed on switch because of current hardware resource constraints. However, small-sized packets lead to performance penalties. To support larger packet size with current parser constraints, we design a *packet recirculator* to not only inspect deep packet payloads [12], but also assemble a completely new packet with correct header checksums. We discuss the recirculation penalties in §4.3.

In summary, we make the following contributions:

- We propose NetEC that offloads EC to switching ASICs, which fundamentally resolves the proportionate goodput problem and relieves CPU overheads (§3).
- We design the RS codec engine, the one-to-many TCP proxy and the packet recirculator to address three challenges (§4).
- We implement NetEC on programmable switches and integrate it with Hadoop Distributed File System (HDFS). Evaluation shows that reconstruction rate is improved by 2.7x-9.0x with low switch memory consumption, and the CPU overhead is completely removed (§6).

We discuss incast and scalability issues in §5. NetEC will not face incast risks and has the potential to scale. Finally, §7 shows related works, and §8 concludes the paper.

2 BACKGROUND

We introduce Reed-Solomon (RS) code in this part and show that encoding and decoding can both be modelled with vector dot-products over Galois Fields (GF). 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 16-bit word.

Encoding. We describe the RS(k, r) coding system using a matrix-vector product[19]. The RS(k, r) code encodes k symbols into r parity symbols with a $(k + r) \times r$ *generating matrix* R (Equation (1)), composed of an identity matrix and a *redundancy matrix*. We use column vector D to denote a set of original data symbols $d_1, d_2, ..., d_k$, and C to denote data and parity symbols $d_1, ..., d_k, p_1, ..., p_r$. 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.

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

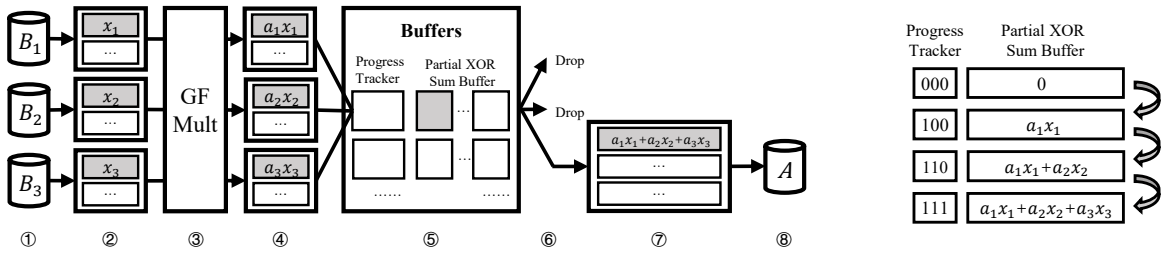
Decoding. RS(k, r) tolerates at most r lost data words through reconstruction using (any) k of the survived symbols. To reconstruct, we build a $k \times k$ *reconstruction matrix* R' by deleting r rows corresponding to lost data symbols from generating matrix D. Applying the same row-deletions on C, 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 .

3 OVERVIEW

Terms. we refer to the node on which a data symbol m is reconstructed as the *reconstructing node* (termed A, ⑧). m is reconstructed from the *symbol-stripe* $x_1, x_2, ..., x_k$ downloaded from *survived nodes* $B_1, B_2, ..., B_k$ (①). To simplify, we term all survived nodes as *Bs*. 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).

Dataplane workflow. For every incoming packet (②), the switch extracts symbols and perform GF multiplication on them (③, detailed in §4.1). The results are written back in the extracted header fields (④). The switch selects a slot in the partial XOR sum buffers, implemented with registers, and XOR the symbols with buffer contents. Slot indexing is shown in §4.2 in detail. The switch also updates the progress tracker, whose i-th bit is flipped if the packet comes from the i-th survived node. As other packets of the packet-stripe arrive, the buffer content changes as shown in Figure 2b. 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



(a) Workflow. **(b) Slot Content Changes.**

Figure 2: NetEC Data Plane.

the whole payload of the last arrived packet with the finished XOR sums (⑦), and forwards it to A (⑧).

4 DESIGN

In this section, we present detailed design for three NetEC modules, each corresponding to a challenge described in §1.

4.1 In-Network RS Codec Engine

GF vector dot products. As is shown in §2, the encoding and decoding process are both modelled with GF vector dot products, which can be transformed to table lookups and partial XOR sum updates.

To illustrate how to perform GF vector dot products, 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 a survived nodes Bs and a_i are pre-computed coefficients. Packets containing x_i are sent from corresponding nodes and arrive in arbitrary order at the RS codec engine, and each x_i is extracted as a metadata field. To multiply x_i with a_i, we have to lookup $\log x_i$ in a pre-installed logarithm table and add $\log x_i$ with a pre-known $\log a_i$: $\log x_i + \log a_i = \log x_i a_i = \log m_i$. Note that the addition here is integer addition. Then we get m_i by looking up m_i in an exponent table. m_i is then XOR-ed with the partial XOR sum of already arrived symbols. Commutativity and associativity of XOR ensures that the arrival order is irrelevant and m is computed after all k symbols arrive. Finally, m is contained as payload of the last arrived packet and forwarded to the reconstructing node.

Logarithm and exponent tables. The logarithm table and exponent table record 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 values have w-bit width. In NetEC, we choose $w = 16$ to be the word size, so the logarithm table takes up 131072 bytes. The exponent table takes up twice as much as logarithm table because the integer sum of two 16-bit integers ranges from 0 to 131070. However, this can be optimized to be the same as logarithm table because the exponent table has repeated contents [19].

SRAM consumption for lookup tables. In current switching ASICs, each match table 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. Suppose we allocate n stages and each stage contains M SRAM. We denote N as the number of processed symbols in one pass, then we have $N \cdot lookup\_table\_size \leq n \cdot M$. Therefore, we can either allocate more SRAM for NetEC to acheive more processed symbols, or allocate less SRAM, yet affecting end-to-end throughput due to small-sized packets.

In reality, our choice of $N = 160B$[1] can be affected by other constraints. Each stage has limited number of sALUs (stateful ALUs) to access registers [27], resulting in limited decoding buffers, thus less N. Also, multiplication of a symbol must be done prior to XOR calculation, so the multiplication lookup tables reside in stages before the stages that hold XOR decoding buffers. .

Note that recent works discuss possible methods to relieve the memory access constraint. dRMT [8] builds a memory pool accessible through a crossbar, and GEM [25] proposes RDMA-based external memory. Also, we can choose to conduct multiplication on end-hosts. In this way, NetEC can still resolve the "proportionate goodput" problem (§1), thus increasing reconstruction speed and removing receiver-side CPU usage. However, the senders will have extra CPU usage for multiplication.

4.2 One-to-many TCP Proxy

Asynchronized packet arrival leads to unpredictable buffer size. Since the first packet of the stripe arrives, the RS codec engine starts to buffer the partial XOR sums in switch memory until the whole stripe is received. Ideally, if packets of the same stripe arrive at the same time, the aggregation is performed almost instantly and no buffer is used. However, if packets arrive *unpaced*, the memory consumption will increase drastically.

Our insight here is to leverage the rate control mechanisms of TCP. TCP already handles rate control with very mature mechanisms. More importantly, TCP ensures that in-flight packets does not exceed receive window size. This provides an upper bound for switch memory consumption.

One-to-many TCP proxy. In NetEC, the survived node Bs shares a *virtual IP* (VIP) and the reconstructing node A connects with this VIP. Packets from A are multicast to several end-hosts. Except hand-shake packets, packets from A only have pure ACKs (ACK without payload). The one-to-many TCP proxy changes the destination address of these packets and multicast them to corresponding outbound interfaces. Packets from Bs are data packets containing NetEC payloads. These packets are processed and aggregated by the RS codec engine. The produced packet gets its network addresses modified and checksum calculated, and is forwarded to A.

[1]The major constraint is actually the limited number of sALUs per stage. It limits the number of registers that decoding buffers can use, and thus limits N.

Translation of TCP SEQ/ACK and IP ID. TCP handshake is performed in the same way. SYN is multicast to *Bs* and SYN-ACK is sent to *A* after receiving all SYN-ACKs from *Bs*. However, different *Bs* provide different Initial Sequence Numbers (ISN) [6] during handshake and the switch chooses its own ISN to interact with *A*. These ISNs should be recorded to translate sequence and acknowledge numbers when relaying subsequent packets. After translation (simple addition and subtraction), the SEQ/ACK number is correct from each end-host's perspective. If TCP SACK (Selective Acknowledgement) [5] is employed, NetEC will have to apply similar translation to SACK option fields. IP identification translation is also needed to support end-host TCP segmentation offload for better throughput.

Rate control and Packet Loss Recovery NetEC prevents senders from overwhelming the receiver and maintains synchronicity of sending rates. From *A*'s perspective, it receives packet as if from the slowest one of the k senders, and replies ACK with the slowest rates. *Bs* behave equally in response to receive buffer changes and will never send faster than *A* can receive. NetEC also tolerates packet losses. Lost packets on path from *A* to *S* and *S* to *Bs* have negligible effects as long as any following ACK packet is successfully transmitted. Lost packets on path from *S* to *A* would result in either duplicate ACKs or timeouts, triggering (fast) re-transmissions. Packet loss on path from *Bs* and *S* can be equivalent to the last scenario, because *S* would not send packets to *A* until all k packets have arrived.

SRAM consumption for decoding buffers. Switch memory holds all unfinished partial XOR sums. This is equal to all in-flight (sent yet not acknowledged) packets of the fastest sender. TCP ensures that in-flight packets will not exceed TCP receive window size. Therefore, the maximum memory consumption is upper bounded by the receiver window size. The receive window size can be manually configured, and will not affect performance as long as it exceeds the bandwidth-delay product (BDP) of the link. In a high throughput, low latency data center cluster, BDP is usually below 100KB, which is manageable with on-chip memory (0.1% in a switch with 100MB total SRAMs).

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 i-th packet sent by *Bs* has the slot index $i \pmod M$. As long as the total SRAM size is configured larger than TCP receive window size, the buffer will never be full, and there will not be collisions of slot usage.

4.3 Packet Recirculator

Current switch hardware can parse limited number of bytes, while NetEC requires processing the entire payload. We design a packet recirculator to not only inspect deeper payloads but also assemble a packet with completely new payloads under the parser limit. We configure end-host Maximum Transmission Unit (MTU) to be $B+H$, where H stands for TCP header size (including TCP options), so that each packet has B size payloads. For each incoming packet, we parse and process the N bytes of packets in the ingress pipeline and truncate these bytes by not emitting them. Then the packet is loop-backed to the ingress pipeline again to parse the next N bytes. After $\frac{B}{N}$ passes, we can inspect the whole payload of the packet. Most incoming packets will be dropped except the last packet in a stripe. The last packet, however, has to go through the pipeline again to retrieve and emit finished XOR sums from corresponding slots. After another $\frac{B}{N}$ passes, the packet is assembled and leaves the switch.

Recalculating L4 checksums. The outgoing packets have completely re-written payloads. Although the switch provides a primitive *CSUM* for IP checksum recalculation and L4 checksum update, we cannot directly use it because we assemble packets by recirculations. In NetEC, we first calculate L4 pseudo header with $checksum = CSUM(l4\_pseudo\_hdr)$. Then in every new pass when N bytes $Fields$ are emitted, the checksum is updated with the formula: $checksum = CSUM(\sim checksum, Fields)$, where "~" refers to bit-wise negation. This is because *CSUM* calculates the **negation** of the 16-bit ones' complement sum of a specified list of fields. $\sim checksum$ is the ones' complement sum of already emitted fields. Feeding it with new fields to *CSUM* again yields the checksum of all considered fields.

Recirculation penalty. More recirculations lower the switch overall throughput, but support larger packet size, thus better end-to-end throughput. To trade off between overall and end-to-end throughput, we choose $B = 320$ and $N = 160$, so that packet recirculates once. On the one hand, with 320B packet size, the end-to-end throughput can at least saturate SSD write speed (1-2 GB/s). On the other hand, recirculating once does not greatly impact switch throughput [12, 29]. 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 (0.25% for 6.4Tb/s capacity).

5 DISCUSSION

Incast issue. Some may worry that simultaneous arrival of packets will incur incast by overloading the output buffer. The reality is that NetEC does not face this issue. Note that most arriving packets only update the partial XOR sums stored in ingress pipeline and get dropped without even entering the switch forwarding engine. Only after a whole packet stripe arrives, a packet is emitted to the output interface. The *pps* (packet per second) of the output interface is approximately equal to *pps* of the inbound interfaces, so there will not be incast risks.

Potential to scale. In this short paper, we do not explicitly address scalability issue, which we leave as a major focus in future work. Here we discuss the scaling potentials. First, NetEC can support higher reconstruction throughput because the transient memory usage is bounded by the BDP. Second, NetEC can hold multiple concurrent reconstruction tasks, also because of the bounded memory usage, with the help of on-switch dynamic memory allocation [27]. Finally, we emphasize that we do not need to consider more survived nodes, because normally we choose RS(3,2), RS(6,3) or RS(10,4) settings [3], with 3, 6 or 10 survived nodes.

NetEC limitations. Despite many efforts to ensure NetEC performance under resource constraints, there are still following limitations. First, NetEC takes up SRAMs that span through one or several pipeline. Then, in the same pipeline(s), NetEC can only co-locate

with "stateless" functionalities, like basic forwarding and protocol translation, or small-scale "stateful" functionalities. Alternatively, we can dedicate a pipeline to NetEC and route reconstruction traffic to this pipeline, yet sacrificing switch interfaces related to this pipeline. Second, we only discuss trade-offs of recirculation under current SSD settings (~1 GB/s). Recirculation might become a bottleneck for in-memory storage or high speed NVMe. We will explore NetEC application to these systems in our future work.

6 EVALUATION

6.1 Prototype and Testbed

We implement a prototype of NetEC on commodity switches. Data plane components are implemented with P4 [7] (720 LoC) and compiled with Barefoot Capilano software suite [2]. The controller of the data plane program is written with python (269 LoC). On server side, we implement a set of self-defined EC policies with Java (2,587 LoC) by extending the native HDFS-EC module. We modify class `BlockReaderRemote` to let the reconstructing node connect to the programmable switch, instead of to all k survivors. Also, we modify data serialization logic, so that packets from survived nodes are properly indexed and aligned. We build a storage cluster using 9 servers each with two 12-core Xeon E5-2650v3 CPUs, 64GB of memory, 40Gbps NIC, 1TB HDD and 400GB SSD. The servers are directly connected with a Barefoot Tofino programmable switch, which runs NetEC and a standard L2 forwarding module. We build three testbeds: RS(3,2), RS(6,3) and replication, and measure NetEC reconstruction rate, resource usage and other benchmarks.

6.2 Reconstruction and Link Usage

NetEC significantly increases reconstruction rate and decreases link usage. In Figure 3, each bar indicates the total network throughput or reconstruction rate.

We first run NetEC in the replication testbed. All traffic traverses the entire NetEC processing pipeline. The left bars in Figure 3a and Figure 3b shows 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 headers.

Figure 3a shows results in an HDD and 1GbE setting. In RS(3,2) (middle bars), HDFS-EC consumes 1.0 Gb/s network throughput but only achieves 0.31Gb/s reconstruction throughput, 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 throughput, which improves over HDFS-EC by 2.7x. In RS(6,3), the improvement becomes even more (9.0x) because the NIC is divided by more downloading streams.

Figure 3b shows result in an SSD and 40GbE setting, where Java socket I/O becomes the bottleneck. A single Java Socket can not exceed 4 Gbps throughput, while multiple sockets can achieve larger aggregate throughput. NetEC now supports two concurrent reconstruction flows by splitting register usage, reaching 8 Gb/s reconstruction throughput (middle and right bars in Figure 3b). It already saturates the write speed of our SSD (1 GB/s), with a network usage of 9.3 Gb/s. On the other hand, HDFS-EC makes use of multiple sockets for downloading and achieves 4Gb/s reconstruction throughput, bounded by the Java socket limit from the sending side. However, HDFS-EC consumes 11.3 Gb/s and 18.6 Gb/s network throughput, significantly higher than NetEC.

6.3 Micro Benchmarks

CPU utilization. We enable Intel ISA-L [4] in HDFS-EC 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%. Figure 3c shows CPU utilization during reconstruction on an HDD at its top writing speed (180 MB/s). CPU utilization can be as high as 350%, and can be even higher when reconstructing SSD. Note that replication also incurs moderate CPU overheads for transactional operations [16], but the additional computational overheads of EC are eliminated by NetEC.

Dynamic memory usage. We measure the buffer usage of a single reconstruction task by periodically pulling data plane metadata that records the sequence numbers of the latest arrived packet and the latest left (aggregated) packet. The difference between these two value reflects the actual buffer consumption at the moment. Figure 4a demonstrates that the switch memory consumption is low under normal condition (below 50KB), while a programmable switch ASIC typically has 50MB to 100MB SRAM memory. Rate-limiting is introduced at time 3s, and we observe a slight increase followed by a drop in buffer usage. When rate is limited, the sender sends data higher than the receiver can receive, so the in-flight packet increases, leading to the increased buffer usage. The average buffer usage is lower after some time because the lowered rate leads to smaller BDP of the link.

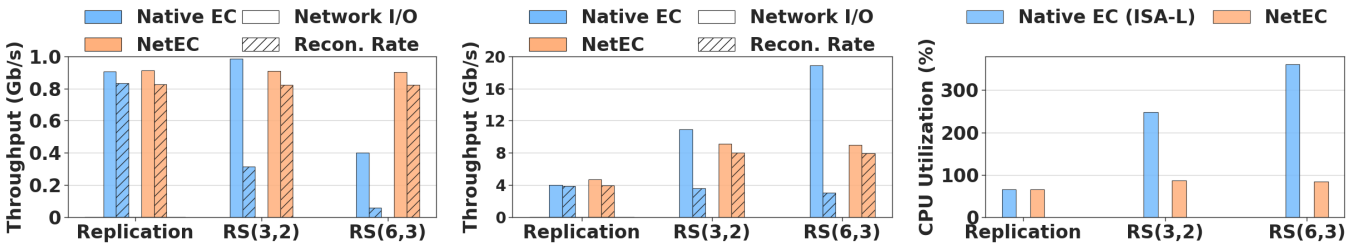
Rate control. To measure rate control effects, we use Linux `tc` to adjust available input bandwidth of one of the survived nodes. The evaluation is conducted under 1GbE HDD setting. Figure 4b shows that when one node undergoes rate-limiting, the other nodes respond almost instantly. The sending rates 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 for three seconds twice with different loss percentage. As shown in Figure 4c, 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 RELATED WORK

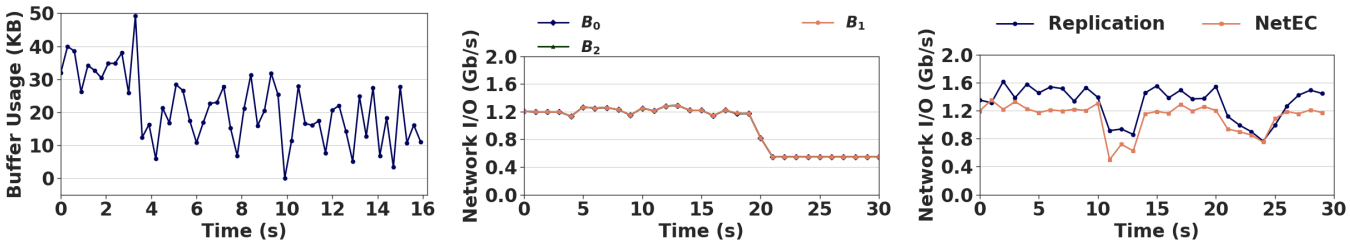
Accelerating erasure coding reconstruction. Many approaches have been proposed to accelerate erasure coding reconstruction from different perspectives. Some new codes are proposed, including rotated-RS [15], PM-RBT[20], PM-MSR[22], etc. Other approaches aim at building more responsive and flexible systems. HACFS[30] uses two different erasure-codes that dynamically adapt to workload changes, and RAFI[9] is a novel risk-aware failure identification scheme.

In-network computation. Researchers have used programmable switches to offload some communication-heavy network applications. SilkRoad [18] achieves fast and cheap L4 load balancing. NetCache [14] utilizes on-switch storage to achieve in-network caching for key-value systems. NetChain [13] accelerates coordination services in distributed systems. *Flow [27], Univmon [17],



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

Figure 3: Reconstruction speed and CPU utilization



(a) Sampled SRAM Usage (b) Rate Control (c) Packet Loss Recovery

Figure 4: One-to-many TCP Proxy

HashPipe [26] are representative network monitoring applications that leverages switching ASICs.

Comparison with XORInc. XORInc [28] has similar motivation with this paper. It performs XOR on switches, while GF computation is still conducted by host CPUs. It is implemented with Open vSwitch and has simulation-based experiments. Compared to XORInc, NetEC considers hardware constraints and resource limitations of real programmable switches, and offloads both GF and XOR computation, achieving higher line rates and elimination of CPU burdens.

8 CONCLUSION

We present NetEC, an in-network EC accelerating framework that fully offloads EC to the new generation programmable switching ASICs to resolve the proportionate goodput issue. We implement NetEC and integrate it into HDFS. Evaluation shows that NetEC significantly improves reconstruction speed and eliminates CPU overheads. We hope our exploration and experience would be valuable for wider adoption of erasure coding and in-network computation in future.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers and our shepherd, Masoud Moshref, for their valuable comments. The research is supported by the National Natural Science Foundation of China under Grant (61625203, 61832013, 61872426), the National Key R&D Program of China under Grant (2017YFB0801701). Mingwei Xu is the corresponding author.

REFERENCES

[1] 2018. Apache Hadoop. (2018). https://hadoop.apache.org/.
[2] 2018. Barefoot Capilano. https://barefootnetworks.com/products/brief-capilano/. (2018).
[3] 2018. HDFS Erasure Coding. https://hadoop.apache.org/docs/r3.1.1/hadoop-project-dist/hadoop-hdfs/HDFSErasureCoding.html. (2018).
[4] 2018. Intel Intelligent Storage Acceleration Library (Intel ISA-L). https://software.intel.com/en-us/storage/ISA-L. (2018).
[5] 2018. RFC2018: TCP Selective Acknowledgment Options. https://tools.ietf.org/html/rfc2018. (2018).
[6] 2018. RFC793: TRANSMISSION CONTROL PROTOCOL. https://tools.ietf.org/html/rfc793. (2018).
[7] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014), 87–95.
[8] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, et al. 2017. drmt: Disaggregated programmable switching. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 1–14.
[9] Juntao Fang, Shenggang Wan, and Xubin He. 2018. RAFI: Risk-Aware Failure Identification to Improve the RAS in Erasure-coded Data Centers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 495–506.
[10] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. *The Google file system*. Vol. 37. ACM.
[11] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, Sergey Yekhanin, et al. [n. d.]. Erasure Coding in Windows Azure Storage.. In *Usenix annual technical conference*. Boston, MA, 15–26.
[12] Theo Jepsen, Daniel Alvarez, Nate Foster, Changhoon Kim, Jeongkeun Lee, Masoud Moshref, and Robert Soulé. 2019. Fast String Searching on PISA. In *Proceedings of the 2019 ACM Symposium on SDN Research*. ACM, 21–28.
[13] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. 2018. NetChain: Scale-Free Sub-RTT Coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association.
[14] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 121–136.

[15] Osama Khan, Randal C Burns, James S Plank, William Pierce, and Cheng Huang. 2012. Rethinking erasure codes for cloud file systems: minimizing I/O for recovery and degraded reads.. In *FAST*. 20.

[16] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. 2018. 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*. ACM, 297–312.

[17] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. 2016. One sketch to rule them all: Rethinking network flow monitoring with univmon. In *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 101–114.

[18] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 15–28.

[19] James S Plank. 1997. A tutorial on Reed–Solomon coding for fault-tolerance in RAID-like systems. *Software: Practice and Experience* 27, 9 (1997), 995–1012.

[20] KV Rashmi, Preetum Nakkiran, Jingyan Wang, Nihar B Shah, and Kannan Ramchandran. 2015. Having Your Cake and Eating It Too: Jointly Optimal Erasure Codes for I/O, Storage, and Network-bandwidth.. In *FAST*. 81–94.

[21] KV Rashmi, Nihar B Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. 2015. A hitchhiker's guide to fast and efficient data reconstruction in erasure-coded data centers. *ACM SIGCOMM Computer Communication Review* 44, 4 (2015), 331–342.

[22] KV Rashmi, Nihar B Shah, and P Vijay Kumar. [n. d.]. Optimal Exact-Regenerating Codes for the MSR and MBR Points via a Product-Matrix Construction. *submitted to IEEE Transactions on Information Theory. Available online at arxiv* 1005 ([n. d.]).

[23] Irving S Reed and Gustave Solomon. 1960. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics* 8, 2 (1960), 300–304.

[24] Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros G Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. 2013. Xoring elephants: Novel erasure codes for big data. In *Proceedings of the VLDB Endowment*, Vol. 6. VLDB Endowment, 325–336.

[25] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. 2016. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 15–28.

[26] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S Muthukrishnan, and Jennifer Rexford. 2017. Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research*. ACM, 164–176.

[27] John Sonchack, Oliver Michel, Adam J Aviv, Eric Keller, and Jonathan M Smith. 2018. Scaling Hardware Accelerated Network Monitoring to Concurrent and Dynamic Queries With* Flow. In *2018 USENIX Annual Technical Conference (USENIX ATC18)*. USENIX Association.

[28] Fang Wang, Yingjie Tang, Yanwen Xie, and Xuehai Tang. 2019. XORInc: Optimizing Data Repair and Update for Erasure-Coded Systems with XOR-Based In-Network Computation. In *2019 35th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 244–256.

[29] Dingming Wu, Ang Chen, TS Eugene Ng, Guohui Wang, and Haiyong Wang. 2019. Accelerated Service Chaining on a Single Switch ASIC. (2019).

[30] Mingyuan Xia, Mohit Saxena, Mario Blaum, and David Pease. 2015. A Tale of Two Erasure Codes in HDFS.. In *FAST*. 213–226.