ZipLine: In-Network Compression at Line Speed

Sébastien Vaucher
Université de Neuchâtel
Neuchâtel, Switzerland
sebastien.vaucher@unine.ch

Nilofar Yazdani
Aarhus Universitet
Aarhus, Denmark
n.yazdani@eng.au.dk

Pascal Felber
Université de Neuchâtel
Neuchâtel, Switzerland
pascal.felber@unine.ch

Daniel E. Lucani
Aarhus Universitet
Aarhus, Denmark
daniel.lucani@eng.au.dk

Valerio Schiavoni
Université de Neuchâtel
Neuchâtel, Switzerland
valerio.schiavoni@unine.ch

ABSTRACT

Network appliances continue to offer novel opportunities to offload processing from computing nodes directly into the data plane. One popular concern of network operators and their customers is to move data increasingly faster. A common technique to increase data throughput is to compress it before its transmission. However, this requires compression of the data—a time and energy demanding preprocessing phase—and decompression upon reception—a similarly resource consuming operation. Moreover, if multiple nodes transfer similar data chunks across the network hop (e.g., a given pair of switches), each node effectively wastes resources by executing similar steps. This paper proposes ZipLine, an approach to design and implement (de)compression at line speed leveraging the Tofino hardware platform which is programmable using the P4 language. We report on lessons learned while building the system and show throughput, latency and compression measurements on synthetic and real-world traces, showcasing the benefits and trade-offs of our design.

CCS CONCEPTS
- Networks → In-network processing; Programmable networks; Network measurement; Network experimentation.

KEYWORDS
P4, Tofino, generalized deduplication, compression, programmable switches

ACM Reference Format:
Sébastien Vaucher, Nilofar Yazdani, Pascal Felber, Daniel E. Lucani, and Valerio Schiavoni. 2020. ZipLine: In-Network Compression at Line Speed. In The 16th International Conference on emerging Networking EXperiments and Technologies (CoNEXT ’20), December 1–4, 2020, Barcelona, Spain. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3386367.3431302

1 INTRODUCTION

Programmable switches allow network engineers to have fine-grain control of the packet flow as well as execute custom operations on those packets, effectively leading to in-network processing. This new paradigm opened the path to offload cumbersome and resource-intensive computations directly into the network data plane. It has been shown to lead to significantly reduced resource usage at end nodes and increase in network throughput [23], despite the special care required to handle novel families of threats [6]. In-network computing has been applied to several domains, including packet aggregation [28], databases [18], machine-learning acceleration [27], data analytics [19], network telemetry [5], and even consensus protocols [11]. This paradigm can also result in significant energy savings [34] and even offer novel ways to defend against byzantine behaviors [26].

The P4 language [9] is becoming the de facto standard supported by switches with programmable data planes. One reason for its popularity is its programming flexibility and the ability to easily verify or test P4 programs [14, 44]. Its adoption is quickly rising, including software-defined network (virtualized) settings [16, 29] and in hardware to synthesize low-level descriptions for field-programmable gate array (FPGA) boards [7]. Lately, network appliance vendors have started offering native support of P4 with comprehensive toolchains, hence paving the way to novel and easy-to-maintain network-related applications which are beyond the limits of FPGA implementations.

This paper presents ZipLine, the first in-network data (de)compressor operating at line-speed. ZipLine leverages a novel data deduplication technique: generalized deduplication (GD) [36], detailed in section 2. Due to the paradigm change imposed by the target hardware, we revise the processing workflow of GD (section 4). ZipLine exploits P4 on Tofino native architecture (TNA) to deploy our implementation on the Barefoot Tofino programmable application-specific integrated circuit (ASIC) [4].

The contributions of this paper are as follows. We introduce some theoretical background on GD in section 2. We survey related work in section 3. Then, we describe how to implement GD on Tofino switches, a non-straightforward endeavor (section 4 and section 5). We report on several implementation caveats and the lessons learned while doing so (section 6). Finally, we show that both static and dynamic learning versions of our approach can achieve line rate speed (i.e., 100 Gbit/s on our hardware) for compression and decompression using synthetic and real-world packet traces, thus offloading end nodes from the heavy (de)compression burden (section 7). We conclude and hint at future work in section 8.
2 BACKGROUND

This section provides an overview of our compression algorithm and the coding techniques used for its implementation.

Generalized deduplication (GD). GD is a generalization of the concept of data deduplication [36], where the system first applies a transformation function on a data chunk to split it into a pair of values: a basis and a deviation. Then, the system proceeds to deduplicate that basis against previously received ones. The (small) deviation is kept to be able to invert the process upon reading the data.

For example, a transformation function like the one described in section 2 would map chunks \{0000000, 0000001, 0000010, 0000100, 0001000, 0010000, 0100000, 1000000\} to a single basis 0000 and a deviation of 3 bits, indicating what bit is set (if any). The same code would map chunks \{1111111, 1111110, 1111101, 1111011, 1110111, 1101111, 1011111, 0111111\} to basis 1111. In general, the longer the chunk’s length (in bits), the more chunks are mapped to the same basis. The result is that thousands or even millions of chunks can be mapped to the same basis, which increases the potential for compression.

Note that a dictionary or registry built based on the most commonly used bases instead of the chunks would be able to represent more (or at least the same number of) chunks, i.e., it allows for a more efficient dictionary to be built. If we used the above example, a 42-bit sequence \{0000000011111111010000011101000101101110\} has six different 7-bit data chunks, while it can be represented with only two bases. Two bases in a dictionary could use an ID of only one bit to identify each base. Thus, the data can be represented with a dictionary of eight bits containing the two bases and a sequence of one-bit ID and 3-bit deviations: 0:000 | 1:000 | 2:111 | 3:010 | 4:010 | 5:110 | 6:110 of only 24 bits. The values for the deviations for this example are described in more detail in Table 2a.

Although originally considered for large scale data storage [21, 32], GD has been adapted to multi-source data compression protocols [15] and file compression for time-series data [35, 37]. This has resulted in lightweight, online compression mechanisms suitable to the Internet of Things (IoT) an file compressors with excellent random access properties.

In this paper, we consider Hamming codes as the core component of the transformation function for GD, as in [15, 36]. The main motivation is that they can be implemented using the cyclic redundancy checks (CRCs) built-in functionality of Tofino switches for fast execution, as discussed below. We study cases where the dictionary is preloaded in the switches, or where the sender and receiver dynamically learn it.

Cyclic redundancy check (CRC). Let us consider a block of data \(B\) with \(n\) bits, where \(B\) can be expressed as a polynomial \(B(x) = b_0x^0 + b_1x^1 + ... + b_{n-1}x^{n-1}\) and \(b_i\) is the \(i\)-th bit of \(B\). We consider \(b_{n-1}\) as the most significant bit (MSB) of \(B\). Computing an \(m\)-bit long CRC requires computing the long division of \(B(x)\) with a generator polynomial \(g(x) = g_0x^0 + g_1x^1 + ... + g_mx^m\).\(^1\) The residue of this division is the CRC value. One of the properties of CRCs is that \(\text{CRC}(A \oplus B) = \text{CRC}(A) \oplus \text{CRC}(B)\). This means that pre-computing all \(n\)-bit sequences with a single non-zero bit allows us to compute any sequence of \(n\) bits by XORing the appropriate CRCs sequences. More explicitly, \(\text{CRC}(B) = b_{n-1}\text{CRC}(1000...0) \oplus b_{n-2}\text{CRC}(0100...0) \oplus ... b_1\text{CRC}(00...010) \oplus b_0\text{CRC}(00...001)\), which can be represented in matrix form as \(\text{CRC}(B) = BH^T\).

Hamming codes. Hamming codes are block codes that convert \(k\) bit messages into \(n\) bit messages by adding \(m\) parity bits [8]. More specifically, \(n = 2^m - 1\) bits and \(k = n - m = 2^m - m - 1\) bits. They are generated using a generator matrix \(G\) as follows:

\[
G = \begin{pmatrix}
g_0,0 & g_0,1 & \cdots & g_0,n-1 \\
g_1,0 & g_1,1 & \cdots & g_1,n-1 \\
\vdots & \vdots & \ddots & \vdots \\
g_{k-1},0 & g_{k-1},1 & \cdots & g_{k-1},n-1
\end{pmatrix}
\]

where the size of the matrix is \(k \times n\).

Such a code can be transformed to a systematic form, where the original message is embedded directly into the codeword and the parity bits are clearly separated from it as \(G = [I_k \ P]\), which is achieved by performing row operations to eliminate components. \(I_k\) indicates an identity matrix of size \(k \times k\).

For our work, we will consider a case where the order of parity and identity matrices are shifted, i.e., \(G = [P \ I_k]\), as it matches the output of CRC functions. The parity-check matrix, \(H\) of size \(m \times n\) is given by \(H = [I_{n-k} \ P]\), which is actually the same as for computing the CRC with the same generator polynomial [8]. A message \(u = (1 \times k)\) is encoded into the codeword \(c = (1 \times n)\) by \(c = uG\).

For decoding, the received sequence is \(B = c + e\), where \(e\) is the error pattern. An all-zero \(e\) means there are no errors. The matrix \(H\) can be used to detect whether any errors occurred by calculating the syndrome vector \(s = BH^T = (c + e)H^T = eH^T\), due to the fact that \(G_kH^T = 0\) (by construction of \(H\) and \(G_k\)). A look up table can be constructed that maps the different 1-bit error patterns to a corresponding \(s\) value.

Connection of hamming codes to CRCs. A CRC-\(m\) using a generator polynomial that is suitable to a Hamming code \((n, k)\) can help compute the syndrome and parity bits in the decoding and encoding steps. More specifically, as long as (1) the generator polynomial for a Hamming code is used as parameter for the CRC-\(m\) generator polynomial, and (2) the size of the input data to be computed by the CRC-\(m\) is \(n = 2^m - 1\) bits as expected by a standard Hamming code, then the computed CRC-\(m\) value will be equivalent to the syndrome computed for a \((n, k) = (2^m - 1, 2^m - m - 1)\) Hamming code. Table 1 provides generator polynomials for Hamming codes and the expected input parameter for the CRC-\(m\) module in Tofino chipsets.

3 RELATED WORK

Programmable network switches have sparked a lot of interest in academia and industry. In particular, the P4 language allows implementing novel tools, e.g., real-time network telemetry and analytics [22, 33], and accelerating existing applications, e.g., stream processing [17]. One of the reasons for P4’s current momentum is that it allows easy verification of in-network programs [20, 30]. We will exploit this opportunity in future work.
Table 1: Generator polynomials for Hamming codes and parameters for a CRC-m.

| Code | Generator polynomial | Parameter for CRC-m |
|------|---------------------|---------------------|
| (7, 4) | $x^3 + x + 1$ | 0x3 |
| (15, 11) | $x^3 + x + 1$ | 0x3 |
| (31, 26) | $x^3 + x^2 + 1$ | 0x05 |
| (31, 26) | $x^3 + x^4 + x^2 + x + 1$ | 0x17 |
| (63, 57) | $x^3 + x + 1$ | 0x03 |
| (127, 120) | $x^3 + x^3 + 1$ | 0x09 |
| (255, 247) | $x^3 + x^4 + x^3 + x^2 + 1$ | 0x1D |
| (511, 502) | $x^3 + x + 1$ | 0x06D |
| (511, 502) | $x^3 + x^3 + x^2 + x^3 + x + 1$ | 0x0F3 |
| (1023, 1013) | $x^{10} + x^3 + 1$ | 0x009 |
| (2047, 2036) | $x^{11} + x^2 + 1$ | 0x005 |
| (4095, 4083) | $x^{12} + x^4 + x^3 + x + 1$ | 0x053 |
| (8191, 8178) | $x^{13} + x^3 + x^2 + x + 1$ | 0x01B |
| (16383, 16369) | $x^{14} + x^3 + x^6 + x + 1$ | 0x143 |
| (32767, 32752) | $x^{15} + x + 1$ | 0x003 |

P4 on Tofino has been successfully used to implement NetEC [24], an efficient in-network erasure coding (EC). Data streams arrive from different switch ports, aggregated using Reed-Solomon EC [25], and leave the switch from yet another port in aggregated fault-tolerant form. Both ZipLine and NetEC offload heavy-duty processing from end-node CPUs: packet compression and packet encoding, respectively. While NetEC mainly operates over packet payloads, P4-enabled switches can be used to deploy more generic forms of aggregations and disaggregation, e.g., over several header frames for fixed-size [40] or variable-size [39] payloads, while still achieving line-rate throughput. In a nutshell, small payloads are stored in the switch’s register array and, upon a new small packet arrival, they are recirculated via the ingress parser until the maximum transmission unit (MTU) is reached, i.e., until the aggregation is complete.

Chen et al. [10] presented a non-trivial design and implementation for an in-network symmetric encryption scheme (AES) for different payload sizes, where packets go through the switch pipeline multiple times to complete the 10 rounds used for AES-128. In contrast, ZipLine does not need packet recirculation as GD can be implemented in a single round, which translates in higher speed and lower delay.

Although there exist switches performing on-the-fly compression of data streams [1-3], these operate over packets at layer 3 or above. Our approach can support a wider set of network and transport protocols by operating at layer 2.

Since ZipLine leverages GD, it performs particularly well under very limited memory constraints [43]. For instance, DEFLATE [12] is a well-known standard compression technique, which requires a minimum of 3 kB to compress data.

IoT applications normally deal with small data chunks and compression per chunk due to memory limitations. However, standard compression techniques are known to perform poorly over small data [41]. GD performs particularly well for small data chunks, which makes ZipLine suitable for a wide range of chunk sizes. In many application scenarios, such as live video streaming, freshness of information plays a key role. Standard compression techniques, i.e., DEFLATE, require enough data to provide satisfactory compression, negatively affecting the freshness of information, as collection of such data takes time. ZipLine is perfectly suitable for applications with timeliness restrictions, as GD provides significant improvements in terms of freshness of information compared to state-of-the-art techniques [42].

4 APPROACH

Our approach uses the GD algorithm presented in section 2 to implement compression of network packets in a way that is implementable on a readily-available programmable switch. The encoding workflow is shown in Figure 1. Let us consider a data payload $B$ of size $n$, part of an incoming network packet. The first action to perform is to compute the syndrome $s$ using a Hamming decoder (mapped to an equivalent CRC) $\mathcal{O}$. The syndrome tells us which bit in the original data needs to be flipped. We achieve this by having a table $\mathcal{O}$ that matches to the correct mask $f$ to be XORed to the original data $B$ $\mathcal{O}$ according to $s$. The result of this XOR operation $b'$ is truncated to the rightmost $k$ bits to form a basis (m_0, ..., m_{k-1}) $\mathcal{O}$. As several data chunks share the same basis, we take this opportunity to replace them with shorter identifiers $I$ should the basis been seen before $\mathcal{O}$. In order for the recipient to be able to find $B$ again, we attach the syndrome $s$ to the compressed packet $\mathcal{O}$.

The decoding workflow is shown in Figure 2. We consider that a packet containing a compressed basis has arrived $\mathcal{O}$. For an uncompressed basis, the workflow instead starts at step $\mathcal{O}$. On the recipient’s side, there must exist a table that maps short identifiers $I$ to basis that is synchronized with its counterpart in the encoder $\mathcal{O}$. We zero-pad the recovered basis $\mathcal{O}$ and then feed it to the same CRC generator as on the encoder $\mathcal{O}$ to get $p$, hence restoring the bits that we truncated in the encoder. In parallel, we use the same syndrome look-up table as in the encoder to identify which bit needs to be flipped according to $s$. This gives us the appropriate mask $f$ to be XORed $\mathcal{O}$. We apply the mask to flip the right bit out of the concatenation of $p$ and the basis $\mathcal{O}$ to successfully restore the original data $B$ $\mathcal{O}$. 

![Figure 1: GD encoding workflow on Tofino.](image-url)
5 IMPLEMENTATION

Our implementation uses the special capabilities of the Tofino chip from Barefoot Networks. We use the P4 language, with Tofino-specific proprietary extensions, i.e., TNA, and Barefoot software development environment (SDE) version 9.2.0. The control plane is implemented using Python and Barefoot runtime (BfRt). We favored simplicity whenever possible. In this mindset, we chose to develop everything from scratch, which also reduces the likelihood of resource allocation issues. We settled on Ethernet-based framing to provide compatibility with regular Ethernet network cards and guarantee inter-connectivity.

We define three different types of packets: (1) regular, yet uncompressed packets; (2) processed, but uncompressed packets, comprised of a basis and a syndrome, and (3) processed and compressed packets, which replace the basis with a shorter identifier. Packet type 1 can be any Ethernet packet that arrives in the switch. Our implementation takes such packets as input, and then transforms them into types 2 or 3. The latter type can only be produced when there exists a basis-ID mapping for the basis that the original type 1 packet maps to. Such mappings are initialized once an unknown basis is computed. They remain valid for as long as possible. A least recently used (LRU) cache eviction policy decides when to recycle an identifier to map to a different basis. More details about this stateful part of our implementation are provided in later paragraphs.

The foundation of our implementation is the GD algorithm as described in section 2. Notably, GD uses Hamming codes to work as they are equivalent to some particular CRCs. Table 2 shows an example of the equivalence between the (7, 4) Hamming code and CRC-3 with a carefully chosen generator polynomial. The Tofino platform offers a native component to compute CRCs, as such codes are commonly found in many network protocols. We extensively rely on this component to efficiently implement the key steps of the GD algorithm on Tofino, namely the computation of syndromes.

The next part of the workflow requires flipping one bit in the input data according to the syndrome that we obtained in the previous step. We use a P4 table with constant entries that are pre-computed using a short C++ program making use of Boost CRC library [38]. The entry that matches the syndrome is XORed to the data, hence flipping the appropriate bit of the sequence. This transformation creates the basis.

In order to allow for the compression of packets, we need to replace syndrome + basis couples with shorter identifiers. One possible way to select an identifier for a given basis would be to use cryptographic hashes, which cannot be computed in one pass on our programmable switch. The alternative is to use arbitrary identifiers (IDs). We involve the control plane to manage the pool of identifiers. Unknown bases are sent up by means of digests, as provided by P416/TNA. Recording a new basis-ID mapping is done in two phases; first, the control plane chooses an identifier to assign to the basis. When there are unused identifiers, the control plane selects the least recently used one. Should all identifiers be in use, an LRU policy is applied to evict and recycle an identifier to accommodate the most recent basis. Setting a time to live (TTL) that is automatically decreased as time elapses on a particular table entry is a feature that is provided by TNA. With the identifier now selected, the control plane first sets the reverse mapping (ID-basis) in the destination switch to make sure that compressed packets can always be uncompressed. The control plane can finally add a corresponding entry in the source switch, to map the newly-discovered basis to the chosen identifier. Subsequent packets sharing the same basis are then transmitted in a shorter syndrome + identifier form.

Last, we add counters to our program to provide easily-accessible statistics of the inner-workings of ZipLine. In particular, packets are classified according to how they are transformed (e.g., raw packet to syndrome + basis, syndrome + identifier to raw packet, etc.).

### Table 2: Hamming code (7, 4) and CRC-3 equivalence

| Error | Bit sequence | Syndrome | Poly. | Bit sequence | CRC-3 |
|-------|--------------|----------|-------|--------------|-------|
| 0     | (00000001)   | (001)    | $x^0$ | (0000001)    | (001) |
| 1     | (00000100)   | (010)    | $x^1$ | (0000010)    | (100) |
| 2     | (00010000)   | (010)    | $x^2$ | (0000100)    | (100) |
| 3     | (00100000)   | (011)    | $x^3$ | (0010000)    | (011) |
| 4     | (01000000)   | (110)    | $x^4$ | (0010000)    | (110) |
| 5     | (01000000)   | (111)    | $x^5$ | (0100000)    | (111) |
| 6     | (10000000)   | (101)    | $x^6$ | (1000000)    | (101) |

### Figure 2: GD decoding workflow on Tofino.

![GD decoding workflow on Tofino](image)

6 LESSONS LEARNED

We report on several lessons that we learned during the development process of ZipLine. In particular, we highlight some potential intricacies that we encountered when developing ZipLine with P416/TNA for the Barefoot Tofino platform. While P416 is a flexible language, it is mainly designed for efficient processing of network packet headers and routing. Similarly, the hardware design of the Tofino chip is tailored to the same goals. Our quite unorthodox use of the combination of P416 and the underlying hardware is far from traditional packet routing. As a consequence, our implementation had to be adapted to circumvent some resulting limitations.

Header declarations in P416 must be aligned on byte boundaries. However, the Hamming codes that we use are never aligned on bytes. While P416 itself treats data at bit-level granularity, the hardware—and the compiler by extension—are optimized for byte-aligned data. A side effect is that we have to introduce padding bits in our P416 program when selecting particular sizes that are not
byte-aligned for the program to compile. In turn, this introduces a loss of goodput when these sizes are selected, limiting the range of interesting parameter values to only a few.

In our original design, we placed as much of the code as possible in the data plane. This was made possible by leveraging *registers*, *i.e.*, user-accessible memory in the Tofino data plane pipeline. Doing so allowed us to achieve line-rate performance and virtually instantaneous learning of new basis-ID pairs. However, as every piece of code running in the data plane must be able to execute in constant time, many algorithms are out-of-reach, *e.g.*, algorithms that need a complete view over an array of registers. Therefore, we settled on storing basis-ID pairs in regular match-action tables and manage them with the control plane, which is implemented in a regular programming language (*i.e.*, Python in our case). This allows us to use features such as *digests* and *per-table-entry TTLs* that the TNA framework provides, conveniently letting us implement an LRU cache for basis-ID pairs. We can also send updates regarding ID-basis pairs to other *ZipLine* instances out-of-band. The drawback of this approach is that updates take a longer time to effectively apply (see section 7).

Finally, we recognize that the Barefoot Tofino platform is flexible in many aspects. One such aspect is the unified ingress-egress pipeline. It is possible to artificially extend the pipeline without recirculating packets by shifting parts of the processing (*i.e.*, GD decoding) to the egress. This provides space for more table entries, as the compiler can place egress tables across stages that the ingress pipeline underuses.

## 7 EVALUATION

Our setup leverages an Edgecore Wedge100BF-32X programmable switch [13]. It is connected at 100 Gbit/s to two Dell PowerEdge R7515 servers through Mellanox ConnectX-5 network cards interfaced with PCI Express 3.0 x16. Each server has an AMD EPYC 7302P processor, 32 GiB of RAM and runs Ubuntu 20.04.1. We tune each server with the `mlnx_tune` utility in `HIGH_THROUGHPUT` mode. Unless stated otherwise, each measurement is repeated 10 times, and we show the average and the 95 % confidence interval.

**Choice of parameters.** It is possible to set several parameters in *ZipLine*. Three of them pertain to Hammering codes: *k*, *n* and *m* (*as introduced in section 2*). Specifically, the values of *k* and *n* are strongly linked to the value of *m* by formulas, so only *m* can in fact be freely selected. Since the Tofino platform has explicit byte-alignment constraints on header fields, every value of *m* that is not a multiple of 8 requires us to include useless padding bits in packets. We select *m* = 8 which is the largest *m* that is a multiple of 8 and that fits hardware constraints.

Additionally, it is possible to independently choose the size of the short identifiers that replace the bases. This parameter dictates how many bases have to be cached by *ZipLine*. Again, byte-alignment constraints have to be adhered to when choosing this value, such that useless padding bits can be omitted. We require one additional bit to store the MSB of the raw data packet—its size, *k*, is always one below a multiple of 8. Therefore, this parameter also needs to be one below a multiple of 8 to omit useless padding bits. Akin to our choice of *m* = 8, we settle for the largest value that is one below a multiple of 8 and fits hardware constraints (especially in terms of resource usage in this case): 15 bit, allowing for $2^{15} = 32,768$ cached bases.

**Dynamic learning.** We want to measure the consequence of our decision to move the code responsible for managing bases-ID pairs to the control plane (see section 6). In the data-plane, every stateful operation appears to be instantaneous and already applies to the next packet in the pipeline. The main drawback of involving the control plane relates to performance, as it is located away from the path that packets take.

In this experiment, we measure the time between the arrival of an unknown basis in the switch and the moment after which the basis is registered in the compression table, and compressed packets start to be produced. To do so, we repeatedly send the same data packet as fast of possible from one server to another. We capture packets on the destination server and measure the amount of time it takes between the arrival of the first packet of type 2 and the arrival of the first packet of type 3 (see section 5 for the definitions of types). Our results indicate that it takes $(1.77 \pm 0.08)$ ms for *ZipLine* to record and apply a new basis-ID pair. During that time, packets that share the same basis stay uncompressed. This loss in compression efficiency is measured next.

**Compression.** The goal of this experiment is to assess the compression ratio that can be obtained by using *ZipLine*. We use two datasets, a synthetic and a real-world one. We engineered the synthetic dataset to be behaviorally close to typical readouts from a sensor. We generate 3,124,000 chunks of 256 bit (matching the parameters we chose), which are then converted to a `pcap` trace of Ethernet packets containing the chunks as payload.

The real-world dataset consists in a day of DNS queries at a 4000 users university campus [31]. To obtain the trace that we replay, we apply filters to the downloaded files to only keep queries of 34 B going to the main DNS resolver of the campus, excluding the DNS transaction identifier which is a random number.

We replay these traces to our switch and monitor which action *ZipLine* undertakes with the payload of each packet. We then deduce the payload size, as each action produces a packet type of a fixed size. The sum of all original chunks represents the baseline. In Figure 3, the bars represent the total size of the payloads of all

---

**Figure 3:** Resulting payload size after traffic is processed with Gzip and *ZipLine*, without, with static-, and with dynamically learned-compression table mappings.
packets after they are transformed by ZipLine. We also show the numerical ratio to the baseline next to each bar.

We measure three cases: (1) no table: the compression table stays empty; (2) static table: we pre-compute the basis of each payload and add a corresponding mapping in the compression table before we start the experiment; (3) dynamic learning: we start the experiment with an empty compression table, which is then automatically filled by ZipLine when unknown bases are encountered, and (4) Gzip: we extract all payloads in a regular file that we compress with the gzip compression tool. In theory, case 1 should be equal to the baseline as applying GD does not introduce additional bits. The 3% overhead is due to padding bits which are necessary to guarantee container alignment on the Tofino platform. We reckon that 8 such padding bits could be eliminated by an expert P4/TNA programmer. Case 2 represents the idealistic scenario where the basis of every packet payload is known beforehand. Case 3 represents a real scenario where the traffic is unknown to the switch prior to its arrival.

We see that ZipLine correctly learns and stores bases in its compression table, providing savings of 89% in terms of bytes transmitted in the synthetic scenario, and up to 90% in the DNS dataset. The compression ratio of ZipLine compares well with an off-the-shelf compression utility (circa 20% difference), gzip, which uses an algorithm (DEFLATE) that doubtlessly cannot be implemented on our hardware P4 target due to its unbounded execution time.

The delta between cases (2) and (3) in the synthetic scenario is first due to the fact that one packet with a payload mapping to each basis must be transmitted without compression to let the recipient know about it. Second, a couple milliseconds are needed between the time a packet with an unknown basis arrives in the switch, and the mapping to its basis becomes effective for subsequent packets (as shown above), meaning that additional packets sharing the same basis stay uncompressed during this processing delay.

Raw performance. Finally, we measure the raw performance of ZipLine using raw_ethernet_ utilities provided by Mellanox. We start by measuring the raw Ethernet throughput between 2 machines through the programmable switch. We transfer Ethernet frames of 3 common sizes for 10 seconds: the minimum frame size of 64 B, the standard 1500 B, as well as jumbo frames of 9 kB. The first scenario ("no op") acts as the baseline, with the switch acting as a regular Ethernet switch. We then repeat the same measurements with the switch performing either the encoding or the decoding phase of ZipLine. Figure 4 shows our results. We notice that the claims put forwards by the vendor of our programmable switch are kept; namely that any P4 program that successfully compiles for the Tofino platform performs at line speed, so long as it does not make use of recirculation, packet duplication, etc. The figures for 64 and 1500 B packets are bottlenecked at around 7 Mpkt/s by the server generating the traffic. In theory, the full line rate of 100 Gbit/s can also be reached with these smaller packets, as per the 4.7 Gpkt/s figure quoted in the datasheet of the switch [13].

Subsequently, we evaluate the latency by having one server sending packets to itself via the programmable switch. We then measure the round-trip time (RTT). Results are shown in Figure 5 and point towards the same findings than our throughput measurements: the addition of ZipLine to the P4 program has no noticeable effect on raw performance.

8 CONCLUSION AND FUTURE WORK
This paper introduces ZipLine, an approach for in-network data compression that can operate at line rate with minimal delay on (de)compression. We adapt the concept of generalized deduplication (GD) to provide an efficient implementation and a mapping of the data that can significantly boost the benefits of limited dictionaries implemented in the switches. We rely on transformations based on Hamming codes, which can be implemented efficiently with the switch’s cyclic redundancy check (CRC) components. Our implementation shows that (de)compression at line rate (100 Gbit/s in our switch) is possible and that both static tables (for the dictionary) and dynamic learning can be implemented without compromising throughput or compression gains.

The use of the CRC module in Tofino switches opens the door to computation of more complex transformations, e.g., BCH codes, by using different generator polynomial parameters [8]. These allow for more chunks to be mapped to each basis, albeit at the cost of a larger deviation in bits. Future work will explore alternative transformations with efficient implementations, e.g., bit-swapping [37], and dynamic approaches to select a wider range of parameters.

ACKNOWLEDGMENTS
We are thankful to Vladimir Gurevich (Barefoot Networks) for his insightful feedback. This work was partially financed by the SCALE-IoT Project (Grant No. 7026-00042B) granted by the Independent Research Fund Denmark, by the Aarhus Universitets Forskningsfond (AUFF) Starting Grant Project AUFF-2017-FLS-7-1, and Aarhus University’s DIGIT Centre.
