Comprex: In-Network Compression for Accelerating IoT Analytics at Scale

Rafael Oliveira, *Georgia Institute of Technology*
Ada Gavrilovska, *Georgia Institute of Technology*

Abstract—To enable the Internet of Things (IoT) to scale at the level of next-generation smart cities and grids, there is a need for a cost-effective infrastructure for hosting IoT analytics applications. Offload and acceleration via smartNICs have been shown to provide benefits to these workloads. However, even with offload, long-term analysis on IoT data still needs to operate on a massive number of device updates, often in the form of small messages. Despite offloading, the ingestion of these updates continues to present server bottlenecks. In this paper, we present domain-specific compression and batching engines that leverage the unique properties of IoT messages to reduce the load on analytics servers and improve their scalability. Using a prototype system based on InnovaFlex programmable smartNICs and several representative IoT benchmarks, we demonstrate that these techniques achieve up to 7× improvement over existing offload approaches.

The number of IoT devices and applications is growing exponentially, spanning diverse industry verticals, from different wearables and smart city applications, distributed agriculture and smart infrastructure, to ultra-low-latency industrial automation and mission-critical control. In response of this trend, commercial cloud (e.g., Amazon AWS, Google, Microsoft) and network (e.g., Deutsche Telecom, SK Telecom, China Mobile, AT&T) operators, are increasingly offering IoT-based hosting services, ranging from programming APIs to fully-operated infrastructure for hosting third-party IoT applications. The scalability and efficiency of this infrastructure tier, is, thus, relevant to a large number of stakeholders. This is expected to be further exacerbated by proliferation of 5G technologies, where ultra-reliable and ultra-low latency capabilities of commercial and private 5G networks are enabling new types of IoT ecosystems with even more stringent end-to-end performance requirements [9].

To understand the requirements for this server infrastructure, we first look at the unique characteristics of IoT. These applications are commonly characterized by massive arrays of heterogeneous sensors, generating periodic updates in the form of small messages. IoT devices are limited in their computational, energy, and communication resources, so applications rely on remote servers, in the cloud or in future edge datacenters [9], to aggregate and process sensor updates. The applications' end-to-end performance requirements include high sustained throughput for the aggregate sensor updates, ability to tolerate unexpected bursts, and low and predictable processing latency for actuation or notification events in response to online analyses of sensor data.

Offload and acceleration to emerging programmable smartNICs are techniques which have been proven effective for delivering low and predictable latency and high throughput for request processing across many domains, including for IoT [3], [5]. However, while these prior solutions make it possible to configure the NIC's compute resources for in-network packet processing of IoT messages, they do not consider that IoT applications include long term data aggregation, analysis and modelling. The compute and storage requirements of these operations, coupled with the complexity of their legacy software stacks, implies that this functionality remains executed on general-purpose host CPUs. As a result, even with smartNIC offload and acceleration, the scalability of the IoT analytics server systems is bottlenecked by the host's packet processing capabilities, due to known limitations of general-purpose CPUs and network stacks to deal with large number of small messages.

By analyzing several real-world IoT applications, we make the following observation. First, IoT applications commonly consist of distinct *critical-path* and *long-term analytics* components. By decomposing applications into their two components, it is possible to extract

XXXX-XXX © IEEE
Digital Object Identifier 10.1109/XXX.0000.0000000

efficiency through *in-network offload and acceleration of the critical-path operations*. Performing the acceleration in-network helps achieve low and predictable latency. Second, IoT applications and their messages are built around the *topic* construct, which provides semantic information regarding the message types and formats to the underlying processing runtime. This further allows for *application-specific batching and compression* to be deployed, but *outside of the critical path*, thereby not impacting critical-path latencies, while scaling the sustainable throughput of the analytics path.

Motivated by these observations, we design **Comprex** – a set of domain-specific batching and compression engines specialized for smartNIC offload and acceleration of IoT. This new topic-aware batching and compression can be integrated with existing smartNIC offload solutions to address the scalability limitations of the smartNIC-host application interface, and to provide significant benefits to IoT analytics. The outcome is a server framework that achieves low and predictable latency for accelerated critical-path IoT operations, with increased throughput scalability of the long-term modeling and aggregation tasks executed via general-purpose CPUs.

The engines are integrated with a smartNIC prototype system based on the Mellanox InnovaFlex smartNICs, equipped with a Xilinx XCKU060 FPGA. Using four representative IoT benchmarks [2], we show that the use of application-specific batching and compression allow Comprex to improve the stable throughput (messages processed per second) by 11× compared to just offload approaches, while not degrading latency for critical-path operations. For a fixed load, it reduces the host's per-core CPU utilization by up to 8×, freeing up those resources to scale the analytics components or to host more applications. The benefits persist even when co-running all four IoT benchmarks to emulate multi-tenant setting.

IoT Acceleration Opportunities and Challenges

IoT Workload Characteristics

While the term IoT refers to a wide range of applications, most IoT applications are characterized by ingesting large volumes of sensor data, analyzing them, and delivering updates to one or more subscribers. Sensor data processing involves a common set of steps, as shown in Figure 1. The boxes in red form a critical path composed of compute kernels that often comes with stringent real-time latency constraints varying from 250 μs to 100 ms, followed by a long-term analytics path (Analytics and Database), composed of compute kernels that operate at much longer time scales. Typical sensors generate small updates of 10s to 1000s of bytes, with existing IoT protocols designed for 20-1600 byte message sizes.

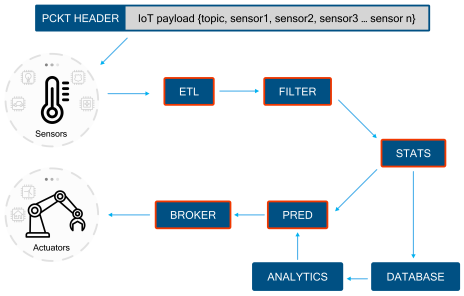

FIGURE 1. A Common IoT application dataflow.

TABLE 1. IoT application workload and evaluated kernels. MLR is Multivariable Linear Regression.

App Name	Number of Fields	Size in bytes	Topic Size	Compute Kernels in Critical Path			
CITY	9	123	35	Raw, ASCII	Range Filter, Bloom Filter	Kalman, Average, Max, Min, Count	Decision Tree
TAXI	27	144	47				MLR
GRID	3	80	30		Range Filter, String Filter		Threshold
FIT	17	153	32				MLR

Motivation for in-network acceleration. Handling these message rates and processing latency requirements with general-purpose server systems is not sustainable, particularly given the small message sizes common in IoT. Realizing a scalable and performant infrastructure for IoT will therefore have to seek efficiency through in-network acceleration.

IoT Modules and Kernels

IoT applications are very modular. They are frequently expressed in a static pipeline of high level compute kernels that goes from parsing data coming from sensors, in a byte format, to server-side friendly format like JSON, to training machine learning models and storing data in persistent memory for future analysis.

These compute kernels are commonly connected in a dataflow format using popular cloud services such as Amazon Kinesis and Lambda functions or Google Cloud functions. The operations shown in the red boxes in Figure 1 form a closed-loop flow capable of detecting and responding to events, near real-time, without any human intervention. In addition, the operations represented in these classes of functionality have already been shown amenable to offload and acceleration [3], [5]. The last two functions operate across aggregates of IoT updates over different spatial and temporal scales, and rely on conventional compute and storage infrastructure on general-purpose servers.

Topic, Publisher and Subscriber

Content of an IoT message. An IoT device serves as a publisher of messages with sensor values, or as a subscriber to messages with actuation commands. A message has the format shown in Figure 1, and consists of a network header and IoT-specific payload. The IoT payload is structured based on one of the common IoT protocols, such as MQTT and CoAP, and it includes a number of fields such as device identifiers (device name, user ID), sensor types (e.g., temperature, luminosity, etc., depending on the sensors available on the device) and sensor data (represented in a well-defined format).

The topic structure. Topic is a key abstraction that connects publishers (IoT devices generating data) and subscribers (control processes and alert systems consuming IoT updates), and is used by the message broker component responsible for distributing data. It is commonly implemented as a string that expresses the hierarchical relationship between the application components.

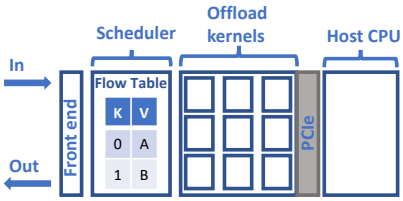

FIGURE 2. Key components of a generic representation of in-network accelerators.

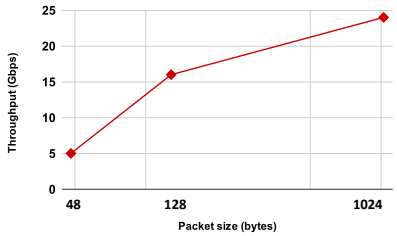

FIGURE 3. Throughput vs. message size on a no-touch flow.

Existing smartNIC Acceleration for IoT

In-network acceleration approaches. Existing in-network accelerators can be grouped into two different categories: Pipeline-of-Offloads NICs (NICA) [3], and re-configurable match-action (RMT) NICs (PANIC) [5]. Figure 2 illustrates the main components present in an in-network accelerator. As packets arrive in the accelerator, they are mapped to an entry in the flow table and scheduled across different offload kernels. The flow table is used to store data regarding how to process the incoming packet. The difference between in-network accelerators can be simplified into how these main components operate in relation to one another. NICA groups offload kernels into a static pipeline that flows can be mapped to. PANIC's offload kernels are connected via crossbar with bi-directional communication, allowing pipelines of kernels to be dynamically created.

Limitation. The smartNIC compute elements can be used to execute the offloaded kernels, at line rate, for each IoT message, process the critical path, and, with low latency, send anomaly notifications to subscribers. However, IoT messages must also be ingested by the host-side analytics path for long(er) term modeling and optimizations. Figure 3 exposes the problem of processing high message rates of small-sized packets. In this microbenchmark, all 5 CPU cores were at 100% utilization. We notice that, with the same number of cores, the throughput can be increased by almost 5x over the minimum size packet. Above 1024 bytes, the number of ingested messages per second, not the message size, becomes the bottleneck. Current smartNIC acceleration solutions provide scalability to host-side analytics by freeing up host resources via offload, but do not explicitly consider the scaling limitations of the large number of IoT messages that still need to be delivered and processed in the analytics data path.

Accelerating the IoT Analytics Path

Given the richness of the IoT space, the operations in the analytics path vary significantly, ranging from database load/store and queries, to machine learning training. As such, they rely on the host-side runtime CPU for their execution. We use two insights to scale up the analytics path on the host-side runtime CPU. First, at high message loads, the host-side runtime CPU becomes a bottleneck for even the basic processing needed to extract individual IoT messages from network packets, underutilizing both the available network bandwidth and PCIe bandwidth. Second, the analytics path is throughput-oriented and does not have strict requirements on latency, unlike the critical path. Leveraging the two insights, we propose Comprex, a NIC-side engine that compresses and batches together a large number of IoT messages to fill out pre-allocated (MTU-sized) buffers that are then passed to the host-side CPU runtime.

Compression and Batching of IoT messages. Fig-

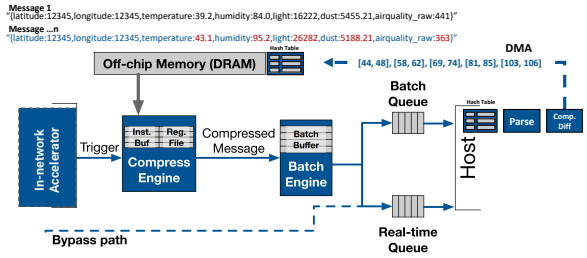

FIGURE 4. Compress Engine and Batch Engine used for accelerating the transfer of IoT messages from the network card to the host CPU.

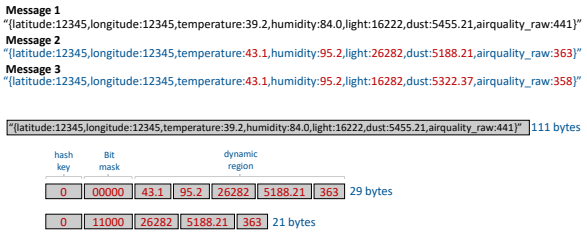

FIGURE 5. Compress Engine region elimination.

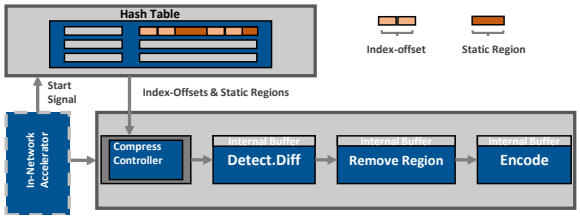

FIGURE 6. Compress Engine components.

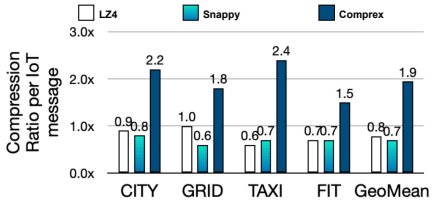

FIGURE 7. Compression gain comparison between Comprex IoT specific compression and generic compression techniques.

ure 4 illustrates how data flows from the in-network accelerator to the Compress and Batch engines. The Compress Engine receives its trigger to start execution when an IoT message is processed by the in-network accelerator that handles the *critical path*. After compression, the Batch Engine combines multiple compressed IoT messages to construct a batch of messages and push them to the batch queue.

There are three cases where the message is pushed to the Real-Time Queue: ① the message bypassed the accelerator using the Bypass path, ② the in-network accelerator did not fully handle the real-time requirement of the message or ③ as explained in the next section, the Compress Engine detected an error and the message was not compressed. The Bypass path is used for messages that are not accelerated.

Comprex Compression

IoT messages are too small to benefit from generic compression techniques like Snappy and LZ4 (see Table 1). In fact, as shown in Figure 7, both LZ4 and Snappy *increase* the overall data volume. However, we observe that a large portion of IoT messages remain unchanged across the two endpoints – the sensors publishing to a certain *topic* and the application server. For instance, a geolocation of a fixed sensor in CITY, or of a taxi waiting for a client in TAXI, user and device ids in FIT and GRID etc., are common examples of parameters that can remain unchanged across messages. However, the device telemetry (temperature, CO2 levels, energy consumption) present in the message body changes more frequently. We refer to the regions of the IoT message that remain unchanged and the regions that change more frequently as static and dynamic regions.

Topic and static regions. A device source IP is rarely static and does not provide information on message structure patterns between devices. The *topic* in the pub-sub model, however, not only remains the same between messages, it indicates that messages published in the same *topic* have the same format [2]. This allows Comprex to detect static regions more reliably between multiple end-points while greatly reducing the memory footprint, since the resources grow with the number of *topics* and not with devices.

Software-hardware co-processing. Detecting the static and dynamic regions of a sequence of messages is not trivial, and it requires that every character in a pair of messages be compared. Unlike the previous version of Comprex [8], where the Compress Engine is used only to remove static regions from messages, this new version of Comprex's Compress Engine is capable of performing string-compare. As such, the host performs the initial analyzes of the static and dynamic regions before offloading the process to the Compress Engine. Pre-computing these regions beforehand yields great hardware optimizations since strings can be parsed in parallel using techniques such as loop-unroll without the additional cost added by a naive implementation of

divide-and-conquer.

When the host server receives a message, it checks to see if there is an entry in the Hash Table that corresponds to the message's *topic*. If the entry does not exist, a function that computes the dynamic region (**Comp.Diff**) marks the entire message as dynamic and adds the new *topic* to the Hash Table. As more messages for the same *topic* arrive, **Comp.Diff** begins to identify the regions of the message that have remained unchanged (static regions) between consecutive messages.

The host-side runtime then loads the Hash Table on the FPGA NIC with the static regions, as well as the index and offsets of the dynamic regions. The Compress Engine uses this information to remove static regions from upcoming messages in the same *topic*, sending only dynamic regions to the host. Furthermore, the Compress Engine stores a number of messages on the same Hash Table entry. If **Comp.Diff** detects any variation in the static region, it instructs the Compress Engine to dump its stored messages into the upcoming packet, which goes into the Real-Time Queue, and the process starts over.

Listing 1. Host side Python source-code sample for processing Comprex messages

```
1  def process(mesg):
2      topic = HT.get_topic(msg.topic)
3      if topic is None:
4          Comp.Diff(msg)
5      else:
6          if topic.N > topic.MAX_N:
7              if Is_Diff(topic, msg):
8                  Comp.Diff(topic, msg)
9                  topic.MAX_N = -1
10             else:
11                 increment(topic.MAX_N)
12         topic.N+=1
13     Compose(topic, msg)
14     Upload(msg)
```

Processing a Compressed Message. Listing 1 is a simplified source code that highlights the steps involved in processing a Comprex message. In lines 1-3, a new *topic* entry is created in case it does not exist. Lines 6-12 illustrate how Comprex relies on the message structure for performance. A *topic*'s **N** and **Max_N**, in line 6, are used to determine how many messages can be processed before invoking **Comp.Diff**, which prevents it from becoming a bottleneck. When **N** becomes greater than **Max_N**, the Compress Engine sends the full message to the host side to be processed by **Comp.Diff**. In line 7, if the static regions have changed, the regions need to be recomputed (line 8) and the confidence (**Max_N**) that the static regions will not change becomes negative (line 9). However, if the static regions remained unchanged, the confidence increases (line 11) until it reaches a user-imposed limit. Lines 13-14 the received message is reconstructed and uploaded to a queue shared with the IoT application that owns the message.

Compress Engine

Once Comprex builds confidence in the static regions of a message, the static regions, the index-offsets and the **Max_N** are added to the in-network accelerator Hash Table and used by the Compress Engine thereafter. Figure 6 illustrates the Compress Engine component pipeline. When the message arrives at the in-network accelerator, it sends a copy of the message to the Compress Engine and signals the Hash Table unit, which triggers the transfer of the index-offsets and static regions for the message being processed.

The Compress Controller uses the index-offsets to extract regions of the original message, pairs them with their corresponding static regions, and transfers them to the **Detect.Diff**. Since the sizes of the regions are known beforehand and the goal of this process is to know whether or not a static region has changed, the **Detect.Diff** stage is capable of processing pairs of 64 bytes per cycle. This is achieved by setting all 64 bytes to 0 and modifying only the $64 - size$ region. For example, if a 50 byte static region remains unchanged, comparing the remaining 14 bytes will not alter the result.

If no change is detected in the static regions, the Remove Region stage removes them from the message and sends the dynamic parts to the Compose stage, which encodes the dynamic regions using the Comprex format. Comprex was designed to work with-and-without the **Detect.Diff** stage. As such, we refer to Comprex equipped with the **Detect.Diff** stage as ComprexDD.

Figure 5 illustrates the process of eliminating duplicated regions from the message payload. All the messages in this example come from the same device, bound to the same *topic*. When **Message 1** arrives in the Compress Engine, no entry for the *topic* exists and the whole message is transferred to the host. When **Message 2** arrives, **Comp.Diff** maps the static regions and loads the accelerator's Hash Table. Even dynamic regions can become static for a short period of time. For example, a stationary vehicle that reports its speed will report the same speed and geolocation in multiple messages. This is illustrated in messages 2 and 3. Eliminating duplicates in dynamic regions is accomplished by adding a 16 bit mask to the message payload that allows the runtime to track the dynamic regions that were eliminated in the Remove Region stage.

Batching

After compression, the Batch Engine combines multiple compressed IoT messages to construct a single large UDP message. The implementation of the Batch Engine is straightforward. It keeps track of the size of the current load, and checks to see if the next message coming from the Compress Engine fits. If it fits, the compressed message is added to the load, and the load size is updated. Otherwise, the current load is transferred to the host-side runtime and a new load is created.

Comprex re-uses the functionality for the network interface packet buffer management on our in-network accelerator SmartNIC, and thus uses maximum size 1500 byte UDP-over-Ethernet network packet buffers for the SmartNIC-host-side runtime communication. Once the batch is received by the host-side runtime, it processes the protocol stack once for the entire batch, decompresses the message by adding the deleted regions, and passes the individual messages to the application. No modifications are required in the application to ingest and process the messages.

One could argue that Snappy and LZ4 will demonstrate better results if used after batching. However, this would impact the number of messages in the batch and require costly variable-length string manipulation operations [1]. Leveraging the well-formed structure of IoT messages allows Comprex to simplify the compression and decompression process by copying data from fixed offsets. It also opens up opportunities for future optimizations such as pre-initializing message buffers.

Software Managed Queues. The current implementation for our prototype uses two different queues to send processed UDP packets to the host-side runtime CPU: (1) a high-priority queue that holds IoT topics that are either not accelerated or not successfully compressed on the SmartNIC; and (2) a low-priority queue that holds the combined large compressed and batched messages. The priorities of the two queues are managed by the Comprex runtime system to ensure fairness and to avoid starvation for different types of queues.

Evaluation

Methodology

To evaluate the impact of domain-specific compression and batching on extending the benefits of SmartNIC offload alone, we use four representative IoT applications, summarized in Table 1, and described in [8]

We use two servers equipped with Intel Xeon X3430 processors running at 2.4 GHz and 16 GB of RAM to emulate IoT traffic at scale. The servers are connected using two 40 Gbps Mellanox Innova Flex SmartNICs, each comprising a Mellanox ConnectX-4 Lx EN ASIC NIC and a Xilinx Kintex UltraScale (XCKU060) FPGA. The first server emulates IoT publishers and subscribers, while the second server hosts the IoT applications.

All results for Baseline and Baseline plus Batch use an in-house C-based server application that operates on top of a light-weight UDP-over-Ethernet protocol. Baseline accepts a single IoT publisher message at a time from the NIC, processes the message to generate and send responses to all subscribers. For Baseline plus Batch, we use the SmartNIC to combine multiple IoT messages from different topics into a single 1500 Byte batch. For Offload, we use the SmartNIC to completely subsume the operations for the critical path in each topic of the evaluated applications. Offload plus Batch further performs batching. Finally, Comprex performs compression and batching to fit a larger number of IoT messages in a single batch. We measure the latency of the critical path as the time from publishing an IoT message to the time when the last IoT notification is received by subscribers.

The reported stable throughput corresponds to a level where message loss does not exceed a current threshold of 0.1% for a duration of 10 minutes. More specifically, for each measurement in this section, we increase the throughput until the host-side becomes unstable. As such, throughput measurements are calculated for messages that have the *critical* and *non-critical sides* properly handled by the system.

Experimental Results

Increase in sustained throughput. Figure 8 compares the throughput in messages-per-second, across different configurations. Baseline plus Batch shows a geomean 1.3× increase in throughput from batching multiple IoT messages into a single network packet, at the cost of ≈ 2× increase in latency, as we show in Figure 9. Further, hardware acceleration alone does not provide a significant improvement in throughput. Offload provides a 2.1× improvement in throughput over Baseline and a 1.6× improvement in throughput over Baseline plus Batch.

By combining the benefits from hardware acceleration and offload with optimizations for domain-specific batching and compression of IoT messages on-the-fly, Offload and Comprex provide a geomean 10.6× and 13.5× increase in performance over Baseline. Unlike the CPU-only Baseline plus Batch, neither Offload plus Batch nor Comprex show a significant increase in latency since the entire critical path is accelerated on the

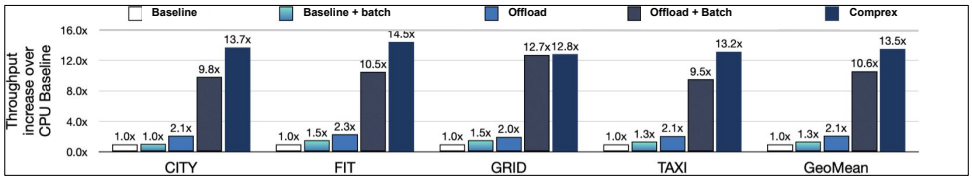

FIGURE 8. Throughput comparison.

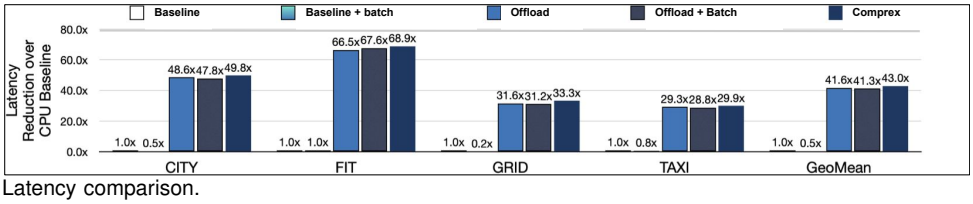

FIGURE 9. Latency comparison.

SmartNIC to generate a reply before applying batching and compression. FIT sees the highest increase in throughput of 14.5× with Comprex since the size of the IoT message is highest in that benchmark. GRID sees the lowest throughput improvement of around 12.8× with Comprex since this benchmark does not benefit significantly from compression.

Scalability of Host Resources. We evaluate the end-to-end impact on the scalability of the host resources. For Baseline, we measure the CPU-side requirements to execute the critical path operations on the host and to pass the data to the long-term analytics component. In the Comprex case, the critical path is executed on the SmartNIC using a pipeline-like accelerator similar to NICA [3]; the host-side runtime component receives the compressed messages and performs decompression before delivering them to the rest of the application. In order to avoid measuring bottlenecks related to our current database or machine learning components, in these measurements the application is an in-memory log. When comparing Baseline and Comprex with respect to the per-core CPU utilization, at the same level of sustained throughput, Baseline utilizes 91-99% CPU, whereas Comprex requires only 11-12%. This is an 8× improvement in CPU efficiency.

Decompression performance. To compare the performance of the Comprex and Snappy decompression parts, we created two dump files with four million compressed messages from the four IoT applications. Only one CPU core was used for this micro-benchmark. As illustrated in Figure 10, Comprex and ComprexDD are heavily penalized by cold start during which **Comp.Diff** is executed for every message. As messages for the same topic start to repeat over time and static regions remain unchanged, calls to **Comp.Diff** are reduced, resulting in a ≈ 2× higher throughput compared to

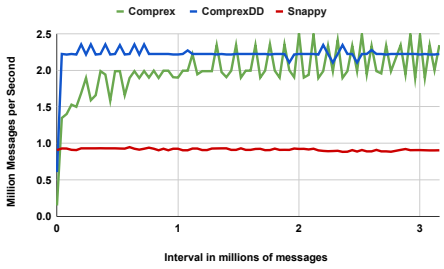

FIGURE 10. Decompression throughput comparison between Comprex, Comprex with offloaded **Detect.Diff** (DD) and Snappy.

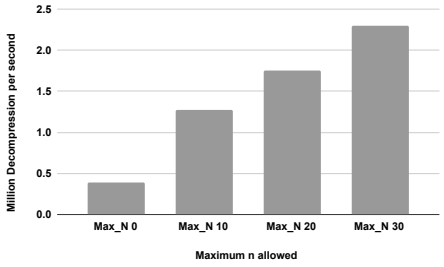

FIGURE 11. Performance measurement for different caps on how much **N** (interval in number of messages) can grow before **Comp.Diff** is executed.

Snappy. The wavy behavior of the Comprex throughput can be attributed to the overhead of detecting a change in the static region. It not only resets the **Max_N** for the given topic, but also forces the Compress Engine to transfer previously stored messages to the host side. However, if the Compress Engine is equipped with **Detect.Diff**, no cap is added to **Max_N** which allows

the confidence in the static regions to grow indefinitely.

Furthermore, since **Detect.Diff** can detect drifts in the static regions on a per-message basis, no previous messages need to be stored and transferred to the host. This greatly reduces the impact of slight variations in messages by reducing the number of calls to **Comp.Diff**. Lastly, if a topic does not have static regions that remain unchanged for at least five messages, Comprex reduces the frequency at which **Comp.Diff** is executed until topic is completely removed from consideration.

Throughput and Lossiness. Figure 11 illustrates the impact of different caps imposed on **N**. If **Max_N** is 0, **Comp.Diff** is executed on every message, greatly reducing the benefits of Comprex. When **Max_N** is set to 10, Comprex can perform more than 1M decompression per second. Above **Max_N** 30, the benefits are negligible. As seen earlier, **Max_N** also indicates how many messages must be stored in the SmartNIC DRAM for Comprex to be lossless.

Comprex vs ComprexDD. Since ComprexDD checks every message for variations on static regions, achieving lossless compression requires storing only one full message per *topic*. As such, while both versions achieve comparable throughput, Comprex consumes **Max_N** times memory per *topic* compared to ComprexDD.

Comprex allows users to determine how much **Max_N** can drift from **Max_Store** before the SmartNIC dumps stored messages at the host's request. For the microbenchmark in Figure 10, no data were lost with **Max_N** and **Max_Store** set to 30 and 10 respectively, even without the **Detect.Diff** stage.

Reduction in Average Latency. Figure 9 compares the reduction in the average latency of Baseline, Baseline plus Batch, Offload, Offload plus Batch, and Comprex compared to Baseline for each of the benchmarks. As shown in the figure, batching has an adverse impact on the latency for the critical path and increases the critical path latency. Offload, Offload plus Batch, and Comprex all use the SmartNIC to yield a significant reduction in latency by completely offloading the operations of the critical path. The FIT benchmark sees the highest reduction in critical path latency from offload since this benchmark has the highest number of parameters used for computations of the critical path. The GRID and TAXI benchmarks see the lowest reduction in latency since these benchmarks have fewer parameters that are processed for the critical path. Nevertheless, all benchmarks see more than 28.8× reduction in latency.

None of the benchmarks show a significant degradation in latency from batching in Offload plus Batch and batching+compression in Comprex, compared to Offload, since batching is performed outside of the critical path of the message latency. This illustrates how Comprex significantly raises end-to-end stable throughput while not having a negative impact on latency for the critical path.

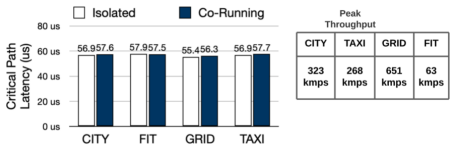

FIGURE 12. Latency for critical path when running a single vs. multiple applications.

Co-running multiple applications. Figure 12 shows the impact of accelerating multiple applications with our system. Isolated execution accelerates only the topics for a single application, shown on the x-axis. Each application (eg. CITY) consists of 5,000 topics, where each topic performs the exact same computations. Co-running execution accelerates multiple applications by randomly interleaving messages from topics in all four benchmarked applications.

As Figure 12 shows, there is no noticeable degradation in latency when co-running multiple IoT applications. The slight variation in latency reported in the figure stems from the networking stack and software overheads when measuring latency. The actual variation in latency within the SmartNIC measured via simulations is less than a tenth of a μs ($\sim$ 10 cycles @150MHz) between the application with the most computations FIT, and the application with the least computations TAXI.

FPGA utilization. We are able to support all four applications using around 2% LUTs and 3% BRAMs resources available in the Xilinx XCKU060 FPGA on the Mellanox SmartNIC.

Related Work

IoT Specific Compression and Batching Several works have demonstrated the benefits of compressing and batching IoT messages at the edge [7]. However, naive adoption of these techniques for accelerated SmartNIC-based IoT applications puts the decompression and batching operations in the critical path, making them not suitable for IoT applications with real-time processing requirements. While Comprex [8] Compress Engine is only capable of removing static regions, ComprexDD leverages pre-computed information done by the host side to perform string-compare on the in-network accelerator at line-rate. This not only greatly reduces memory footprint but also offloads computations to specialized hardware.

SmartNIC acceleration frameworks. A number of related efforts have explored the benefits of in-network acceleration or developed programming interfaces for computation offload to SmartNICs. NICA [3] provides a framework for general server application acceleration on FPGA-based SmartNICs. It uses a low-level socket interface for steering traffic to accelerators. sPIN [4] and INCA [10] provide programming models for in-line message processing in HPC systems. INCA uses tag-matching for selecting instructions, analogously to Comprex topic abstraction. COPA [6] presents a generic architecture for integration of inline or lookaside accelerator cores with the SmartNIC packet processing path.

The work presented in this paper is complementary to those efforts, as it focuses on further amplifying the benefits of SmartNIC offload via more effective compression and batching, as shown by the differences between Comprex and offload measurements in Figure 8.

Conclusion

The rapid increase in the number of IoT devices and the real-time ultra-low latency requirements of new 5G applications challenges IoT service providers. Solutions that combine offload to SmartNICs and acceleration of common latency critical operations present in these workloads, provide significant benefits. We present a solution that amplifies those benefits through use of an IoT-specific compression and batching engine, that further improves the efficiency and scalability of the IoT analytics server platforms. Our results with 4 real-world applications enable offloads to SmartNIC to reduce critical path latency by 43× while increasing the stable throughput 13.5×, and show 8× increase in CPU efficiency. The solution also paves the way for integration of other, potentially lossy, compression techniques which leverage the IoT message structure to provide further benefits.

Acknowledgent

We thank the anonymous reviewers for their valuable feedback. Hardik Sharma, Haggai Eran, Hadi Esmaelizadeh and Mark Silberstein helped in various stages of this project. This work was partially supported by NSF awards SPX-1822972 and CNS-2016701, and by the ADA and PRISM centers, via the joint SRC and DARPA JUMP programs.

REFERENCES

1. B. Abali and B. B. et al, “Data Compression Accelerator on IBM POWER9 and z15 Processors : Industrial Product,” in 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), 2020, pp. 1–14.
2. S. Anshu, C. Shilpa, and S. Yogesh, “Riotbench: A real-time iot benchmark for distributed stream processing platforms,” TPCTC, 2016. [Online]. Available: https://arxiv.org/pdf/1701.08530.pdf
3. H. Eran, L. Zeno, M. Tork, and M. Silberstein, “NICA: An infrastructure for inline acceleration of network applications,” atc, 2019.
4. T. Hoefler, S. Di Girolamo, K. Taranov, R. E. Grant, and R. Brightwell, “sPIN: High-performance streaming processing in the network,” in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. ACM, 2017, p. 59.
5. B. E. S. Jiaxin Lin, Kiran Patel, “Panic: A high-performance programmable nic for multi-tenant networks,” USENIX, 2020.
6. V. Krishnan, O. Serres, and M. Blocksome, “Configurable Network Protocol Accelerator (COPA) - An Integrated Networking/Accelerator Hardware/Software Framework,” in Hot Interconnects’20, 2020.
7. T. Lu, X. Zou, Q. Xia, and W. Xia, “Adaptively compressing iot data on the resource-constrained edge,” hotedge, 2020.
8. R. Oliveira and A. Gavrilovska, “In-network compression for accelerating iot analytics at scale,” in 2023 IEEE Symposium on High-Performance Interconnects (HOTI), 2023, pp. 15–24.
9. M. Satyanarayanan, “The emergence of edge computing,” Computer, vol. 50, no. 1, pp. 30–39, 2017.
10. W. Schonbein and R. E. e. a. Grant, “INCA: in-network compute assistance,” in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2019, pp. 1–13.

Rafael Oliveira is a PhD candidate in the School of Computer Science at Georgia Tech, specializing in system support for in-network acceleration of IoT applications.

Ada Gavrilovska is associate professor in the School of Computer Science at Georgia Tech. Her research is supported by the National Science Foundation, the US Department of Energy, the JUMP programs by the Semiconductor Research Corporation and DARPA, and a number of industry grants.