Stream Aggregation with Compressed Sliding Windows

Citation for the original published paper (version of record):
Ramakrishnan Geethakumari, P., Sourdis, I. (2023). Stream Aggregation with Compressed Sliding Windows. ACM Transactions on Reconfigurable Technology and Systems, 16(3).
http://dx.doi.org/10.1145/3590774

Stream Aggregation with Compressed Sliding Windows

PRAJITH RAMAKRISHNAN GEETHAKUMARI and IOANNIS SOURDIS, Computer Science and Engineering Department, Chalmers University of Technology, Sweden

High performance stream aggregation is critical for many emerging applications that analyze massive volumes of data. Incoming data needs to be stored in a *sliding window* during processing, in case the aggregation functions cannot be computed incrementally. Updating the window with new incoming values and reading it to feed the aggregation functions are the two primary steps in stream aggregation. Although window updates can be supported efficiently using multi-level queues, frequent window aggregations remain a performance bottleneck as they put tremendous pressure on the memory bandwidth and capacity. This article addresses this problem by enhancing StreamZip, a dataflow stream aggregation engine that is able to compress the sliding windows. StreamZip deals with a number of data and control dependency challenges to integrate a compressor in the stream aggregation pipeline and alleviate the memory pressure posed by frequent aggregations. In addition, StreamZip incorporates a caching mechanism for dealing with skewed-key distributions in the incoming data stream. In doing so, StreamZip offers higher throughput as well as larger effective window capacity to support larger problems. StreamZip supports diverse compression algorithms offering both lossless and lossy compression to integers as well as floating-point numbers. Compared to designs without compression, StreamZip lossless and lossy designs achieve up to 7.5× and 22× higher throughput, while improving the effective memory capacity by up to 5× and 23×, respectively.

CCS Concepts: • **Computer systems organization → Reconfigurable computing**; • **Information systems → Data compression**; **Stream management**; • **Hardware → Hardware accelerators;**

Additional Key Words and Phrases: Compression, dataflow, aggregation, sliding windows, stream processing

ACM Reference format:
Prajith Ramakrishnan Geethakumari and Ioannis Sourdis. 2023. Stream Aggregation with Compressed Sliding Windows. *ACM Trans. Reconfig. Technol. Syst.* 16, 3, Article 37 (June 2023), 28 pages.
https://doi.org/10.1145/3590774

1 INTRODUCTION

The massive volumes of data produced globally enable a large number of emerging stream processing applications [8]. Such applications are used in various domains, e.g., financial and transportation, to analyze large unbounded streams of data and make fast, sophisticated decisions. However,

This article is an extension of our conference paper, "StreamZip: Compressed Sliding Windows for Stream Aggregation," *International Conference on Field Programmable Technology (ICFPT) 2021*, Auckland, New Zealand, Dec. 6–10, 2021, 203–211 [11]. The authors would like to thank Maxeler Technologies for providing infrastructure for running the experiments. This work was partly funded by the Swedish Research Council under the ScalaNetS project (2016-05231) and the Swedish Foundation for Strategic Research under the PRIDE project (CHI19-0048).
Author's address: P. Ramakrishnan Geethakumari and I. Sourdis, Computer Science and Engineering Department, Chalmers University of Technology, Rännvägen 6, 41296, Gothenburg, Sweden; emails: {ramgee,sourdis}@chalmers.se.

1936-7406/2023/06-ART37 $15.00
https://doi.org/10.1145/3590774

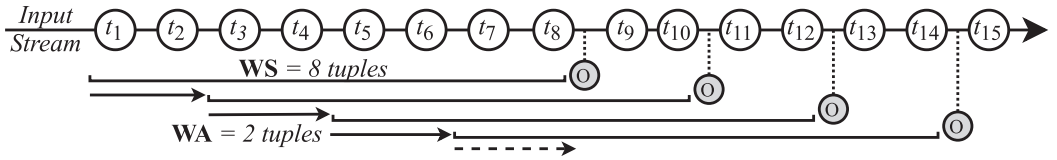


Fig. 1. Sliding-window stream aggregation with Window Size (WS) = 8 tuples and Window Advance (WA) = 2 tuples for an input data stream, e.g., a vehicular sensor emitting tuples $t_1, t_2, \ldots$ with each tuple containing a timestamp, vehicle ID, and speed. For simplicity, only tuples from a single vehicle (key) are shown. The grey tuples indicate the aggregate output generated when the sliding window gets full (e.g., `top-3`, `average`, and `median` speed).

consuming large data volumes at line rates requires *high processing throughput* and sometimes, e.g., in financing, *low latency*.

Stream aggregation is one of the most challenging tasks in stream processing. An example is depicted in Figure 1. It can be described by applying the traditional relational database aggregation semantics to a sliding window. Such a **window of size (WS)** is updated with incoming elements (values carried by incoming tuples). Upon aggregation, the window "slides" by a particular number of elements (**Window Advance (WA)**) to produce the aggregated values, that is, the window contents before sliding [1]. The aggregated values are subsequently fed to one or multiple functions that compute an output every time the window slides. Considering a key-value pair system, incoming tuples carry values of different keys, which are aggregated separately using a separate sliding window per key. This description fits **Sliding-Window stream Aggregation (SWAG)** that follows a tuple-based window policy, meaning WS and WA are measured in terms of the count of elements. An alternative windowing policy is time based, where the size and slide are defined by time intervals.

For some problems, the sliding-window aggregations can be simplified by computing them incrementally [19, 22, 24]. However, many others need to follow the ***Single sliding-window* stream aggregation (Single-SWAG)** approach [9, 27], which is the focus of this article. That is the case for problems that use *non-associative* aggregation functions, which cannot be computed incrementally, e.g., `median` [12], or problems that would be more expensive to compute incrementally than using Single-SWAG, e.g., frequent aggregations of multiple aggregation functions in geo-tagged data [16], social media data [14], or manufacturing equipment data [13].

Single-SWAG is a memory-intensive problem [9]. The memory needs to be accessed to update the window for each incoming tuple, and, when ready for aggregation, the entire window should be read. These two Single-SWAG steps, *window update* and *window aggregation*, generate tremendous memory pressure, the latter especially for queries with frequent aggregations (small WA). Current state-of-the-art FPGA-based Single-SWAG approaches employ **dataflow engines (DFEs)** and mitigate the memory bandwidth bottleneck due to window updates using **multi-level queues (MLQs)** [10]. However, this only addresses the first Single-SWAG step, i.e., window updates. For queries with frequent aggregations, as the dominant part of the single window rests in the farthest and slowest memory, previous approaches suffer from low processing throughput. This is due to the memory bandwidth bottleneck posed by the large volume of window aggregation traffic.

One way to alleviate the window aggregation memory bottleneck is to compress the sliding window. Existing studies on real-world streaming datasets have shown that a dominant part of them consisting of performance counter, sensor, geolocation, and other time-series data have significant redundancy that can be exploited through data compression [25]. As an example, Figure 2 illustrates the tremendous potential gains in Single-SWAG processing throughput by reducing the tuple's value size, e.g., up to 28× higher throughput with a 32× data reduction. However, compression

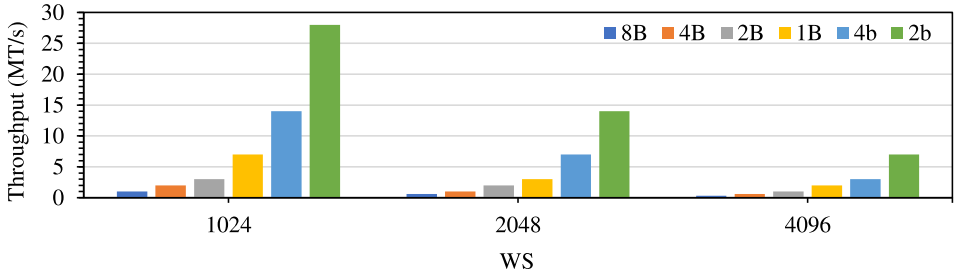


Fig. 2. Processing throughput in million tuples per second vs. WS in tuples for an FPGA-based MLQ Single-SWAG dataflow engine [10] at 156.25 MHz with varying value sizes and WA = 1 tuple, showing the potential for data compression.

complicates window management and introduces dependencies. Moreover, due to the on-the-fly processing, high-throughput, and low-latency requirements of stream processing, sophisticated compression schemes cannot be afforded due to their high complexity and high latency.

In the past, compression has been proposed for stream processing in TerseCades [25]. However, TerseCades only supports batch processing of separate non-overlapping tumbling windows, i.e., WA = WS, rather than true stream processing with WA ≤ WS; therefore, it avoids data overlap between different window instances and hence avoids the greatest challenge of applying compression to SWAG. In addition, TerseCades is software based and requires data to be stored in memory before processing, introducing significant latency. An interesting contribution of that work is the support for processing directly on compressed data, which requires support by both the compression algorithm and the aggregation function and is orthogonal to our work.

This article is an extension of StreamZip, which is the first true stream processing engine with compression support for sliding windows and WA ≤ WS [11]. StreamZip is based on previous FPGA-based MLQ DFE for SWAG systems and is able to support lossless and lossy compression algorithms aiming to mitigate the memory bandwidth bottleneck of window aggregations. StreamZip achieves this by addressing a number of concurrency control challenges introduced by the addition of the compression and decompression steps to the pipeline. This is especially true for skewed key distributions in which bursts of incoming tuples of the same key (or small subset of keys) enter the system in close succession and is prevalent in real-world datasets. The stream aggregation pipeline needs to ensure it generates correct results despite tuples of the same key being processed in parallel at different stages and without hampering the performance of the system. Moreover, frequent aggregations in skewed key distributions create redundant memory accesses that waste bandwidth and limit performance; this performance hit is exacerbated by high-latency decompressors. StreamZip devises efficient concurrency control mechanisms to mitigate such data and control dependencies in the pipeline. As a result, StreamZip achieves substantial reduction of aggregated data volumes and offers up to an order of magnitude higher processing throughput and reduction in effective memory capacity compared to current state-of-the-art Single-SWAG systems.

The contributions of this article extending StreamZip [11] are the following:

- Adds support for XOR compression algorithm to support lossless floating-point compression showing the applicability of Streamzip for diverse compression schemes and handling the challenges posed by different characteristics of the compression algorithm
- Devises efficient concurrency control schemes such as *independent compressed blocks* and *compressed block interleaving* to mitigate the sequential dependency in the compression algorithm

Table 1. Data Types and Compression Schemes in StreamZip

Data Types/Compression	Lossless	Lossy
Fixed point	Base-Delta	SZ
Floating point	XOR	SZ

- Achieves up to 5.6× compression ratio and up to 7.5× higher processing throughput for the lossless floating-point design using XOR compression than current state-of-the-art FPGA-based MLQ stream processing engine
- Introduces a caching mechanism to handle skewed key distributions without penalizing performance and achieves up to 3× higher processing throughput compared to Streamzip without caching

The remainder of this article is organized as follows. Section 2 offers background about compression and discusses related work. Section 3 describes the StreamZip design. Section 4 presents the evaluation and compares StreamZip with related works. Finally, Section 5 summarizes our conclusions.

2 BACKGROUND AND RELATED WORK

This section offers background on data compression in relation to stream processing and discusses related work.

2.1 Compression Algorithms

FPGA-based Stream processing engines use deep dataflow pipelines to achieve high processing rates. As a consequence, they require fine-grained pipelining with as few data dependencies as possible. Adding compression to such a system introduces a number of challenges. Briefly, the challenges pertaining to the choice of compression algorithm are related to the arithmetic complexity of compression and the dependencies between computations of consecutive values. Other compression characteristics that require attention when designing a SWAG with compression are related to the nature of the algorithm, i.e., being lossy, thus introducing error, versus lossless, as well as the target compression ratio. Next, we discuss our choice of the diverse compression algorithms used in StreamZip as illustrated in Table 1, namely, the **Base-Delta (BD)** lossless algorithm applied to fixed-point numbers, XOR lossless algorithm applied to floating-point numbers, and the **Squeeze (SZ)** lossy algorithm applied to fixed- and floating-point numbers.

Base-Delta encoding is a simple but efficient lossless compression scheme that has been used for decades in computing systems [6, 31] and offers a competitive compression ratio for a wide range of fixed-point integer data, including performance counter, geolocation, sensor, and other time-series data. Briefly, a value is represented as an offset (Delta) from a constant (Base); if that is not possible, a new base is selected. Multiple bases can also be used. Figure 3 shows a stream of 8-byte values compressed with Base-Delta encoding to 1-byte deltas. As such, there are no dependencies in computing (compressing/decompressing) consecutive values and processing is minimal as a single addition is sufficient. StreamZip applies Base-Delta to fixed-point numbers.

The second algorithm used in StreamZip is based on the XOR lossless compression scheme for floating-point numbers [20, 26, 28]. Base-delta lossless encoding does not work well as it does not reduce the number of bits sufficiently for floating-point numbers and hence results in low compression ratios [25]. On the other hand, XOR is a reversible operation that turns identical bits into zeros. Since the sign, the exponent, and the top mantissa bits occupy the most significant bit positions in the IEEE 754 standard, the XOR result of close values would have a substantial number

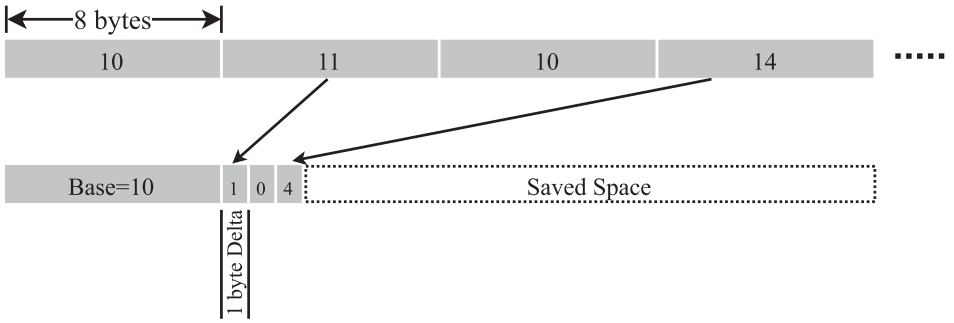


Fig. 3. Base-Delta compression of a stream of 8-byte values to 1-byte deltas.

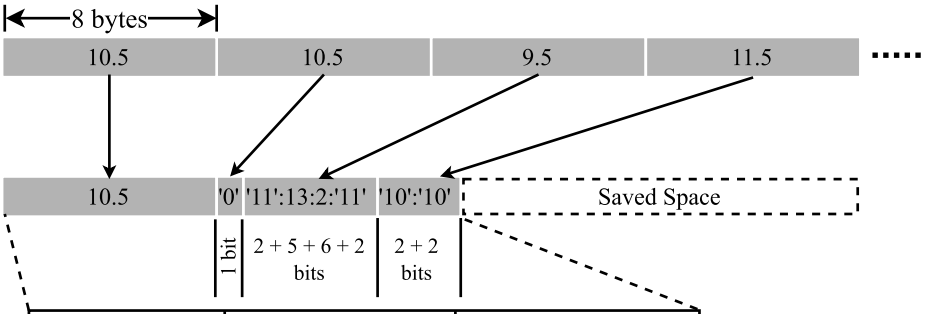
of leading zeros. Hence, it can be encoded and compressed by a leading zero count followed by the remaining bits. The incoming tuple's double-precision floating-point values are variable-length encoded using XOR compression [26] as below:

- The initial value is stored uncompressed.
- If XOR with the previous is zero, i.e., same value, a single "0" bit is stored.
- When XOR is non-zero, the number of leading and trailing zeros in the XOR are calculated and a "1" bit is stored followed by either:
  - Control "0" bit: If the number of leading zeros and trailing zeros is at least as many as in the previous XORed value, the block of meaningful bits falls within the block of previous meaningful bits. As a result, the block position information is reused and just the meaningful XORed value stored.
  - Control "1" bit: The number of leading zeros is stored in the next 5 bits followed by the length of the meaningful XORed value in the next 6 bits. Finally, the meaningful bits of the XORed value are stored.

Figure 4 shows the XOR compression in action over a stream of 8-byte floating-point values. Both the previous floating-point value and the previous XORed value are utilized in XOR compression. This leads to an additional compression factor as the number of leading and trailing zeros are very similar in a sequence of XORed values in time-series data [26]. There has been a recent study analyzing the efficacy of XOR compression algorithms for **Time Series Management Systems (TSMSs)**, which further proposes an updated version of the above XOR compression algorithm called Chimp [20]. This study offers a detailed performance evaluation of both Gorilla [26], which is the state-of-the-art lossless compression algorithm for floating-point numbers currently employed in most widely used TSMSs, and Chimp for different datasets, offering compression ratios up to 5.4x. Note that in this version of StreamZip, the Gorilla XOR compression algorithm is implemented, but StreamZip has the flexibility to adapt the dataflow-based stream processing pipeline to incorporate diverse streaming compression algorithms that work well for different time-series data types.

The third algorithm used in StreamZip is the lossy SZ for both fixed- and floating-point numbers. SZ compresses a sequence of numeric values by describing each value X_i as a function of the three preceding values $[X_{i-3}, X_{i-2}, X_{i-1}]$, according to a predefined function model [4]. A typical SZ supports four such function models (to be encoded with 2 bits per value) [5] as shown in Figure 5(a), namely:

- a *constant* value is approximated as equal to the nearest preceding one (Equation (1)),
- a *linear* value is extrapolated from the preceding two values (Equation (2)),



Decimal Value	Hex-representation	XOR with previous
10.5	0x4025000000000000	
10.5	0x4025000000000000	0
9.5	0x4023000000000000	0x0006000000000000
11.5	0x4027000000000000	0x0004000000000000

Fig. 4. XOR lossless compression of a stream of 8-byte floating-point values. Encodings in quotes represent actual bits. XORing the same floating-point values results in 0 and is encoded with a single "0" bit header. The third (9.5) and second (10.5) values are different and there are only 2 meaningful bits in the XOR. This is encoded with a 2-bit header ("11"), 13 leading zeros, 2 meaningful bits, and the actual XORed meaningful value ("11"). XORing the fourth (11.5) and the third (9.5) values in the stream shows that the meaningful bits fall within the block of the previous meaningful bits. This is encoded with just a 2-bit header ("10") and the actual meaningful value, "10."

- a *polynomial* value fits on the cubic curve described by the preceding three values (Equation (3)), or
- if none of these models describes a value with an acceptable error, the value is an *outlier* and stored explicitly. To start, the sequence the first three values are stored uncompressed (*seeds*).

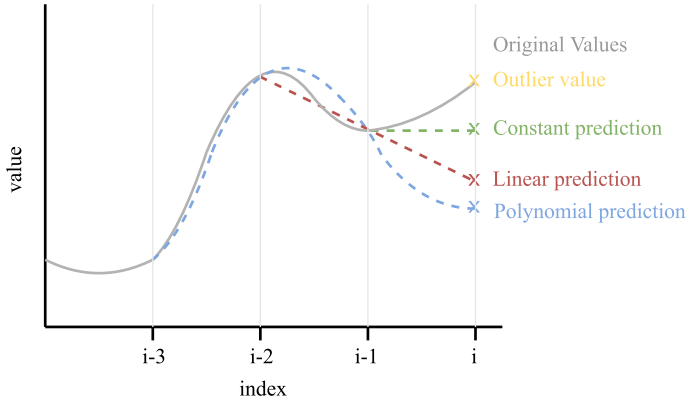
$$X_i^C = X_{i-1} \tag{1}$$

$$X_i^L = 2X_{i-1} - X_{i-2} \tag{2}$$

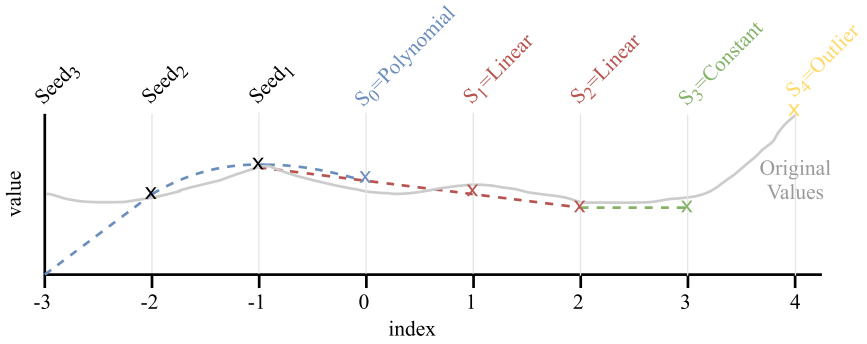
$$X_i^P = 3X_{i-1} - 3X_{i-2} + X_{i-3} \tag{3}$$

It is clear that computing a value depends on the computations of its previous three, and hence the computing complexity is important. In case computing a value takes more than a cycle, then throughput can be limited. As shown by Eldstål-Ahrens and Sourdis, computing a value can be reduced for the most complex function (polynomial) to two pipeline stages, each with a 3-operand addition [5]. To stress the system, SZ is applied to floating-point numbers. As a result, compressing or decompressing a value requires multiple cycles. In general, SZ introduces an error that can be controlled by the compressor; i.e., if the error is too high an outlier is created, but it offers roughly 4× higher compressibility than lossless compression schemes. Figure 5(b) shows SZ decompression by reconstructing the value stream using the initial set of *seeds*, the compressed *symbols,* and the explicit *outlier* values stored.

In the past, SZ has been used in the GhostSZ FPGA-based system for I/O compression [35]. GhostSZ breaks the dependencies between computing consecutive values by splitting them into multiple parallel sequences that are compressed separately, thereby increasing throughput.



(a) Four function models in SZ compression.



(b) Reconstruction of SZ compressed value stream.

Fig. 5. SZ compression.

2.2 Stream Processing Platforms

In this section, we discuss related works that use different computing platforms for stream processing. Various computing alternatives have been used for stream processing and stream aggregation in particular, each having different functionality and performance potential.

Distributed **Stream Processing Engines (SPEs)** running on conventional CPUs like Apache Flink, Spark, and Storm provide generic stream processing capabilities and ease of deployment [7, 15, 32, 36]. These software-based distributed stream processing engines are easy to configure, are flexible to allow for a multitude of operations and analysis on the data, and can process a large amount of data located in different servers. Nevertheless, as with any general-purpose software implementation, their performance depends on the underlying hardware and can never match the throughput or latency offered by dedicated implementations (e.g., custom FPGA-based systems). In the particular focus of this work, software approaches are not able to cope with the challenges posed by non-incremental group-by aggregation on large windows (large WS), at high rates (small WA), and with large number of keys.

Multicore CPU and GPU-based stream processing systems are able to sustain high processing throughput but have wasteful memory management as they require redundant memory accesses to store incoming tuples from the network to DRAM even before processing starts [23]. Some

CPU-based approaches propose algorithmic modifications to reduce latency but are constrained to only associative aggregation functions [29, 33]. StreamBox-HBM exploits high-bandwidth HBM memories to improve the processing throughput [21]. TerseCades achieves the same goal with compression [25]. However, both StreamBox-HBM and TerseCades only support aggregation queries with tumbling windows (WS = WA) and not high-frequency aggregation with small WA.

The most popular GPU-based stream processing systems are SABER [18] and FineStream [37], which perform only incremental aggregations, and Gasser [3], which supports non-incremental aggregations but supports queries with only a single key, hence small problem sizes. Moreover, these GPU-based stream processing systems incur a high latency of hundreds of milliseconds for aggregation queries mainly due to the unnecessary data movement requiring redundant memory accesses. Even though FineStream utilizes the CPU-GPU integrated architectures, which avoid the data movement between the two memory hierarchies via PCI-e, it still suffers the data movement of the tuples from the network interface to the memory before processing. Moreover, these GPU systems buffer input tuples in batches and then offload the processing of the batches on the GPU, rather than processing each incoming tuple in a true streaming fashion, which could negatively impact the processing latency.

On the contrary, FPGA-based stream processing systems with customized deep pipelines (as well as the direct network connection) *minimize data movement,* offering high-throughput and low-latency stream aggregation. FPGA-based designs can specialize their memory hierarchy and customize their compute kernels to the specific aggregation query at hand [10]. Thereby, they achieve better performance and energy efficiency compared to a GPU system, and in some cases support types of compute functions (i.e., holistic) and problem sizes (WS x #keys) currently not available in existing GPU systems [9, 27]. As a consequence, some FPGA-based systems deliver both high processing throughput, on par with the GPU and CPU systems, and ultra low latency in the order of tens of microseconds [9, 22, 24], at least 3 orders of magnitude lower than CPUs and GPUs.

FPGA designs based on incremental aggregation algorithms do not support *non-associative* functions [19, 24]. Recently, there has been work on FPGA-accelerated approximable query processing using sketches [2], which maintains statistics rather than explicit values and thus can only provide estimates. On the other hand, FPGA-based single window stream aggregation for tuple-based [9] and time-based [27] windowing policies explicitly stores values and supports generic aggregation functions (including non-associative), varying window slides, large window sizes, and a large number of keys. However, these works use only DRAM for maintaining the single window state, which requires slow and bandwidth-wasteful read-modify-writes for window updates. Recently, an MLQ approach for single window aggregation was proposed to offer faster window updates [10]. MLQ constructs logical queues for storing sliding windows composed of BRAM, off-chip SRAM, and DRAM. The tail of the queue is always in the BRAM offering high bandwidth window updates, i.e., enqueues, thereby improving performance. However, window aggregations are limited by the available memory bandwidth. StreamZip builds on top of MLQ and compresses sliding windows to reduce their size and alleviate the memory bandwidth pressure of the aggregation step, thereby improving processing throughput.

3 STREAMZIP DESIGN

StreamZip is a reconfigurable, stream aggregation DFE with compression support for sliding windows that reduces the volume of aggregated data, improving processing throughput and effective memory capacity. It can be reconfigured to support different queries or aggregation problems based on the application at hand. As in MLQ [10], StreamZip windows are stored in multi-level queues, but in this case, they are compressed. Figure 6 illustrates the StreamZip pipeline. Incoming tuples

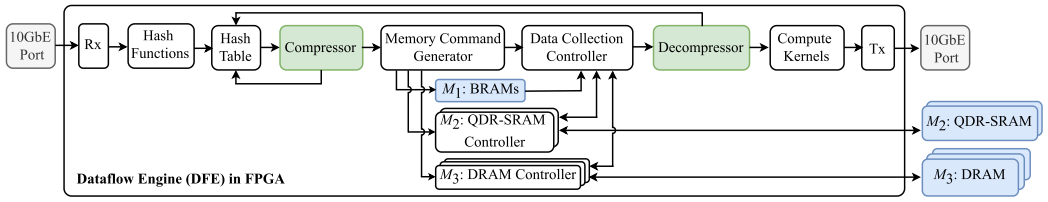


Fig. 6. Top-level view of the StreamZip compression pipeline for dataflow-based stream aggregation.

of the form $\langle ts, key, value \rangle$ are carried by network packets and received by the receiver module (Rx). The key of each tuple is first hashed to the hash table. Multiple hash functions are used to reduce collisions [17]. Each hash table entry corresponds to a key and stores metadata needed for *compression*, *multi-level memory management*, and *decompression* of this key's incoming tuples.

After the hash table stage, the tuple's value is compressed and memory commands are generated to update the window at each memory level based on the metadata state. The compressor also feeds back the compressed state to the hash table to update the metadata. The data collector acts as a buffer to synchronize the dataflow of the compressed single-window compartments from the various memory levels. On aggregation trigger, the entire compressed single window of the key is streamed to the decompressor and the decompressed values are fed to the compute kernel(s), where the aggregation function(s) are computed. The decompressor feeds back the information regarding the invalid values evicted upon window-slide following the aggregation to the hash table stage. The result of the aggregation function is finally transmitted back to the network through the Tx module. The dataflow between the stages is controlled through FIFOs, which stall the pipeline via back-pressure when necessary.

3.1 Hash Table Organization

The hash table stores the metadata required for managing the window update and aggregation steps of single-window stream aggregation. In our implementation, the table is direct-mapped and uses dual-port BRAMs with a depth chosen to support a large number of concurrently active key entries. Alternatively, associativity can be added to the hash table to reduce collisions [17]. Each entry of the hash table corresponds to a single key and stores various fields separated in three banks for the management of the sliding window, the compressor, and the decompressor as shown in Figure 7.

3.1.1 Window Management Bank. The window management bank contains (1) a valid bit to indicate whether the hash table entry is valid; (2) the key assigned to the entry; (3) tuple counter, t_c, to determine when the key is ready for aggregation for tuple-based windows; (4) window start timestamp, ts, for key replacement in case of collision and to trigger aggregation in time-based windows; (5) optionally, starting stream number ($\#stream$), used in case a key's compressed stream is divided into multiple sub-streams to account for the compressor and decompressor pipeline latency, which is described in Section 3.4.1; and (6) outstanding aggregation counter, O_a, which stores the count of the outstanding aggregations triggered per key used to solve dependency between consecutive aggregations and discussed in Section 3.4.2.

3.1.2 Compressor Bank. The compressor bank contains (1) current seed(s), denoted by S_C, used to compress the incoming tuple's value; (2) write pointers to each memory level, w_i, pointing to the tail index of each memory level for inserting the upcoming compressed value; and (3) an extra write pointer to M_2, denoted by w_{2S}, pointing to the index to store the updated base upon BD or the outlier upon SZ compression. The extra pointer is needed for alignment reasons as the larger

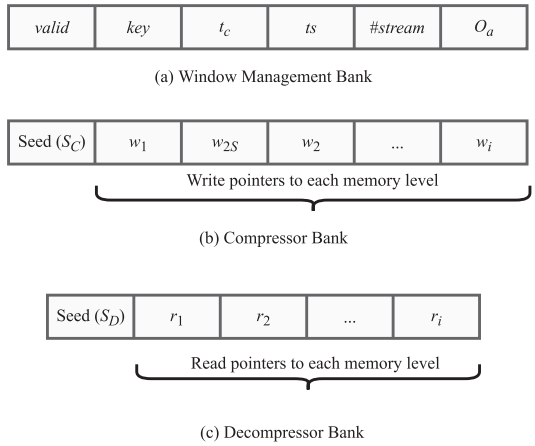


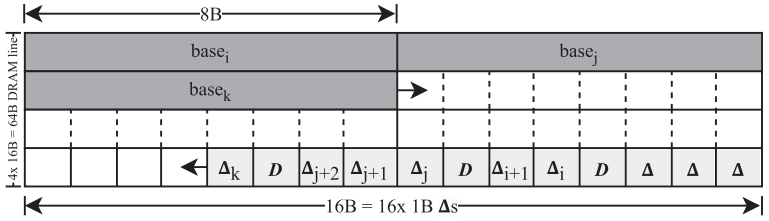
Fig. 7. Hash table organization.

base or outlier values are stored from the most significant side of a memory line, as opposed to the smaller compressed symbols, which are stored staring from the least significant side of the block. This extra pointer is not needed for XOR compression as the scheme involves variable-length encoding and the compressed symbols are written in a continuous fashion in the memory hierarchy. The memory packing and alignment for the various compression schemes are explained in Section 3.2. In case of BD encoding, the current seed is the latest base utilized for the compression of the incoming stream. For XOR compression, the seed field stores the latest incoming value and the leading and trailing zeros from the latest XOR operation. Similarly, for SZ, as the upcoming compression depends on the computations of the preceding three values, the three latest selected predictions based on the function models are stored.

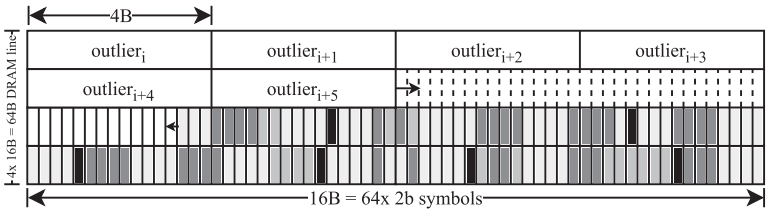
3.1.3 Decompressor Bank. The decompressor bank contains (1) starting seed(s), S_D, for decompressing the window, used to start the decompression of a window on aggregation trigger, and (2) read pointers to each memory level, r_i, pointing to the head index in level i from which to start reading the compressed values. The seeds stored here for the selected compression scheme are similar to the ones in the compressor bank, except that these seed(s) are the oldest ones from the start of the single window of a key used to initiate the decompression.

Partitioning the compressor and decompressor metadata into separate hash table banks enables to update them at different instances following the compressor and the decompressor pipeline stages, respectively. The hash table is accessed in a pipelined fashion using multiple hash functions [17]. First, the selected hash table entries are checked to match the requested key. In case of a miss, a new entry is made evicting the least recently used key out of the ones identified by the hash functions and the corresponding (de)compressor entries reset. If the evicted key is still active, a flag is raised indicating collision and the information is sent to software. The memory hierarchy can be used to extend the hash table and/or a software process could handle the keys that do not fit in the on-chip hash table due to collisions.

A hit in the hash table enables fetching the remaining metadata fields of the associated key from the compressor and decompressor banks. The compressor metadata determine (1) the current seed(s) needed for the compression of the incoming tuple's value; (2) the write address to insert the compressed value upon compression in M_1; (3) the address in M_2 based on w_{2S} to flush to, which is used only in case of BD and SZ if the compression results in a new base or an outlier, respectively; (4) indicate whether a memory level is full and needs to be flushed to the next level;



(a) Base-Delta block format of a StreamZip compressed sliding window



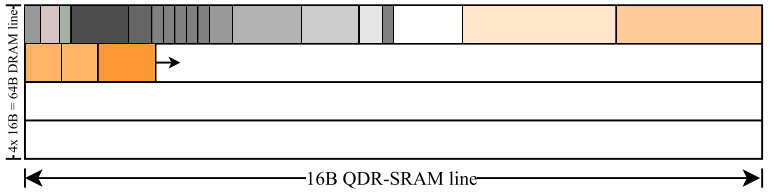
(b) SZ block format of a StreamZip compressed sliding window

Fig. 8. Block format of a compressed sliding window for two different compression algorithms: (a) Base-Delta fixed-point lossless compression. Bases, denoted by $base_*$, and deltas, denoted by Δ_*, are written from the most and least significant addresses, respectively. D represents the demarcation in the compressed sequence to denote a new base to be picked up from the most significant side. (b) SZ lossy floating-point compression. Outliers and symbols are written from the most and least significant addresses, respectively. Black symbol indicates an outlier and the three grey shades indicate constant, linear, and polynomial prediction compressed symbols.

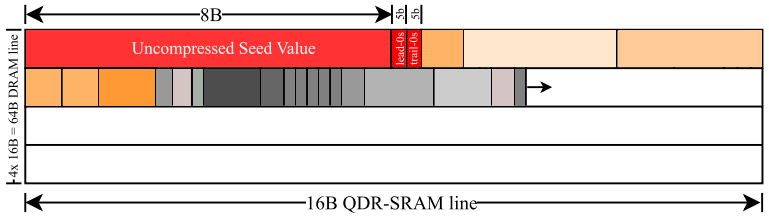
and (5) provide the write address of the successive level for insertion of the flushed block. If the window is full, the key is ready for aggregation and the metadata from the decompressor bank are used to determine (1) the starting seed(s) needed to decompress the window and (2) the read addresses of each memory level to be passed on to the Memory Command Generator for fetching the compressed single-window parts spread across the memory levels. In parallel to decompression, sliding the window requires a number of values (defined by WA) to be invalidated and evicted from the window. Accordingly, the window read pointers for each memory level are updated and fed back to the hash table. The detailed example of memory management and dataflow using the hash table is discussed in Section 3.3.

3.2 Memory Alignment and Packing

The format used for packing and storing the output of the compressor to the memory levels is key for the efficient utilization of the available memory space and bandwidth. It determines memory footprint of the window and the number of accesses required to read and update it. In our implementation, three memory levels are used, namely, on-chip BRAMs (M_1), off-chip QDR-SRAM (M_2), and off-chip DRAM (M_3), as shown in Figure 6. A compressed window is partitioned in blocks of fixed size defined by the access granularity of the last memory level (M_3/DRAM), which is 64B. A compressed block is denoted by CB. In order to maximize the number of keys supported for stream aggregation, BRAMs are mostly used for storing metadata, so M_1 stores only a few compressed values. Then, M_2 is configured to store one block of 64B per key matching the access granularity of M_3. The access granularity of M_2 is 16B, and so four M_2 lines compose one block that can be stored in one M_3 line. Figure 8 shows the format of a CB for BD and SZ compression algorithms



(a) XOR Compressed block format with dependent DRAM lines



(b) XOR Compressed Block format with independent DRAM line

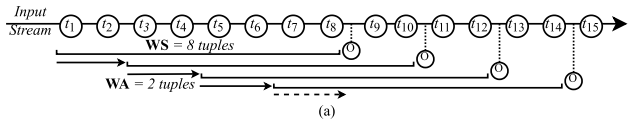
Fig. 9. Block format of StreamZip compressed sliding window for XOR compression: (a) for dependent DRAM lines and (b) for independent DRAM lines with starting uncompressed value (8B) and the leading (5b) and trailing (5b) zeros in red to aid in decompression by preventing the sequential dependency between consecutive compressed blocks. The various shades of yellow and grey in both formats indicate different compressed variable-length encodings generated by XOR compression and written continuously from the most significant side.

used in StreamZip. The size of an uncompressed value in a tuple is 8B. BD uses full-size Bases (8B) and Δs are set to 1B. SZ encoding uses half-precision outliers (4B) (if permitted within acceptable error thresholds) and compressed symbols of just 2 bits. Using half-precision seeds and outliers reduces the constant capacity overhead, enabling to support larger problem sizes. To simplify the data alignment in a block, we separate the data in a block based on their size. More precisely, values (bases or outliers) are stored starting from one end of the block and compressed values (Δs or 2-bit symbols) starting from the other end. This format simplifies block accesses during decompression, providing simpler address calculations for reading each type of data. It also packs data more efficiently in the available block space. Finally, it prevents multiple write accesses to M_2 due to misalignment, improving M_2 bandwidth utilization. In the case of BD compression, a demarcation, D, is used to denote a base change. The largest value of a Δ is reserved to represent such demarcation. For SZ, the demarcation is implicit as outliers are marked by an already reserved combination of the 2-bit compressed symbol.

On the other hand, XOR compression is variable-length encoded and therefore, the compressed data in a block cannot be separated based on their size. Instead, the compressed data is written in continuous fashion from the most significant side of the compressed block. This is as shown in Figure 9(a). Multiple write accesses to M_2 due to misalignment are avoided by writing the spilled-over compressed bits, if any on a flush, to the first level in the memory hierarchy, thereby improving the bandwidth utilization of the second level.

3.3 Memory Management and Dataflow

StreamZip memory management and dataflow are illustrated by the example in Figure 10. Here, lossless *base-delta* compression is described, but the dataflow is similar for other compression



(a)

tup: value	t_c	Compressor S_C	w_1	w_{2S}	w_2	w_3	Decompressor S_D	r_1	r_2	r_3	Memory Operations	M_1 M_2 M_3
Init	0	0	0	0	3	0	0	0	3	3	-	0
t_1: 50	1	50	0	0	3	0	50	0	3	3	-	-
t_2: 51	2	50	1	0	3	0	50	0	3	3	M_1-*Wu* (Insert Δ_2 in M_1)	1
t_3: 52	3	50	0	0	1	0	50	0	3	3	M_1-*Ru*; M_2-*Wu* (Flush M_1 with Δ_3 to M_2)	-
t_4: 53	4	50	1	0	1	0	50	0	3	3	M_1-*Wu* (Insert Δ_4 in M_1)	3
t_5: 54	5	50	0	0	4	0	50	0	3	3	M_1-*Ru*; M_2-*Wu* (Flush M_1 with Δ_5 to M_2)	-
t_6: 55	6	50	1	0	3	4	50	0	3	3	M_1-*Wu*; M_2-*Ru*; M_3-*Wu* (Insert Δ_6 in M_1; Flush M_2 to M_3)	5
t_7: 320	7	320	0	0	1	4	50	0	3	3	M_1-*Ru*; M_2-*Wu* (Flush M_1 with D to M_2)	-
t_8: 330	6	320	1	2	1	4	50	0	3	2	M_2-*Ra*; M_3-*Ra*; M_1-*Wu*; M_2-*Wu* (Aggregate t_1:t_8; Evict t_1:t_2; Insert $base_8$ in M_2; Insert Δ_8 in M_1)	10
t_9: 331	7	320	0	0	1	8	50	0	3	2	M_1-*Ru*; M_2-*Ru*; M_2-*Wu*; M_3-*Wu* (Flush M_1 with Δ_9 to M_2; Flush M_2 to M_3)	-
t_{10}: 332	6	320	1	0	1	8	50	0	3	0	M_2-*Ra*; M_3-*Ra*; M_1-*Wu* (Aggregate t_3:t_{10}; Evict t_3:t_4; Insert Δ_{10} in M_1)	12
t_{11}: 333	7	320	0	0	4	8	50	0	3	0	M_1-*Ru*; M_2-*Wu*; (Flush M_1 with Δ_{11} to M_2)	-
t_{12}: 580	6	580	1	0	3	12	50	0	3	6	M_2-*Ra*; M_3-*Ra*; M_1-*Wu*; M_3-*Wu* (Aggregate t_5:t_{12}; Evict t_5:t_6; Flush M_2 to M_3; Insert D in M_1)	D
t_{13}: 600	7	580	0	2	1	12	50	0	3	6	M_1-*Ru*; M_2-*Wu* (Flush M_1 with Δ_{13} to M_2; Insert $base_{12}$ in M_2)	-
t_{14}: 601	6	580	1	0	3	0	320	0	3	10	M_2-*Ra*; M_3-*Ra*; M_1-*Wu*; M_3-*Wu* (Aggregate t_7:t_{14}; Evict t_7:t_8; Flush M_2 to M_3; Insert Δ_{14} in M_1)	21
t_{15}: 602	7	580	0	0	1	0	320	0	3	10	M_1-*Ru*; M_2-*Wu* (Flush M_1 with Δ_{15} to M_2)	-

(b)

Fig. 10. (a) A stream of tuples $t_1, t_2, \ldots$ of a key aggregated with WS = 8 and WA = 2 tuples. (b) Base-delta compression metadata management in the hash table stage and compressed dataflow through the memory hierarchy for the above stream. The red numbers in the *Init* row indicate the indices of each memory cell. $base_i$ and Δ_i denote the base and delta corresponding to t_i. Δ requires only one memory cell and a base (value) requires two cells. M_1, M_2, and M_3 are configured to store up to 1 Δ, 4 Δs, and 8 values; M_i-*Wu*, M_i-*Ru*, and M_i-*Ra* denote writes due to window update, reads due to window update, and reads due to aggregation in memory level i, respectively. The green and blue highlights indicate flushes due to window update between memory levels and aggregation reads from each level, respectively.

choices too. As discussed in Section 3.1, t_c in the hash table stores the count of the tuples in the sliding window and is used to determine when the window gets full and trigger aggregation. In the compressor bank, S_C stores the current *base,* which is the seed to perform the base-delta compression on an incoming tuple. w_1, w_2, and w_3 are the write pointers to the tail indices of the first, second, and third memory levels, respectively, for updating the compressed window spanning across the memory hierarchy. w_{2S} is the extra write pointer to the second memory level pointing to the index to store the updated base as the larger bases are stored from the most significant side, as opposed to the smaller compressed deltas, which are stored starting from the least significant side of the compressed block, as shown in Figure 8(a). Finally, in the decompressor bank, S_D stores the starting base needed to initiate the base-delta decompression of the window upon aggregation trigger. r_1, r_2, and r_3 are the read pointers to the head indices of the first, second, and third memory levels, respectively, from which to start reading the compressed values and stream to the decompressor upon aggregation. Now let's discuss the dataflow in action.

On arrival of t_1 with value 50, t_c is incremented by 1. As this is the first tuple of the key, the compressor and decompressor partitions are set with $S_C = S_D = 50$ and the read and write pointers at each memory level are initialized. Note that for M_2, there are two write pointers to indicate the base side written from the most significant memory cell, denoted by w_{2S}, and the delta side written from the least significant cell, denoted by w_2. When t_2 arrives with value 51, t_c is incremented to 2 and the compressor takes as input the previous $S_C = 50$ and performs base-delta compression to produce $\Delta_2 = 51 - 50 = 1$. The compressor metadata partition is updated by incrementing w_1 by 1 as the Δ gets written to M_1. On arrival of t_3, M_1 is full, so it is flushed to M_2 along with the new $\Delta_3 = 52 - 50 = 2$. This causes w_1 to reset to 0 as M_1 gets empty and $w_2 = 3 - 2 = 1$ as the two Δs get written from the least significant side of the M_2 line. Similarly, for t_4, $\Delta_4 = 53 - 50 = 3$ gets written to M_1, and for t_5, M_1 gets flushed to M_2 with the new $\Delta_5 = 4$. When t_6 enters the system, $\Delta_6 = 5$ gets written to M_1, and as M_2 is full ($w_2 = 4$), it is flushed to M_3, which leads to resetting w_2 to 3 and w_3 points to the next line at index 4. Note that by using MLQ, wasteful read-modify-writes to M_3 (DRAM) are completely eliminated by writing first to the nearer and faster memories in the hierarchy (M_1 and M_2), ensuring faster window updates. Tuple t_7 carries a value of 320 and this causes a base change in the compressor. A base change is marked by a demarcation (D) on the delta side as described in Section 3.2. Then, S_C gets updated to 320, and M_1 is flushed along with D to M_2, leading to w_1 getting reset to 0, w_2 shifting to 1, and w_3 remaining at 4.

Arrival of tuple t_8 makes the window full ($t_c = 7 = \text{WS}-1$) and triggers aggregation, causing the compressed sequence $\{M_3[1, 2, 3, 4]; M_2[5, D]; M_1[320]\}$ to be read and passed to the decompressor with the starting seed (base), $S_D = 50$, and the current tuple, t_8:330. The decompressor, upon traversing the compressed sequence, marks the delta read index corresponding to the WA for *eviction* and feeds it back to the decompressor partition in the hash table stage. In this case, eviction of invalid tuples t_1: t_2 during window slide is marked by shifting r_3 to 2. In order to maintain the continuity in the compressed sequence for decompression, S_C=320 is written to M_2, leading w_{2S} to point to 2 as a base occupies two cells. The compressed $\Delta_8 = 10$ gets written to M_1, leading to $w_1 = 1$. w_2 and w_3 remain unchanged at 1 and 4, respectively. t_c is decremented by WA = 2, making the total tuple count in the single window 6. Now M_2 is full as the most significant side with base and least significant side with deltas have overlapped and on arrival of t_9, M_2 gets flushed to M_3. Similarly, on arrival of t_{10}, t_{12}, and t_{14}, aggregation gets triggered and $\{M_3[2, 3, 4, 5, D, 320]; M_2[10, 11]\}$, $\{M_3[4, 5, D, 320]; M_2[10, 11, 12, 13]\}$, and $\{M_3[D, 320, 10, 11, 12, 13]; M_2[D, 580, 20]\}$ get streamed to the decompressor. Note that on arrival of t_{14}, M_2 with $\{580, 20, D\}$ gets flushed to M_3, causing w_3 to wrap around and point to 0, thus maintaining the circular buffer per key, which is statically allocated in the memory hierarchy.

An interesting point to note here is that in this small example with an original value of size 2B (assuming a memory cell is a byte) being compressed to 1B using base-delta encoding, the aggregation reads from the farthest and slowest memory (M_3) have been reduced by half, thereby alleviating the memory bandwidth bottleneck. This enables StreamZip to achieve overall higher processing throughput. In addition, the effective memory capacity improves, leading to support of larger problem sizes (WS × number of keys).

3.4 Data Dependencies and Concurrency Control Mechanisms

There are various concurrency control challenges in adding compression support to a stream aggregation DFE pipeline without negatively impacting processing rate. Some of these challenges are generic and others are artifacts of particular compression characteristics. In general, the stream aggregation pipeline needs to ensure it generates correct results despite things happening in parallel at different stages. For example, one tuple may be updating the window while another tuple of the same key has already initiated an aggregation. In other words, the design needs to correctly handle data dependencies between tuples of the same key entering the pipeline in close succession, which is prevalent in real-world datasets with skewed key distributions. Skewed key distribution implies bursts of incoming tuples of the same key (or small subset of keys) entering the system in successive cycles. Besides making the pipeline longer, adding compression introduces the following challenges. First, it creates more dependencies between consecutive values, when the compression algorithm requires the previous values to generate the next. Second, a decompression triggered by an aggregation may depend on ongoing compression of values before it can start decompressing the window. Third, aggregations cause a window to slide and evict a number of values, but in a compressed window, evictions can be performed only after decompression; this creates a dependency between successive aggregations. Fourth, handling a compressed window stored in multiple memory levels makes data collection (upon aggregation) more complex than an uncompressed window. Finally, frequent aggregations in skewed key distributions create redundant memory accesses, which wastes bandwidth, and this performance hit is exacerbated with decompressors with high latency. Some of these challenges could be handled by pipeline stalls or bubbles; however, this would compromise performance. Below we discuss how StreamZip addresses these challenges without giving up performance.

3.4.1 Intra-(de)compressor Pipeline Dependency. There are two inefficiencies within the (de)compr-essor pipeline affecting the processing throughput of StreamZip that need to be addressed:

(i) (De)compressor latency due to arithmetic complexity: The varied arithmetic complexity of the compression algorithms needs to be dealt with. Base-delta requires a single addition to compress or decompress a value, and when applied to fixed-point numbers, this can be handled in a single cycle without affecting StreamZip's processing rate. Similarly, XORing two values and calculating the leading and trailing zeroes for XOR compression can be handled in a single cycle without affecting the processing throughput of StreamZip. On the contrary, SZ requires more complex computations and, applied to floating-point numbers, needs about 20 cycles at the operating frequency that supports line-rate processing. This may reduce processing throughput 20-fold if not addressed, since compressing or decompressing a value depends on completing the computation of the previous value first.

StreamZip deals with this using *value interleaving* similar to previous designs used in other domains [5, 35]. The incoming values of each key are divided into *multiple independent sub-streams* as shown in Figure 11. Then, a pipelined version of the SZ computations can handle these sub-streams without data hazards. This is performed by storing the initial seeds

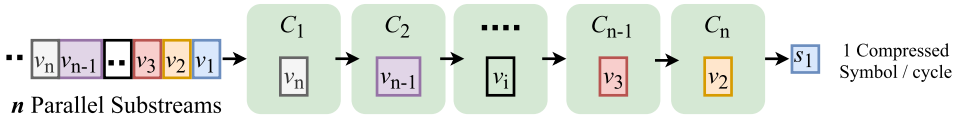


Fig. 11. *Value interleaving* to support multiple sub-streams in StreamZip. $C_1, \ldots, C_n$ denote the n pipeline stages of the (de)compressor. The various color-coded uncompressed values denoted by $v_1, \ldots, v_n$ represent the n substreams that can be interleaved and fed to the compressor to process one value every FPGA cycle to generate a compressed symbol, s_i, every cycle.

for (de)compression for each stream separately in the hash table stage and maintaining a sub-stream counter to identify the current active sub-stream for a key, as indicated in the window management bank of the hash table in Section 3.1. Although this is sufficient for maintaining correctness and high processing throughput, it complicates window aggregation and may affect compression quality. This is because, without sub-stream support, the data of each key are stored in a contiguous, statically allocated block in memory. Then, upon aggregation, a contiguous memory region is read with a single memory command, leading to efficient use of the DRAM bandwidth. Should these subs-treams store their data in non-contiguous memory blocks, accessing them would need multiple DRAM commands, resulting in reduced memory performance. To prevent this, StreamZip shares the same contiguous memory block for storing data of all sub-streams of a key. This simplifies window aggregation as only a single contiguous memory read is necessary, improving memory bandwidth utilization and hence processing throughput. Moreover, multiple decompressor pipelines are added, each handling a window of a different key, to cope with the processing rates. In our SZ StreamZip implementation, up to 128 parallel decompressor pipelines are used.

(ii) Sequential Dependency between Symbols in Compressed Blocks: At the decompressor, the *sequential dependency* between consecutive symbols in compressed blocks needs to be dealt with. In order to achieve the best possible processing throughput, the available memory bandwidth needs to be fully utilized without the decompressor becoming the bottleneck. The decompressor is fed with 64-byte compressed blocks from the memory, and in order to achieve the best processing throughput, the decompressor should be able to decompress the CBs at bandwidth provided by the memory interface. XOR decompressor is fed with up to 512 compressed symbols per CB. XOR decompressor operates at 1 bit per cycle as the minimum granularity of a XOR compressed symbol is a single bit. As a consequence, the XOR decompressor requires 512 cycles before which a new CB can be pulled into the decompressor. This is due to dependency between consecutive symbols in CBs. Due to this sequential dependency, the processing of consecutive CBs cannot be pipelined (cannot decompress the next CB before first completing the previous one), limiting the processing throughput by 512-fold (to 1 CB/512 cycles). This is unlike base-delta encoding where the new bases are explicitly stored in the block format and the starting base for the successive memory line is already available at the start of decompressing a memory line. Moreover, for base-delta encoding, the decompressor stage performs a precompute step on the compressed block to find the bases, if any, in the compressed block based on the demarcations. This allows the decompression of all the deltas per CB in parallel and enables the BD decompressor to process one compressed block every cycle without any sequential dependency.

For a compressed single-window size of up to a single CB, line-rate processing throughput can also be achieved for XOR compression by using pipelined parallel decompressors as there are no sequential dependencies between CBs as shown in Figure 12. However, for larger

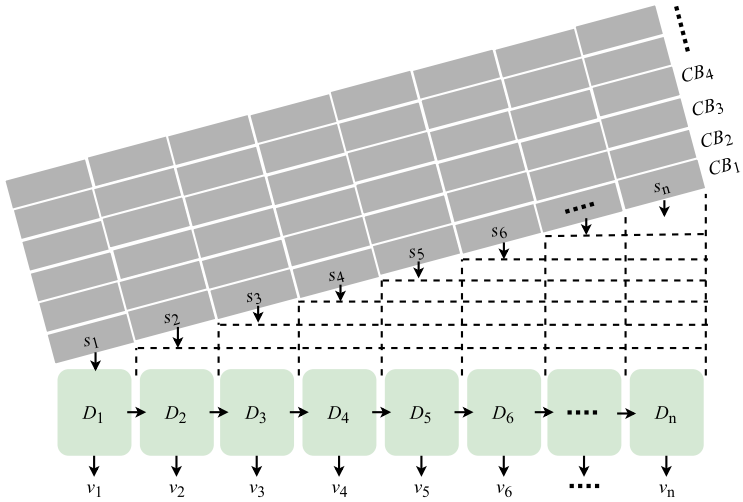


Fig. 12. Compressed blocks denoted by $CB_1 \ldots CB_n$ processed once every cycle when there are no dependencies between consecutive CBs. n denotes the number of compressed symbols in a block that are fed to the unrolled decompressor stages, $D_1, \ldots, D_n$. $s_1, \ldots, s_n$ denote the various symbols that are buffered and fed to the decompressor in consecutive cycles.

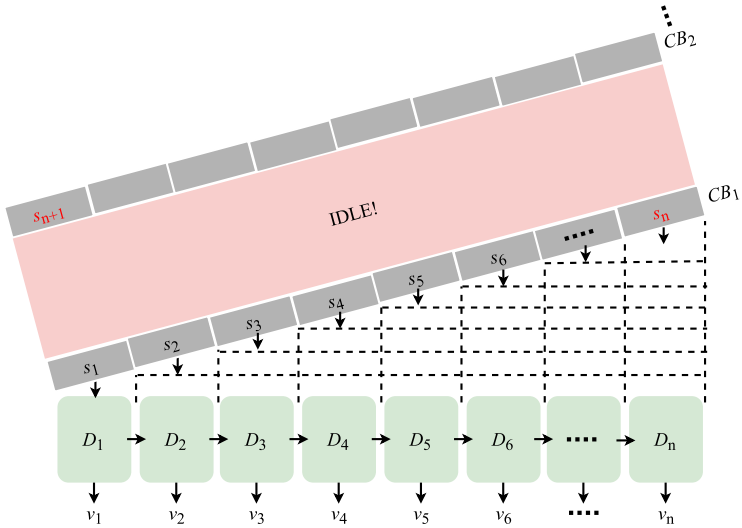
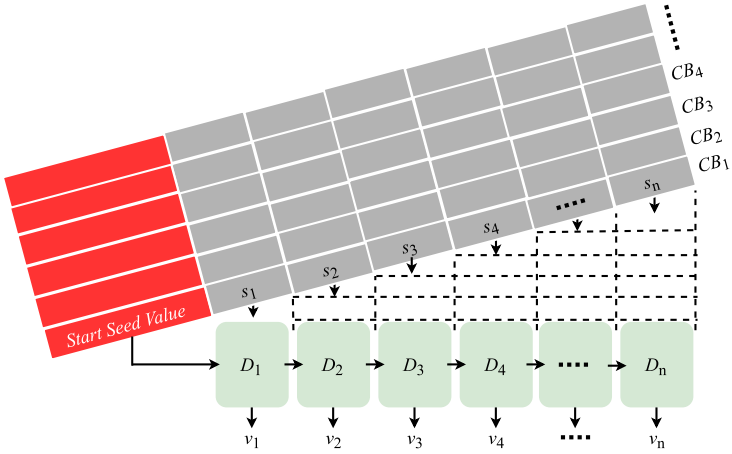


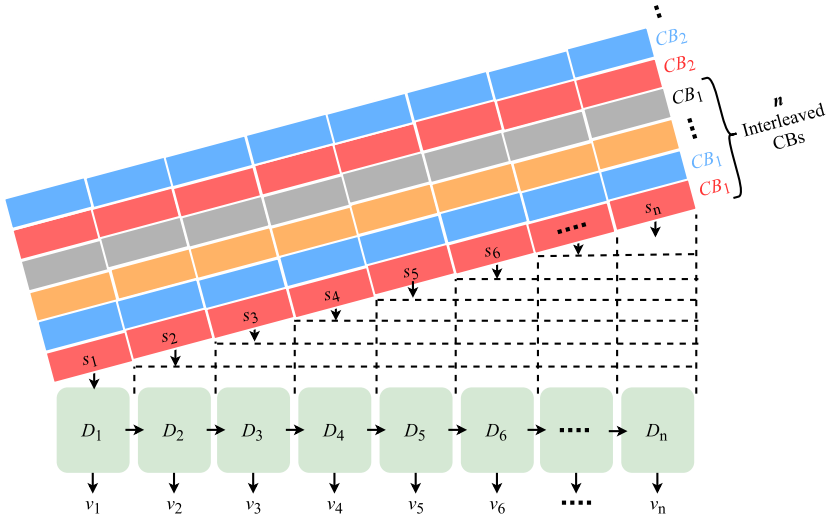
Fig. 13. Idling due to sequential dependency between compressed blocks costing processing throughput. Here CB_2 can be fed to the decompressor only after n cycles after processing CB_1 as the decompression of symbol s_{n+1} in CB_2 requires the decompressed value from the symbol s_n in CB_1. $s_1, \ldots, s_n$ denote the various symbols that are buffered and fed to the decompressor in consecutive cycles.

windows and keys triggering aggregation in close succession, the processing throughput is penalized due to the idle time between consecutive CBs as shown in Figure 13.

One way to solve this problem is by making the compressed blocks independent as shown in Figure 14(a) using the block format shown in Figure 9(b). However, making compressed



(a) Sequential Dependency Solution 1: *Independent compressed blocks*. Using the starting uncompressed seeds denoted in red, each compressed block denoted by $CB_1, CB_2, CB_3, CB_4, \ldots$ can be fed to the deompressor pipeline stages in consecutive cycles, enabling a processing throughput of 1 CB/cycle. $s_1, \ldots, s_n$ denote the various symbols which are buffered and fed to the decompressor in consecutive cycles.



(b) Sequential Dependency Solution 2: *Compressed block interleaving* across keys. Each coloured compressed block denoted by CB_* belongs to a separate key and these interleaved CBs across keys enables to increase the dependence distance to n so that on arrival of symbol s_{n+1} of a key, s_n from the same key would have been decompressed.

Fig. 14. Solutions for sequential dependency in compressed blocks.

blocks *independent* by storing the starting uncompressed seed per CB can lead to lower compression ratios.

Another approach pursued in StreamZip is to perform *compressed block interleaving* across keys so that each block entering the decompressor pipeline is from a different key in consecutive cycles as shown in Figure 14(b). This would require buffering per key after the decompressor to store the intermediate decompressed values for computation. However, for skewed key distributions, it would not be possible to interleave compressed blocks, which can be solved using caching of the decompressed values as discussed in Section 3.4.4.

3.4.2 Dependencies between Consecutive Aggregations. Stream aggregation queries with small WA may cause tuples of the same key to trigger aggregations in close succession or even in consecutive cycles. However, each aggregation causes the window to slide and some of its elements (values) to be evicted. Evictions are implemented during decompression by updating the hash table, but the latency between the hash table and decompressor is prohibitive for supporting successive evictions and therefore successive aggregations. StreamZip solves this problem by maintaining an *outstanding aggregations field,* which stores the count of the outstanding aggregations triggered per key in the hash table using a separate field as shown in Figure 7(a). The outstanding aggregations count for a key is reset to 0 on arrival of the first tuple and incremented by 1 on each aggregation trigger. Upon aggregation, the compressed single window is read as usual based on the available read and write pointers in the hash table stage, and the outstanding aggregation count is also passed to the decompressor. This count enables the decompressor to identify the unreported invalid compressed values in the window (count $\times$ WA) and to skip them. The count is decremented by 1 each time the hash table stage receives feedback from the decompressor for a completed aggregation. Although this works for tuple-based windowing policy, for time-based windows, storing just the count would not suffice. This is because the number of evicted tuples in a WA is time dependent and so a cumulative sum of the number of tuples in the WA time-unit per outstanding aggregation needs to be maintained and stored in the outstanding aggregation field.

3.4.3 Inter-compressor-decompressor Dependency. When an aggregation is triggered for a key, which has some tuples pending compression, the decompressor would need to stall until the compression of the pending tuples is completed. This would limit processing throughput especially for compression methods with high latency/deep pipelines (e.g., SZ). This is detrimental to the processing throughput of the pipeline, which in case of SZ can be 20-fold (SZ compressor pipeline depth) reduction if tuples of the same key trigger frequent aggregations (small WA). StreamZip addresses this by buffering the uncompressed values of each key under compression in the *Inter-buffer* until they are compressed, as shown in Figure 15. Then, on an aggregation trigger, the *Inter-buffer* is searched for any tuples of the key that triggered aggregation that are pending compression. If there are such tuples, their uncompressed values are forwarded to the compute kernel using a separate bypassing path as shown in Figure 15 and annotated as ❶, thereby preventing the decompressor from stalling. The inter-buffer is dequeued each time the compressor comes back with a completion response. Note that bypassing is not beneficial for low-latency (i.e., single-cycle) compression algorithms such as base-delta and XOR.

3.4.4 Caching Optimization for Skewed-key Distribution. For skewed-key distributions and especially for large WS and frequent aggregations, redundant memory reads for the single window of the same key waste valuable memory bandwidth and thereby become a performance bottleneck in StreamZip. In order to alleviate this bottleneck, the idea is to cache the decompressed window of a few recently seen keys that triggered aggregation at the decompressor DFE pipeline stage using on-chip memory as shown in Figure 15. Moreover, reusing the single window of a

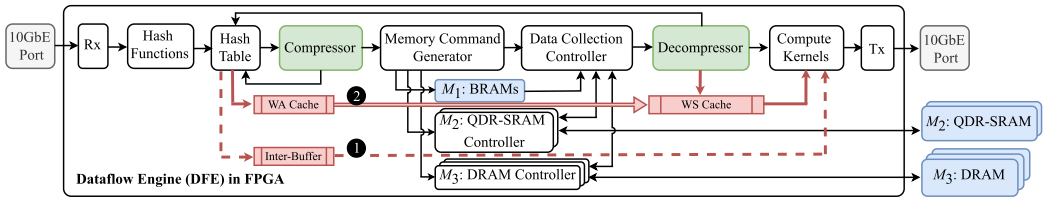


Fig. 15. StreamZip pipeline optimizations: (1) *Inter-buffer* to solve Inter-compressor-decompressor Dependency and (2) Caching Optimization for skewed-key distribution in StreamZip.

key in on-chip memory would prevent the need to go all the way to DRAM to fetch the window upon aggregation, thereby reducing DRAM pressure and enabling higher processing throughput for skewed-key distributions. The size of a cached block per key at the decompressor side is equal to WS, and the oldest tuples corresponding to WA will be invalidated upon window slide during aggregation. For tuple-based windows, the size is deterministic as the WS and WA are defined on the count of the tuples. However, for time-based windows, an upper bound on the WS is measured based on a per-key arrival rate and is used as the cache block size.

At the compressor side, the recently seen keys that triggered aggregation are marked and uncompressed values for those keys up to WA tuples are buffered. These cached uncompressed values of the key are streamed directly to the compute kernel upon the next aggregation trigger using a bypass path, annotated as ❷ in Figure 15 and used in conjunction with the cached decompressed values at the decompressor. The replacement policy of the caches follows an LRU policy based on the incoming keys similar to the cache structure in Time-SWAD [27]. In our implementation, we buffer the decompressed windows of up to two keys using on-chip memory, and the interface width of the cache at the decompressor side can feed the compute kernel up to 128 decompressed values per cycle.

3.4.5 Data Collection. The compressed window of each key is scattered across the memory levels. In addition, parts of the window may need to be taken from the buffer next to the compressor that stores uncompressed values for ongoing compressions, as explained in Section 3.4.3. To make matters worse, data may be moving from one memory level to the next at any point in time. Upon aggregation, StreamZip uses a data collection controller to ensure that all parts of the particular window instance are gathered correctly to be forwarded to the compute kernel. Separate queues are maintained for metadata and compressed data for each memory level and the uncompressed buffer. Then, the controller synchronizes the contents of the queues to deliver data in order to the decompressor. Value reordering is needed because the aggregation read operations are performed in parallel for all memory levels that contain valid data and correct value order is required for computing non-commutative aggregation functions like `rank`.

4 EVALUATION

The performance of StreamZip is evaluated in terms of processing throughput and latency. First, the experimental setup is discussed. Then, the implementation and performance results are presented and compared to existing approaches.

4.1 Experimental Setup

All designs are implemented on a Maxeler N-series ISCA (MAX4AB24B) PCIe card with Altera Stratix V (5SGXAB) that provides a 10 Gb/s direct network connection to the FPGA. The board offers 6 MB on-chip BRAMs, 72 MB off-chip QDR-SRAM, and 24 GB off-chip DDR3 DRAM. The

Table 2. Resource Utilization for the FPGA Implementations

Design/Resource	Logic (ALMs)	BRAMs	DSP
DFE-Base	86,208 (24%)	1,136 (43%)	0
MLQ	93,392 (26%)	1,347 (51%)	0
Streamzip-Lossless-Fixed	104,168 (29%)	1,584 (60%)	0
Streamzip-Lossless-Float	219,112 (61%)	1,796 (68%)	0
Streamzip-Lossy-Float	308,912 (86%)	2,139 (81%)	165 (47%)

designs are implemented in MaxJ, a Java-based **High Level Synthesis (HLS)** language, and compiled using MaxCompiler.

Three different types of FPGA-based single-window SWAG dataflow engines (DFEs) are implemented: first, a baseline SWAG engine that uses only DDR3 DRAM, denoted as DFE-Base, that follows the designs described in [9, 27]; second, a SWAG engine with a three-level MLQ memory system using BRAM, off-chip QDR-SRAM, and DRAM, denoted as MLQ, that follows the best-performing previous design [10]; and third, *StreamZip*, the compressed SWAG engine using lossless (base-delta encoding for fixed-point numbers and XOR compression for double-precision floating-point numbers) and lossy (SZ for double-precision floating-point numbers) compression schemes with the three-level MLQ memory system. These designs are denoted by StreamZip-Lossless-Fixed, StreamZip-Lossless-Float, and StreamZip-Lossy-Float, respectively, aptly capturing the principles of our approach.

The memory configuration of each design (partitioning per memory level) was generated using the partitioning algorithm in [10] with MLQ configured to store one and eight values in the M_1 and M_2 levels, respectively. StreamZip-Lossless-Fixed using the BD compression scheme is configured with 1 and up to 64 Δs, StreamZip-Lossless-Fixed using XOR compression is configured with 8 bytes and 64 bytes, and StreamZip-Lossy design is configured with 4 and up to 256 symbols per key in the M_1 and M_2 levels, respectively. M_3, being the last level, needs to have a capacity of a number of tuples equal to the WS per key.

The Google compute **cluster monitoring (CM)** [34] real-world dataset is used as the input streaming data. The tuples from the task events stream from the cluster usage traces are used for the experiments. For the standard group-by stream aggregation query that we consider in the article, we require only three fields for the timestamp, key, and value in the original tuple, namely, the timestamp (32-bit integer), job ID (32-bit integer), and resource request for CPU cores (64-bit floating point), respectively. The CPU resource usage field in the original dataset is originally in 64-bit floating-point format and is used as is for the Streamzip designs that work on floats. For designs that require the value field in the tuples to be integers such as Streamzip-Base-Delta, the floats were cast to integers. The *tcpreplay* tool [30] is used to inject the captured packets at varying injection rates to determine the highest sustainable system throughput.

The implemented query, comprising algebraic, distributive, and holistic aggregation functions, is the following: "Find the `average`, `minimum`, `maximum`, and `median` CPU usage for each job ID for the last WS tuples and return the aggregate every WA tuples" [18]. The WS ranges from 64 to 4K tuples; the WA varies from 1 to WS tuples with support for up to 16K concurrently active keys.

4.2 Implementation Results

The resource utilization of the evaluated designs is as shown in Table 2. For StreamZip-Lossless-Fixed, the increase in logic utilization over MLQ is attributed to the extra (de)compressor metadata

management and encoding. StreamZip-Lossless-Float requires a deeper decompressor pipeline due to the XOR compression scheme and hence a higher logic utilization. StreamZip-Lossless-Float also implements the independent compressed blocks to tackle sequential dependency. StreamZip-Lossy-Float has about 3× higher logic utilization due to the replicated parallel decompressor pipelines (up to 128) across keys and the associated state machine to control the dataflow per key. StreamZip-Lossy-Float relies on value interleaving to tackle the sequential dependency in the SZ compressed block.

In the case of StreamZip-Lossless-Fixed, an extra seed (base) for initiating the (de)compressor process is required per key in the hash table stage, which is attributed to the increase in BRAM utilization. StreamZip-Lossless-Float needs to store up to 8 bytes of compressed symbol(s) in BRAMs and consumes more BRAMs than StreamZip-Lossless-Fixed. StreamZip-Lossy needs to store three (de)compressor seeds per key in the hash table stage to aid in (de)compression and, as a result, has the highest resource utilization. DSP blocks are also utilized for the StreamZip-Lossy floating-point computations. In order to fit StreamZip floating-point designs in the FPGA, only one aggregation function (`min`) is implemented in the compute kernel stage and the reported resource utilization is based on this implementation.

All designs operate at 156.25 MHz supporting one incoming tuple (128 bits) every two FPGA cycles, fully utilizing the 10 Gb/s 64-bit network interface bandwidth. This translates to a theoretical line rate of 78.125 million tuples/sec. However, in practice, the highest measured rate of incoming tuples on the board is about 70 million tuples/sec, so about 90% of the theoretical. Finally, off-chip SRAM and DRAM are clocked at 350 MHz and 800 MHz, respectively.

4.3 Performance Results

The graphs in Figures 16(a), 16(b), 17(a), and 17(b) show the processing throughput in million tuples per second and latency in microseconds for WS ranging from 64 to 4K tuples for the various designs. A general observation is that the throughput of all designs in Figures 16(a), 16(b), 17(a), and 17(b) is reduced for small WA, especially for large WSs. This is in line with the trend observed in the previous work [9, 10, 27] because smaller WA queries trigger aggregations more frequently, and in addition, the larger windows aggregate more data. This leads to the problem becoming more memory bandwidth intensive. However, with StreamZip, this phenomenon is less pronounced as the number of compressed DRAM lines read upon aggregation is smaller compared to DFE-Base and MLQ by a factor of the overall compression ratio (CR, calculated as the ratio between the overall memory footprint of a design point without and with compression). This alleviates the DRAM bandwidth bottleneck to a large extent, leading to better processing throughput even for problems with extremely frequent aggregations (WA = 1). In case of the CM stream aggregation query, StreamZip-Lossless-Fixed (BD) and StreamZip-Lossless-Float (XOR) designs achieve, on average, a compression ratio of 5.3× and 5.6×, respectively. The StreamZip-Lossy (SZ) design achieves a compression ratio of 23×. In general, it is observed that the gains in processing throughput compared to designs with no compression are proportional to this ratio and the corresponding reduction in the number of DRAM lines read upon aggregation.

The latency follows an opposite trend as, the larger the WS and smaller the WA, the tuples will have to suffer longer queuing latency in the Data Collection Controller waiting for the outstanding aggregations to be completed before they can be processed. Compression helps to reduce the number of DRAM lines to be read for aggregation, and this helps to reduce the DRAM read traffic and hence the queuing latency suffered by the tuples in the pipeline. The achieved reduction is proportional to the CR especially for queries with frequent aggregations (WA < 16).

The DRAM-only design, DFE-Base, which follows the design principles of [9, 27], achieves the lowest throughput out of the three designs, supporting up to 15% of the line rate. This is primarily

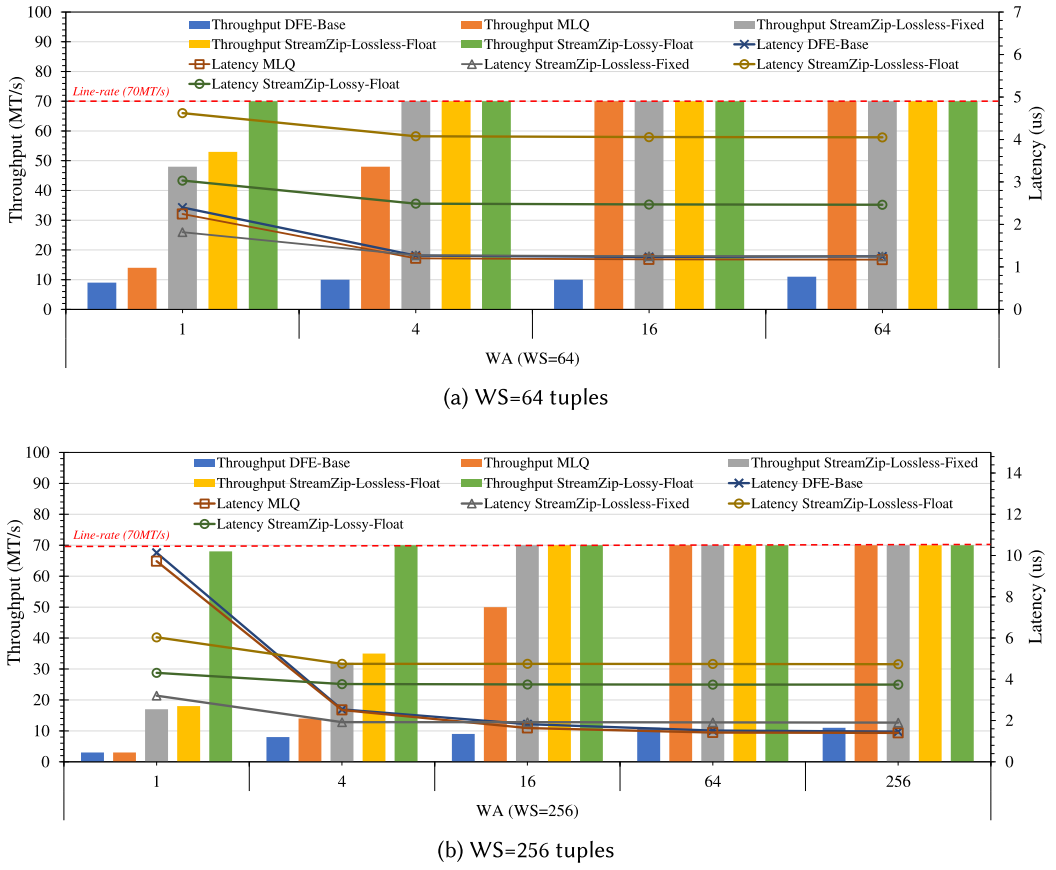


Fig. 16. Throughput in million tuples per second (MT/s) and latency in microseconds of the various designs for WS of (a) 64 and (b) 256 tuples.

because it handles the window update accesses inefficiently. More precisely, it requires slow and bandwidth-wasteful DRAM read-modify-writes, since an incoming tuple's value (8B) is smaller than the DRAM line (64B).

MLQ mitigates the above window-update problem by utilizing the multi-level queues that span across three memory levels in the platform to completely eliminate the wasteful read-modify-writes. Then, window-update writes of an incoming tuple happen always directly to the on-chip BRAM, which is configured to store one value per key and is flushed to the off-chip SRAM once every two tuples. As off-chip SRAM offers direct writes and the capacity required for storing an entire DRAM line of key-values before flushing to DRAM, MLQ completely eliminates the read-modify-writes. This allows to achieve up to 90% of the theoretical line-rate, which in practice matches the actual maximum rate of incoming tuples on the board. However, for small WA, it suffers a similar throughput reduction as DFE-Base. This is due to the following reasons. First, in these cases, despite MLQ's efficiency in handling window updates, the aggregation traffic is the bottleneck. Second, MLQ cannot take advantage of the aggregated bandwidth offered by BRAM and off-chip SRAM because, even for small WS, e.g., 64 as shown in Figure 16(a), the dominant portion of the window is stored in DRAM, which becomes the bottleneck.

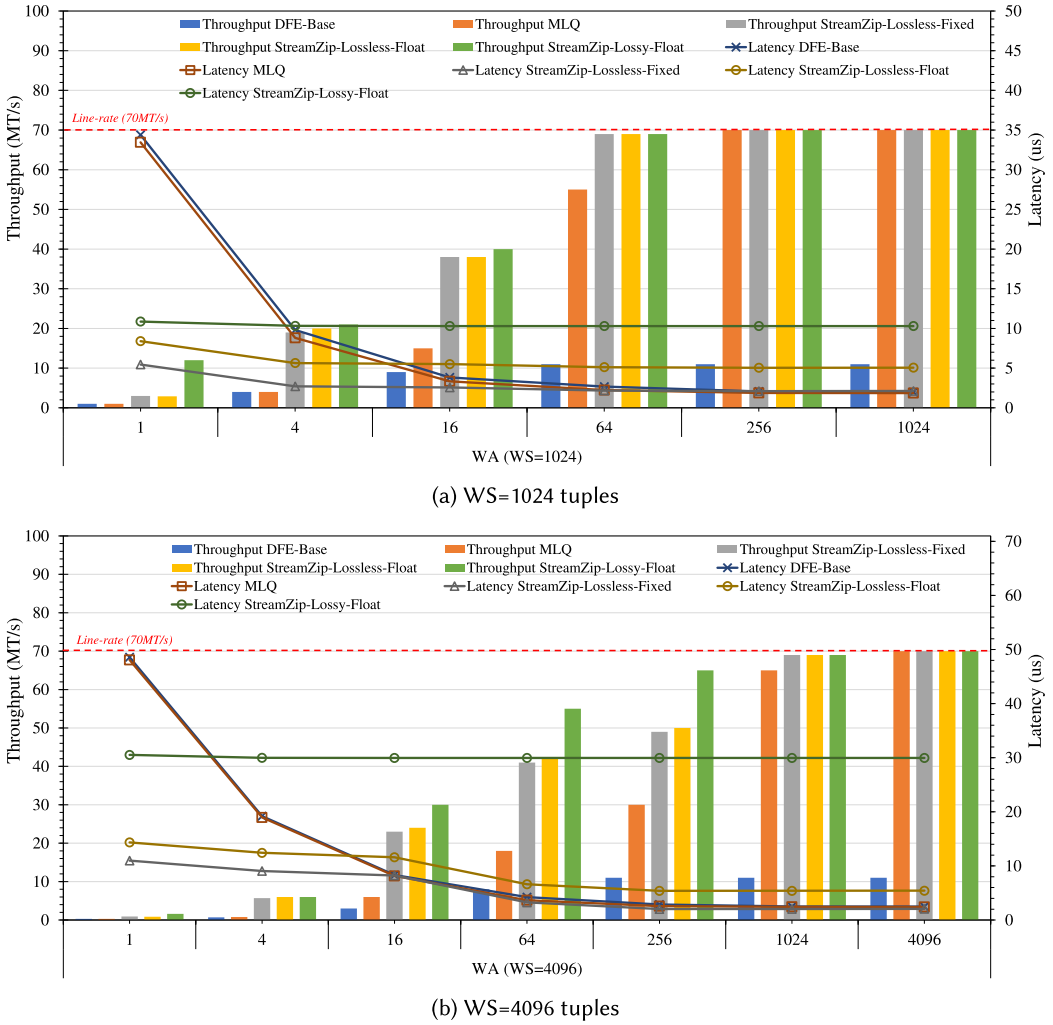


Fig. 17. Throughput in million tuples per second (MT/s) and latency in microseconds of the various designs for WS of (a) 1K and (b) 4K tuples.

StreamZip mitigates the above two problems to a large extent as the tuple's values are compressed on the fly during window updates, and this helps to improve the processing throughput and latency for smaller WA proportional to the compression ratio offered by the compression scheme. For instance, without compression, upon aggregation of a window of size 256, data of up to 32 DRAM lines need to be read and streamed to the compute kernel. With StreamZip-Lossless-Fixed (BD encoding), compressing the values to 1-byte Δs, the DRAM read traffic due to aggregation reduces by a factor of up to 5.7×, leading to proportional improvement in processing throughput as shown in Figure 16(b). StreamZip-Lossless-Float (XOR compression) achieves a compression ratio of 6× for this design point.

The StreamZip-Lossless-Float (XOR compression) design mitigates the performance bottleneck due to the sequential dependency between the CBs using *independent CBs* as discussed in Section 3.4.1. However, the tradeoff here is the reduction in compression ratio as a result of adding

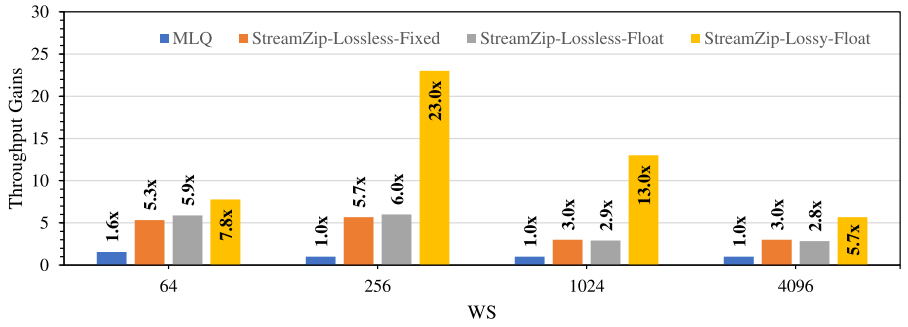


Fig. 18. Throughput gains for various designs normalized to DFE-Base design for WA = 1 tuple.

the extra starting seeds to the beginning of each CB versus the benefit of processing one CB every FPGA cycle. In our experiments, we see that this benefit outweighs the reduction in compression ratio as the decompressor can consume one CB every FPGA cycle, enabling to achieve better processing throughput, especially for larger compressed windows with mutliple CBs. As a consequence, StreamZip-Lossless-Float has the independent CB feature enabled by default and the numbers reported are for this design. Without using independent CBs, StreamZip-Lossless-Float is able to achieve a compression ratio of 6.5×, which gets reduced to 5.6× with the extra starting seeds per CB for mitigating the sequential dependency.

This is even further improved by utilizing lossless compression (SZ), which at best compresses a value to just 2 bits, offering better compressibility and hence higher throughput. Another interesting design point is WS = 64 in Figure 16(a), where StreamZip takes advantage of the aggregated bandwidth offered by BRAM and off-chip SRAM, because employing compression enables a large fraction of the window to remain in the first two memory levels, which in turn increases processing throughput up to line-rate. In our implementation, the output error for lossy SZ compression is configurable and set to 1%. The effective capacity gain is proportional to the compression ratio and StreamZip-lossless and StreamZip-lossy increase it by 5× and 23×, respectively.

From an end user's perspective of StreamZip, the idea here is to have more knobs in the user's arsenal to enable efficient use of various compression methods that may have different characteristics, while offering high-performance stream aggregation.

4.3.1 Overall Throughput Gains. Adding compression to the MLQ design improves the performance of extremely frequent aggregations with small WA by reduction of data volume being read upon aggregation. Figure 18 shows the overall throughput gains achieved by the various compression schemes for extremely frequent aggregations with WA = 1. The gains are normalized to the throughput achieved by the DFE-Base baseline design. Compared to DFE-Base and MLQ system, StreamZip-Lossless-Fixed, StreamZip-Lossless-Float, and StreamZip-Lossy-Float achieve processing throughput gains of up to 5.7×, 6×, and 23×, respectively.

4.3.2 Caching Results for Skewed Key Distribution. Figure 19 shows the effect of adding the stream cache to buffer the decompressed values of the most recently seen keys for varying WS. In order to stress the system, the experiment uses a synthetic dataset with a single key triggering extremely frequent aggregations with WA = 1, which is the worst-case scenario for a system with no caching support. This causes each tuple entering StreamZip to trigger aggregation. In our current implementation, the cache stores the window of only a single key because for systems with large window sizes the FPGA resources were not enough to implement larger caches. In general, StreamZip with cache support would be able to offer the reported performance also for N keys

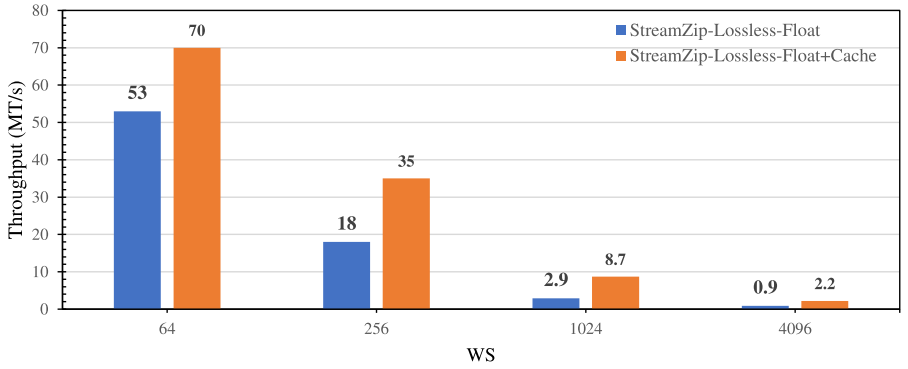


Fig. 19. Throughput of StreamZip-Lossless-Float with Caching. A synthetic benchmark is used with a single key triggering aggregation in every cycle with WA = 1 tuple.

provided the cache is able to store windows of N keys. This is an expected implementation/resource limitation, which in our view does not reduce the value of the proposed caching mechanism in general.

Compared to StreamZip-Lossless-Float, StreamZip-Lossless-Float+Cache achieves up to 3× better processing throughput. In this implementation, the cache output interface width is 128 values and the compute kernel is fed up to 128 values every cycle. So, for window sizes greater than 128, we see a drop in line-rate processing throughput. Should the cache interface be bigger, the DFE would be able to achieve higher processing rates.

4.3.3 Comparison with Related Work. Overall, StreamZip offers higher performance by alleviating the memory bandwidth bottleneck of window aggregations using compression, especially for smaller WA. In addition, there is a reduction on the memory footprint, offering higher effective memory capacity available for solving larger stream aggregation problems. Compared to the best-performing previous work, MLQ [10], StreamZip-Lossless-Fixed, StreamZip-Lossless-Float, and StreamZip-Lossy offer up to 7×, 7.5×, and 22× better processing throughput, respectively. In terms of latency, for larger WS and smaller WA (<4), StreamZip-Lossless-Fixed, StreamZip-Lossless-Fixed, and StreamZip-Lossy-Float offer up to 6×, 4×, and 3× lower latency than MLQ. However, for larger WS and WA, StreamZip-Lossless-Float and StreamZip-Lossy-Float have up to 12× higher latency. This is due to the deep pipeline required during aggregation for decompressing a large number of values packed per read DRAM line. Nevertheless, it is still orders of magnitude better than CPU and GPU systems. We also tested with uniformly random values (zero compressibility), and the processing throughput of Streamzip for this worst case is similar to the MLQ system with a slight overhead of up to 5% on average. This overhead is mainly attributed to the increase in data footprint due to the extra bits required for compression-packing.

It is worth noting that, similar to MLQ, StreamZip matches GPU processing throughput and offers substantially better latency. Gasser is the fastest GPU system in literature that supports non-associative functions and therefore follows a non-incremental aggregation approach [3]. However, it supports queries with only a single key, as opposed to 16K keys supported by our StreamZip implementation. As a consequence, Gasser handles only small problem sizes and also does not capture tuples from the network. We experimented with a single key query and StreamZip was able to achieve similar throughput (up to line-rate) to Gasser for varying WS/WA. However, StreamZip achieves this at a much lower latency, which is three to four orders of magnitude lower than Gasser.

5 CONCLUSION

This article extended StreamZip, a dataflow stream aggregation engine that is able to compress the sliding windows, alleviates the memory pressure posed by window aggregation traffic, and improves performance as well as effective memory capacity. StreamZip addresses a number of concurrency control challenges to integrate a compressor in the stream aggregation pipeline. StreamZip supports both lossy and lossless compression algorithms with diverse characteristics, applied to both fixed- and floatingpoint numbers. Compared to designs without compression, StreamZip lossless and lossy designs achieve up to 7.5× and 22× higher throughput, respectively, while reducing effective memory capacity by up to 5× and 23×, respectively.

REFERENCES

[1] Henrique C. M. Andrade, Buğra Gedik, and Deepak S. Turaga. 2014. *Fundamentals of Stream Processing: Application Design, Systems, and Analytics*. Cambridge University Press.

[2] Grigorios Chrysos, Odysseas Papapetrou, Dionisios Pnevmatikatos, Apostolos Dollas, and Minos Garofalakis. 2019. Data stream statistics over sliding windows: How to summarize 150 million updates per second on a single node. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL'19)*. 278–285. https://doi.org/10.1109/FPL.2019.00052

[3] T. De Matteis, G. Mencagli, D. De Sensi, M. Torquati, and M. Danelutto. 2019. GASSER: An auto-tunable system for general sliding-window streaming operators on GPUs. In *IEEE Access*, vol. 7 (2019), 48753–48769.

[4] Sheng Di and Franck Cappello. 2016. Fast error-bounded lossy HPC data compression with SZ. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS'16)*. IEEE, 730–739.

[5] Albin Eldstål-Ahrens and Ioannis Sourdis. 2020. MemSZ: Squeezing memory traffic with lossy compression. *ACM Transactions on Architecture and Code Optimization (TACO)* 17, 4 (2020), 1–25.

[6] Matthew Farrens and Arvin Park. 1991. Dynamic base register caching: A technique for reducing address bus width. In *Proceedings of the 18th Annual International Symposium on Computer Architecture (ISCA'91)*. 128–137.

[7] Apache Flink. 2022. https://flink.apache.org/.

[8] John Gantz and David Reinsel. 2012. The Digital Universe in 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East. (December 2012).

[9] Prajith Ramakrishnan Geethakumari, Vincenzo Gulisano, Bo Joel Svensson, Pedro Trancoso, and Ioannis Sourdis. 2017. Single window stream aggregation using reconfigurable hardware. In *2017 International Conference on Field Programmable Technology (ICFPT'17)*. 112–119. https://doi.org/10.1109/FPT.2017.8280128

[10] Prajith Ramakrishnan Geethakumari and Ioannis Sourdis. 2021. A specialized memory hierarchy for stream aggregation. In *2021 31st International Conference on Field-programmable Logic and Applications (FPL'21)*. 204–210. https://doi.org/10.1109/FPL53798.2021.00041

[11] Prajith Ramakrishnan Geethakumari and Ioannis Sourdis. 2021. StreamZip: Compressed sliding-windows for stream aggregation. In *2021 International Conference on Field-programmable Technology (ICFPT'21)*. 203–211. https://doi.org/10.1109/ICFPT52863.2021.9609952

[12] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. 1997. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery* 1, 1 (1997), 29–53.

[13] Vincenzo Gulisano, Zbigniew Jerzak, Roman Katerinenko, Martin Strohbach, and Holger Ziekow. 2017. The DEBS 2017 grand challenge. In *ACM International Conference on Distributed and Event-based Systems (DEBS'17)*. ACM, 271–273.

[14] Vincenzo Gulisano, Zbigniew Jerzak, Spyros Voulgaris, and Holger Ziekow. 2016. The DEBS 2016 grand challenge. In *ACM International Conference on Distributed and Event-based Systems (DEBS'16)*. ACM, 289–292.

[15] Vincenzo Gulisano, Ricardo Jiménez-Peris, Marta Patiño-Martínez, Claudio Soriente, and Patrick Valduriez. 2012. StreamCloud: An elastic and scalable data streaming system. *IEEE Transactions on Parallel and Distributed Systems* 23, 12 (2012), 2351–2365. https://doi.org/10.1109/TPDS.2012.24

[16] Vincenzo Gulisano, Yiannis Nikolakopoulos, Ivan Walulya, Marina Papatriantafilou, and Philippas Tsigas. 2015. Deterministic real-time analytics of geospatial data streams through scalegate objects. In *Proceedings of the 9th ACM International Conference on Distributed Event-based Systems (DEBS'15)*. ACM, 316–317.

[17] A. Kirsch, M. Mitzenmacher, and G. Varghese. 2010. Hash-based techniques for high-speed packet processing. In *Algorithms for Next Generation Networks*.

[18] Alexandros Koliousis, Matthias Weidlich, Raul Castro Fernandez, Alexander L. Wolf, Paolo Costa, and Peter Pietzuch. 2016. SABER: Window-based hybrid stream processing for heterogeneous architectures. In *Int. Conf. on Manag. of Data*.

[19] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. 2005. No pane, no gain: Efficient evaluation of sliding-window aggregates over data streams. *ACM SIGMOD Rec.* 34, 1 (2005), 39–44.
[20] Panagiotis Liakos, Katia Papakonstantinopoulou, and Yannis Kotidis. 2022. Chimp: Efficient lossless floating point compression for time series databases. *Proceedings of the VLDB Endowment* 15, 11 (2022), 3058–3070.
[21] Hongyu Miao, Myeongjae Jeon, Gennady Pekhimenko, Kathryn S. McKinley, and Felix Xiaozhu Lin. 2019. Streambox-hbm: Stream analytics on high bandwidth hybrid memory. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*. 167–181.
[22] Rene Mueller, Jens Teubner, and Gustavo Alonso. 2009. Streams on wires: A query compiler for FPGAs. *VLDB* 2, 1 (2009), 229–240.
[23] Mohammadreza Najafi, Kaiwen Zhang, Mohammad Sadoghi, and Hans-Arno Jacobsen. 2017. Hardware acceleration landscape for distributed real-time analytics: Virtues and limitations. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS'17)*. IEEE, 1938–1948.
[24] Yasin Oge, Masato Yoshimi, Takefumi Miyoshi, Hideyuki Kawashima, Hidetsugu Irie, and Tsutomu Yoshinaga. 2013. An efficient and scalable implementation of sliding-window aggregate operator on FPGA. In *International Symposium on Computing and Networking (CANDAR'13)*. IEEE, 112–121.
[25] Gennady Pekhimenko, Chuanxiong Guo, Myeongjae Jeon, Peng Huang, and Lidong Zhou. 2018. Tersecades: Efficient data compression in stream processing. In *2018 USENIX Annual Technical Conference (USENIX ATC'18)*. 307–320.
[26] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. 2015. Gorilla: A fast, scalable, in-memory time series database. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1816–1827.
[27] Prajith Ramakrishnan Geethakumari, Vincenzo Gulisano, Pedro Trancoso, and Ioannis Sourdis. 2019. Time-SWAD: A dataflow engine for time-based single window stream aggregation. In *2019 International Conference on Field-programmable Technology (ICFPT'19)*. 72–80. https://doi.org/10.1109/ICFPT47387.2019.00017
[28] Paruj Ratanaworabhan, Jian Ke, and Martin Burtscher. 2006. Fast lossless compression of scientific floating-point data. In *Data Compression Conference (DCC'06)*. IEEE, 133–142.
[29] Kanat Tangwongsan, Martin Hirzel, and Scott Schneider. 2017. Low-latency sliding-window aggregation in worst-case constant time. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems (DEBS'17)*. ACM, 66–77.
[30] Tcpreplay. (n. d.). http://tcpreplay.appneta.com/.
[31] Walter F. Tichy. 1985. RCS–A system for version control. *Software: Practice and Experience* 15, 7 (1985), 637–654.
[32] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. 2014. Storm@ Twitter. In *ACM SIGMOD Int. Conf. on Management of Data*. ACM, 147–156.
[33] Alvaro Villalba, Josep Lluís Berral, and David Carrera. 2019. Constant-time sliding window framework with reduced memory footprint and efficient bulk evictions. *IEEE Transactions on Parallel and Distributed Systems* 30, 3 (2019), 486–500.
[34] John Wilkes and Charles Reiss. 2011. Google Cluster Data. https://github.com/google/cluster-data/blob/master/ClusterData2011_2.md.
[35] Qingqing Xiong, Rushi Patel, Chen Yang, Tong Geng, Anthony Skjellum, and Martin C. Herbordt. 2019. Ghostsz: A transparent FPGA-accelerated lossy compression framework. In *2019 IEEE 27th Annual International Symposium on Field-programmable Custom Computing Machines (FCCM'19)*. IEEE, 258–266.
[36] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized streams: Fault-tolerant streaming computation at scale. In *ACM Symposium on Operating Systems Principles*. 423–438.
[37] Feng Zhang, Lin Yang, Shuhao Zhang, Bingsheng He, Wei Lu, and Xiaoyong Du. 2020. FineStream: Fine-grained window-based stream processing on CPU-GPU integrated architectures. In *2020 USENIX Annual Technical Conference (USENIX ATC'20)*. 633–647.

Received 7 February 2022; revised 10 January 2023; accepted 17 March 2023