THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

Reconfigurable-Hardware Accelerated Stream Aggregation

PRAJITH RAMAKRISHNAN GEETHAKUMARI

Division of Computer and Network Systems
Department of Computer Science & Engineering
Chalmers University of Technology
Gothenburg, Sweden, 2022

Reconfigurable-Hardware Accelerated Stream Aggregation

PRAJITH RAMAKRISHNAN GEETHAKUMARI

Advisor: Ioannis Sourdis, Professor at Chalmers University of Technology

Examiner: Ulf Assarsson, Professor at Chalmers University of Technology

Thesis Opponent:
Dirk Koch, Senior Lecturer at University of Manchester, UK

Grading Committee:
Walid A. Najjar, Professor at University of California Riverside, USA
Apostolos Dollas, Professor at Technical University of Crete in Chania, Greece
Luciano Lavagno, Professor at Politecnico di Torino, Italy
Lars Svensson, Senior Lecturer at Chalmers Univ. of Technology (deputy member)

Copyright ©2022 Prajith Ramakrishnan Geethakumari
except where otherwise stated.
All rights reserved.

ISBN 978-91-7905-610-0
Doktorsavhandlingar vid Chalmers tekniska högskola
Series Number: 5076
ISSN: 0346-718X
Technical Report Number: 208D

Department of Computer Science & Engineering
Division of Computer and Network Systems
Chalmers University of Technology
Gothenburg, Sweden

This thesis has been prepared using LaTeX.
Printed by Chalmers Reproservice,
Gothenburg, Sweden 2022.

Abstract

High throughput and low latency stream aggregation is essential for many applications that analyze massive volumes of data in real-time. Incoming data need to be stored in a *single sliding-window* before processing, in cases where incremental aggregations are wasteful or not possible at all. However, storing all incoming values in a single-window puts tremendous pressure on the memory bandwidth and capacity. GPU and CPU memory management is inefficient for this task as it introduces unnecessary data movement that wastes bandwidth. FPGAs can make more efficient use of their memory but existing approaches employ only on-chip memory and therefore, can only support small problem sizes (i.e. small sliding windows and number of keys) due to the limited capacity. This thesis addresses the above limitations of stream processing systems by proposing techniques for accelerating single sliding-window stream aggregation using FPGAs to achieve line-rate processing throughput and ultra low latency. It does so first by building accelerators using FPGAs and second, by alleviating the memory pressure posed by single-window stream aggregation. The initial part of this thesis presents the accelerators for both windowing policies, namely, *tuple-* and *time-* based, using Maxeler's *DataFlow Engines* (DFEs) which have a direct feed of incoming data from the network as well as direct access to off-chip DRAM. Compared to state-of-the-art stream processing software system, the DFEs offer 1-2 orders of magnitude higher processing throughput and 4 orders of magnitude lower latency. The later part of this thesis focuses on alleviating the memory pressure due to the various steps in single-window stream aggregation. Updating the window with new incoming values and reading it to feed the aggregation functions are the two primary steps in stream aggregation. The high on-chip SRAM bandwidth enables line-rate processing, but only for small problem sizes due to the limited capacity. The larger off-chip DRAM size supports larger problems, but falls short on performance due to lower bandwidth. In order to bridge this gap, this thesis introduces a specialized memory hierarchy for stream aggregation. It employs *Multi-Level Queues* (MLQs) spanning across multiple memory levels with different characteristics to offer both high bandwidth and capacity. In doing so, larger stream aggregation problems can be supported at line-rate performance, outperforming existing competing solutions. Compared to designs with only on-chip memory, our approach supports 4 orders of magnitude larger problems. Compared to designs that use only DRAM, our design achieves up to 8× higher throughput. Finally, this thesis aims to alleviate the memory pressure due to the window-aggregation step. Although window-updates can be supported efficiently using MLQs, frequent window-aggregations remain a performance bottleneck. This thesis addresses this problem by introducing *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 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 fixed- as well as floating- point numbers. Compared to designs using MLQs, 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.

Keywords

Stream, Aggregation, Dataflow, FPGA, Memory Hierarchy, Compression

Acknowledgment

I would like to extend my deepest gratitude to my advisor, Yiannis, for being this pillar of support guiding me through the ups and downs during the course of my studies. I would like to thank Pedro, Vincenzo, and Joel for their valuable insights and suggestions for my research. Also a big thanks to all the nice people at Chalmers for creating a great environment to work in.

I would also like to thank Maxeler Technologies for providing the infrastructure for running the experiments. This work was supported by the Swedish Research Council under the ScalaNetS project (2016-05231) and the European Commission under the Horizon 2020 Program through the COSSIM (644042) project.

My family was always there to support and take care of me. They stood by me at all times and their prayers and blessings have guided me all my life. Thank you.

List of Publications

This thesis is based on the following publications:

[A] **Prajith Ramakrishnan Geethakumari**, Vincenzo Gulisano, Bo Joel Svensson, Pedro Trancoso and Ioannis Sourdis
"Single Window Stream Aggregation using Reconfigurable Hardware"
International Conference on Field Programmable Technology (ICFPT) 2017, Melbourne, Australia, Dec. 11-13, 2017, pp. 112-119.

[B] **Prajith Ramakrishnan Geethakumari**, Vincenzo Gulisano, Pedro Trancoso and Ioannis Sourdis
"Time-SWAD: A Dataflow Engine for Time-based Single Window Stream Aggregation"
International Conference on Field Programmable Technology (ICFPT) 2019, Tianjin, China, Dec. 9-13, 2019, pp. 72-80.

[C] **Prajith Ramakrishnan Geethakumari** and Ioannis Sourdis
"A Specialized Memory Hierarchy for Stream Aggregation"
International Conference on Field-Programmable Logic and Applications (FPL) 2021, Dresden, Germany, Aug. 30 - Sept. 3, 2021, pp. 204-210.

[D] **Prajith Ramakrishnan Geethakumari** and Ioannis Sourdis
"StreamZip: Compressed Sliding-Windows for Stream Aggregation"
International Conference on Field Programmable Technology (ICFPT) 2021, Auckland, New Zealand, Dec. 6-10, 2021, pp. 203-211.

[E] **Prajith Ramakrishnan Geethakumari** and Ioannis Sourdis
"Stream Aggregation with Compressed Sliding-Windows"
Under submission as journal extension to [D].

Other publications

The following publications were also published during my PhD studies. However, they are not appended to this thesis because their contents are not related to the thesis.

[a] Yang Ma, **Prajith Ramakrishnan Geethakumari**, Georgios Smaragdos, Sander Lindeman, Vincenzo Romano, Mario Negrello, Ioannis Sourdis, Laurens W.J. Bosman, Chris I. De Zeeuw, Zaid Al-Ars and Christos Strydis
"Towards real-time whisker tracking in rodents for studying sensorimotor disorders"
International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS) 2017, Pythagorion, Greece, July 17-20, 2017, pp. 137-145.

[b] Andreas Brokalakis, Nikolaos Tampouratzis, Antonios Nikitakis, Ioannis Papaefstathiou, Stamatis Andrianakis, Danilo Pau, Emanuele Plebani, Marco Paracchini, Marco Marcon, Ioannis Sourdis, **Prajith Ramakrishnan Geethakumari**, Maria Carmen Palacios, Miguel Angel Anton and Attila Szasz
"COSSIM: An Open-Source Integrated Solution to Address the Simulator Gap for Systems of Systems"
Euromicro Conference on Digital System Design (DSD) 2018, Prague, Czech Republic, Aug. 29-31, 2018, pp. 115-120.

Contents

Abstract iii

Acknowledgement v

List of Publications vii

1 Introduction 1
- 1.1 Problem Statement . . . 2
- 1.2 Thesis Objectives and Contributions . . . 4
  - 1.2.1 High Throughput & Low Latency Single-SWAG Acceleration 4
  - 1.2.2 Alleviating Memory Pressure due to Frequent Updates . . . 6
  - 1.2.3 Alleviating Memory Pressure due to Frequent Aggregations . 8
- 1.3 Thesis Outline . . . 9

2 Single Window Stream Aggregation using Reconfigurable Hardware 11
- 2.1 Background - Stream Aggregation . . . 12
- 2.2 Related Work . . . 15
- 2.3 A Data-Flow Engine for Single Window Stream Aggregation . . . 16
  - 2.3.1 Receiver and Transmitter . . . 18
  - 2.3.2 Hash Function . . . 18
  - 2.3.3 Hash Table . . . 18
  - 2.3.4 Concurrency Control Queues . . . 19
  - 2.3.5 DRAM access . . . 19
    - 2.3.5.1 Placement of key values in DRAM . . . 19
    - 2.3.5.2 DRAM Read . . . 20
    - 2.3.5.3 DRAM Write . . . 20
  - 2.3.6 Compute Stage . . . 21
- 2.4 Evaluation . . . 21
  - 2.4.1 Experimental Setup . . . 21
  - 2.4.2 Resource Utilization and Maximum Frequency . . . 22
  - 2.4.3 Throughput . . . 22
  - 2.4.4 Latency . . . 27
  - 2.4.5 Power . . . 28
- 2.5 Conclusion . . . 28

3 Time-SWAD: A Dataflow Engine for Time-based Single Window Stream Aggregation 31
3.1 Background - Time-based SWAG . . . 33
3.2 Related Work . . . 34
3.3 Time-SWAD Design . . . 36
3.3.1 Flexible and expandable circular buffers . . . 38
3.3.2 Hash Table . . . 40
3.3.3 Memory subsystem . . . 41
3.3.3.1 DRAM Single-Line (SL) Read-Modify-Write . . . 42
3.3.3.2 DRAM Multiple-Line (ML) Read . . . 43
3.4 Evaluation . . . 43
3.4.1 Experimental Setup . . . 43
3.4.2 Resource Utilization and DFE Frequency . . . 44
3.4.3 Throughput and Latency . . . 44
3.4.4 Comparison with alternative systems . . . 46
3.5 Conclusion . . . 47

4 A Specialized Memory Hierarchy for Stream Aggregation 49
4.1 MLQ Analytical Model . . . 51
4.1.1 Window Updates . . . 52
4.1.2 Aggregation . . . 53
4.1.3 Average Throughput . . . 53
4.1.4 Automatic memory configuration generation . . . 53
4.2 Reconfigurable Single Window SWAG with MLQ . . . 55
4.2.1 Hash Table . . . 55
4.2.2 Memory Command Generator . . . 58
4.2.3 Data Collection Controller . . . 58
4.2.4 Computer Kernels . . . 59
4.3 Evaluation . . . 59
4.3.1 Experimental Setup . . . 59
4.3.2 Implementation Results . . . 60
4.3.3 Processing throughput and latency . . . 61
4.3.4 Accuracy of the analytical performance model . . . 63
4.3.5 Comparison with related work . . . 63
4.4 Conclusion . . . 65

5 StreamZip: Compressed Sliding-Windows for Stream Aggregation 67
5.1 Background and Related Work . . . 69
5.1.1 Compression algorithms . . . 69
5.1.2 Stream processing platforms . . . 72
5.2 StreamZip Design . . . 73
5.2.1 Hash table organisation . . . 75
5.2.1.1 Window management bank . . . 75
5.2.1.2 Compressor bank . . . 75
5.2.1.3 Decompressor bank . . . 76
5.2.2 Memory alignment and packing . . . 76
5.2.3 Data dependencies and Concurrency control mechanisms . . . 78
5.2.3.1 Intra-(De)Compressor Pipeline Dependency . . . 79
5.2.3.2 Inter-Compressor-Decompressor Dependency . . . 81

5.2.3.3 Dependencies between Consecutive Aggregations . 83
5.2.3.4 Data Collection 83
5.2.3.5 Caching Optimization for skewed-key distribution . 83
5.2.4 Memory Management and Dataflow 85
5.3 Evaluation 87
5.3.1 Experimental Setup 87
5.3.2 Implementation Results 88
5.3.3 Performance results 89
5.3.3.1 Overall throughput gains 92
5.3.3.2 Caching results for skewed key distribution 92
5.3.3.3 Comparison with related work 93
5.4 Conclusion 93

6 Conclusions **95**
6.1 Summary 95
6.2 Contributions 96
6.3 Future Work 97

Bibliography **99**

Chapter 1

Introduction

The number of connected devices grows rapidly along with the amount of data they produce and exchange. Processing such *big data* brings tremendous opportunities in various domains (e.g. financial, transportation) enabling real-time sophisticated decisions that were never possible before. However, analysing unbounded streams is challenging, it requires *high processing throughput* to cope with massive volumes of data and if real-time processing is expected, calls for *low latency* to respond fast.

Stream aggregation is one of the fundamental and computationally challenging types of stream processing [1]. Streams of values (tuples) are handled in windows of a particular size, WS, which are "slid" by a particular window advance, WA. Then, an aggregation output is produced per window, computed based on a *function* that uses as input the window values. The result of a Sliding-Window AGgregation (SWAG) is a stream of aggregated values [2]. Often values in a stream are *grouped-by a key*, then, values of different keys are processed separately. SWAG may follow a *tuple-based* windowing policy, meaning WS and WA are measured in terms of the count of elements. They are suitable for applications with fixed data rates and have a fixed memory footprint. An example of tuple-based SWAG is depicted in Figure 1.1. An alternative windowing policy is the *time-based*, where the WS and WA are defined by time intervals. Time-based SWAG allows varying data-arrival rates which naturally fits the time-series data produced by most Internet-of-Things (IoT) devices [3–5]. Nevertheless, the number of tuples contained in a time-based window can vary making the memory and compute resources needed to produce the aggregation result unpredictable.

Some aggregation functions can be computed incrementally using either multiple windows [6] or panes [7, 8]. Incremental aggregation computes and stores partial results, rather than storing all the incoming values of a window before computing the full function. In doing so, usually memory pressure is reduced (both bandwidth and capacity) and performance is improved [9]. However, for some queries, especially with small WA and multiple aggregation functions, incremental aggregation has the opposite effect causing an excessive number of memory accesses that limit performance [9], e.g., cases of processing geo-tagged data [5], social media data [10] or manufacturing equipment data [11]. Then, *Single* Sliding-Window AGgregation (Single-SWAG) which is a non-incremental approach that explicitly stores all the incoming values in a single-window before processing is more efficient [9]. More importantly, storing values in a single-window is unavoidable when computing *holistic functions*; these

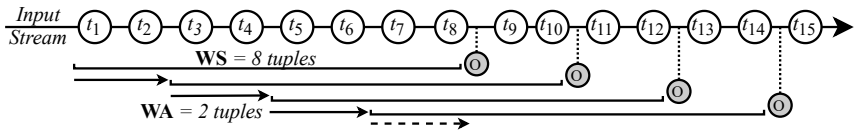


Figure 1.1: Sliding-window stream aggregation with Window Size (WS) = 8 tuples and Window Advance (WA) = 2 tuples for a input data stream, e.g. a vehicular sensor emitting tuples $t_1, t_2, \ldots$ with each tuple containing a timestamp, vehicle-ID and speed. For simplicity, only tuples from a single 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).*

are functions that have no constant bound on the size required to store a partial result, such as `median` [12].

In general, non-incremental, Single-SWAG is more suitable for general data mining and machine learning functions such as classification (e.g., decision trees, random forest, support vector machines, KNN), as well as for most of the inductive machine learning algorithms [4]. However, *temporarily storing all incoming values during processing puts tremendous pressure to the memory*, which often becomes the bottleneck. For each incoming tuple, the single sliding-window of the respective key is updated with the newly arrived value(s). In addition, every time the window advances, its entire contents need to be read and fed to the aggregation function(s) to produce a result. These two Single-SWAG steps, *window-update* and *window-aggregation*, generate tremendous memory pressure, the latter especially for queries with frequent aggregations (small WA).

This thesis addresses the overheads and limitations of stream processing systems by proposing techniques for accelerating single-window stream aggregation using FPGAs, thereby supporting holistic aggregation functions with high throughput and low latency.

The rest of this introductory Chapter is organized as follows: The problem statement is presented in Section 1.1 followed by a discussion of the objectives and contributions of this thesis in Section 1.2.

1.1 Problem Statement

Single Sliding-Window AGgregation (Single-SWAG) supporting holistic aggregation functions is *data-intensive* and becomes a major performance bottleneck for stream processing systems. Below follows the list of inefficiencies observed in existing literature, and tackled in this thesis.

Unnecessary data movement costs performance: Stream processing and stream aggregation systems in particular, have been implemented on various compute platforms. Multicore CPU and GPU systems are able to sustain high processing throughput but fall short in delivering low latency [13]. They require redundant memory accesses managed by an operating system to store incoming tuples in DRAM even before processing starts. This, besides the long latency, wastes a significant fraction of valuable memory bandwidth reducing performance. This performance gap provides an opportunity to consider FPGAs to implement customized dataflow engines which

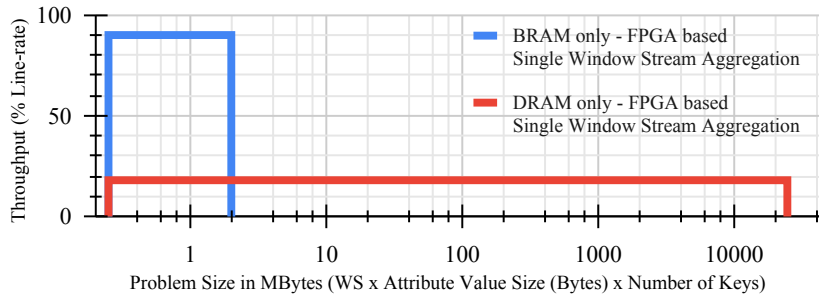


Figure 1.2: Processing throughput as a percentage of incoming tuple network line-rate vs. problem size for an FPGA-based Single-SWAG at 156.25 MHz using only BRAMs (2 MB) or only DRAM (24 GB). WS= 2-96K values; WA=WS; 128K keys and tuple's value size of 2 bytes.

naturally match the stream processing characteristics to provide both high processing throughput and low latency.

Unsupported holistic functions and small problem sizes: Existing FPGA solutions focus on incremental aggregation approaches using multiple window or pane-based designs [6–8]. As a consequence, queries that require holistic functions or small WA have poor performance or are not supported at all. Moreover, most existing FPGA designs do not use external DRAM and therefore support only small problem sizes (WS $\times$ number of keys), which are not practical for many real stream processing problems [5, 10, 11]. This lack of support for holistic functions and larger problem sizes provides an opportunity for building Single-SWAG accelerators using FPGA, and by utilizing the DRAM available in the FPGA platform, to support larger problem sizes.

Memory pressure due to window-update step: For every incoming tuple of a key, the value(s) need to be updated in the single-window corresponding to that key. As illustrated in Figure 1.2, for a particular stream aggregation query, with predominantly window-updates (tumbling windows with WA=WS), running on a FPGA based Single-SWAG system, the higher bandwidth and limited capacity of on-chip BRAM enables line-rate processing but supports only small problem sizes. The larger but lower bandwidth off-chip DRAM can handle larger problems, but with limited performance. This performance gap provides an opportunity to utilize multiple levels of memory hierarchy available in the platform for structuring the single-window per key. The thesis aims to bridge this gap to perform faster window-updates by tapping the benefits of the various levels in the memory hierarchy, and thereby improving the processing throughput of the system.

Memory pressure due to window-aggregation step: For queries with frequent aggregations (small WA), previous Single-SWAG approaches suffer from low processing throughput due to the memory bandwidth bottleneck posed by the large volume of window-aggregation traffic. Existing studies on real-world streaming datasets have shown that a dominant part of them consisting of performance counters, sensor, geolocation and other time-series data have significant redundancy that can be exploited through data compression [14]. As an example, Figure 1.3 illustrates the tremendous potential gains in Single-SWAG processing throughput for frequent ag-

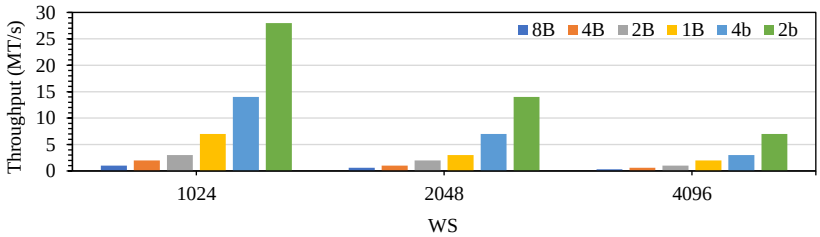


Figure 1.3: Processing throughput in million tuples per second vs. WS in tuples for an FPGA-based Single-SWAG dataflow engine [15] at 156.25 MHz with varying tuple's value sizes from 8 bytes to 2 bits and WA = 1 tuple, showing the potential for data-compression.

gregation queries (WA=1) by reducing the tuple's value size, e.g., up to 28× higher throughput with a 32× data reduction. However, 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. This thesis aims to mitigate the memory pressure due to window-aggregations by incorporating efficient compression schemes in Single-SWAG, thereby, improving the overall performance of the Single-SWAG engine.

1.2 Thesis Objectives and Contributions

The primary objective of this thesis is to mitigate the performance bottleneck of the data-intensive Single-SWAG. We first aim to alleviate the unnecessary data movement seen in CPU/GPU systems by building an *accelerator* using FPGA to achieve high throughput and low latency Single-SWAG. In order to support larger problem sizes, we aim to utilize the external DRAM available in the FPGA platform. Subsequently, the performance of the accelerator is improved by using a *specialized memory hierarchy* for stream aggregation. Finally, we aim to improve the overall performance of the Single-SWAG accelerator by targeting the window-aggregation step using *data-compression*. Below follows a more detailed description of the objectives with some related work and the approach pursued in this thesis.

1.2.1 High Throughput & Low Latency Single-SWAG Acceleration

The first objective of the thesis is to build the Single-SWAG accelerator by alleviating the performance bottleneck due to unnecessary data movement and to support large problem sizes. To this end, two designs are discussed below for supporting the two windowing policies, namely, *tuple-based* and *time-based*.

Tuple-based windowing policy: The first design aims to achieve high throughput and low latency *tuple-based* SWAG. The objectives of this design are to:

- *Support holistic functions;*
- *Support aggregation queries with large WS, small WA, and multiple aggregation functions;*

- *Minimize memory accesses to maximize processing throughput;*
- *Utilize dataflow computing to maximize processing throughput; and*
- *Utilize direct network and off-chip memory connectivity to minimize latency and support large problem sizes.*

Time-based windowing policy: The second design described in this thesis is to support *time-based* Single-SWAG. This design has the following objectives in addition to the ones presented in the previous design:

- *Support fluctuating data arrival rates for time series data; and*
- *Alleviate memory pressure of the single-window approach especially for skewed data distributions.*

Related Work: Generic software approaches, such as Apache Flink, Spark, and Storm, running on general-purpose CPU offer a wide range of stream processing capabilities, including support for time-based and holistic aggregations with ease of deployment but have limited throughput and high latency [16–18]. Other CPU-based sliding-window aggregation algorithms use data structuring and algorithmic techniques such as DABA [2] and MTA [19] and improve latency of in-memory aggregation but are constrained to associative aggregation functions.

One of the best systems that uses GPUs is SABER, a relational stream processing system targeting heterogeneous machines equipped with CPUs and GPUs [20]. SABER achieves high throughput but at high latency of hundreds of milliseconds for aggregation queries. Moreover, it supports only incremental aggregate computations utilizing the commutative and associative property of some aggregation functions and therefore can implement only distributive (`count`, `sum`) and algebraic (`average`) functions. Another work that uses GPUs is Gasser [4]. As opposed to SABER, Gasser supports holistic functions at high processing throughput by offloading batches containing, in the order of thousands of windows to the GPU, which negatively impacts the processing latency. Moreover, Gasser supports only a single key and only tuple-based windowing policy.

Existing FPGA-based stream processing systems supports sliding-window aggregation for distributive (`count`, `sum`, `min`, `max`) and algebraic (`average`) aggregation functions [6, 8, 21–24] . Nevertheless, the existing FPGA designs use only the on-chip BRAMs for storing aggregation states and therefore supports only smaller problem sizes. Moreover, they do not support holistic functions, as it relies on incremental aggregation.

In summary, existing CPU and GPU fall short on performance due to unnecessary data movement and existing FPGA solutions support only smaller problem sizes and do not support Single-SWAG for holistic functions.

Thesis Approach: *Chapters 2* and *3* in this thesis addresses the above limitations describing, FPGA-based Single-SWAG for *tuple-based* and *time-based* windowing policies, respectively.

Chapter 2 describes tuple-based Single-SWAG using FPGA which is a Dataflow Engine (DFE) implemented in a Maxeler N-series FPGA card [25]. The DFE is fed with incoming tuples through a direct network connection and provides direct access

to DRAM through its own memory controller. With the single-window approach, the DFE is able to implement challenging realistic queries of any holistic, distributive or algebraic aggregation functions and using external DRAM, the DFE is able to support large number of keys and window sizes. The single-window design provides high throughput and the DFE's direct network connection and access to external DRAM, enables it to achieve ultra low latency compared to other software and GPU implementations.

Chapter 3 addresses the two additional challenges of time-based single-window stream aggregation [26]. First, the unbounded number of tuples in a time-based sliding-window is facilitated by a flexible circular buffer that stores the window values. We apply the idea of panes [7] to a single-window [9] creating a circular buffer that supports bulk evictions. In addition, this buffer can be expanded dynamically with one or more unused identical buffers originally meant for other keys. Thereby, time-based windows of varying size can be stored. Second, the memory pressure of single-windows, caused by their need to store all incoming data, is alleviated with a caching scheme which is used to merge multiple requests to the same DRAM location without limiting performance in skewed key distribution.

Contributions: We introduce the first FPGA-based accelerator for single-window stream aggregation. Our approach:

- supports realistic aggregation queries with distributive, algebraic, and holistic functions for both *tuple-based* and *time-based* windowing policies and large problem sizes;
- achieves 1-2 orders of magnitude higher processing throughput than a state-of-the-art stream processing software system;
- offers ultra low processing latency of less than 10 μs, at least 4 orders of magnitude faster than software; and
- is at least 1 order of magnitude more energy efficient than a state-of-the-art stream processing software;

1.2.2 Alleviating Memory Pressure due to Frequent Updates

The next design described in this thesis is to improve the processing throughput of Single-SWAG DFE described in the previous section by mitigating the memory pressure due to the *window-update* step. For each incoming tuple, the memory needs to be accessed to update the window. FPGAs can make more efficient use of their memory but existing approaches employ either, only on-chip memory (i.e. SRAM) or the designs in the previous section use only off-chip memory (i.e. DRAM) to store the aggregated values. The high on-chip SRAM bandwidth enables line-rate processing, but only for small problem sizes due to the limited capacity. The larger off-chip DRAM size supports larger problems, but falls short on performance due to lower bandwidth. This design aims to utilize the best of both worlds and the objectives are to:

- *Prevent slow and bandwidth wasteful read-modify-writes in DRAM during window-updates;*

- *Utilize the high on-chip SRAM bandwidth to ensure that the window is always updated at line-rate; and*
- *Utilize the large off-chip DRAM size to support larger problems.*

Related Work: The DRAM-only designs described in the previous section (*Chapters 2* and *3*) requires slow and bandwidth wasteful read-modify-writes in DRAM during window-updates, since value size is typically smaller than DRAM line. Queue buffers composed of two memory types have been designed in the past. More precisely, about two decades ago, 2-level hybrid SRAM/DRAM packet buffers were introduced for network processing [27–31] offering SRAM speed and DRAM capacity. Although our approach is in the same direction, there are several fundamental differences. Firstly, the SRAM/DRAM packet buffers implement queues that support read and write operations at the granularity of a single element and this is less bandwidth demanding compared to the stream aggregation. In addition, a network packet size is at least equal to a DRAM line (64 bytes) and therefore fits DRAM better than the stream aggregation accesses which are often finer and hence require expensive read-modify-write operations. Finally, the hybrid packet buffers were limited to two levels.

Thesis Approach: *Chapter 4* introduces Multi-level Queues (MLQ), the first memory hierarchy specialized for stream aggregation systems aiming to alleviate their memory bandwidth bottleneck [15]. The proposed memory system offers a higher and better utilized bandwidth as well as off-chip DRAM capacity to enable higher processing throughput for larger problem sizes, i.e., WS $\times$ number of keys. Multiple memory levels are used to form logical queue buffers, each buffer storing the contents of a sliding-window for a particular key. Each multi-level logical queue needs to support (i) *single element write* and (ii) *all elements read* operations for window-updates and window-aggregations, respectively. More precisely, for a window-update, a new value needs to be enqueued. The head of the MLQ can be at any memory level, but the tail is always at the fastest (and smaller) first level which is the on-chip SRAM. This ensures that the window is always updated at the highest speed in the fastest level-1 which offers single cycle access and therefore does not require the slow and bandwidth wasteful read-modify-writes to DRAM. Moreover, the more recently received values of each key are at the lower levels supporting better performance. Then, when the window advances, the contents of the entire window are read utilizing the aggregate bandwidth of all memory levels and subsequently, WA number of elements are discarded by just updating the head pointer. Another advantage of this compared to DRAM-only designs is that it handles skewed key distributions without additional support. Compared to a BRAM-only design, MLQ offers higher capacity. Compared to a DRAM-only design, it offers faster window-updates at on-chip SRAM speed as well as faster aggregation using the aggregate bandwidth of all levels, rather than only the DRAM one.

Contributions: We introduce Multi-Level Queues (MLQs), a specialized memory hierarchy for Single-SWAG. Our approach:

- uses multiple memory levels to form logical queues that offer on-chip SRAM (BRAM) bandwidth for window-updates and DRAM capacity;
- supports 4 orders of magnitude larger problems compared to BRAM-only designs; and

- achieves up to 8× higher throughput compared to DRAM-only designs.

1.2.3 Alleviating Memory Pressure due to Frequent Aggregations

The final design described in this thesis is to improve the overall Single-SWAG processing throughput by reducing the memory pressure due to the *window-aggregation* step. Upon aggregation trigger, the entire single-window is read from memory. This generates tremendous memory pressure especially for queries with frequent aggregations (small WA). For queries with frequent aggregations, as the dominant part of the single-window rests in the farthest and slowest memory, previous MLQ approach suffer from low processing throughput due to the memory bandwidth bottleneck posed by the large volume of window aggregation traffic. This design aims to alleviate this bottleneck and the objectives of this design are to:

- *Improve processing throughput of the MLQ system for queries with frequent aggregations; and*
- *Exploit the redundancy in time-series data to reduce the aggregation traffic.*

Related Work: The MLQ system in *Chapter 4* [15] cannot take advantage of the aggregated bandwidth offered by BRAM and off-chip SRAM especially when the dominant portion of the window is stored in DRAM, which becomes the bottleneck. One way to alleviate the window-aggregation memory bottleneck is to compress the sliding-window. In the past, compression has been proposed for stream processing in TerseCades [14]. However, TerseCades only supports batch-processing of separate non-overlapping tumbling windows, i.e., WA=WS, rather than true stream processing with WA $\leq$ 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.

Thesis Approach: *Chapter 5* introduces StreamZip, the first true stream processing engine with compression support for sliding-windows and WA $\leq$ WS [32]. StreamZip is based on previous FPGA-based multi-level queue (MLQ) DFE for SWAG systems (*Chapter 4*) 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. First, the inter-(de)compressor pipeline dependency is solved in StreamZip using multiple sub-streams per key and careful memory packing and alignment of compressed data. Second, intra-compressor-decompressor dependency is solved by speculatively buffering the incoming tuples and bypassing the compressor stage to directly feed the computation of the aggregation functions. Thirdly, the dependency between stream aggregation and the decompressor latency is solved by keeping a tab on outstanding aggregations and feeding the decompressor with this information to aid in exact bulk-eviction of the invalid tuples upon window-slide.

Contributions: We introduce, StreamZip, the first true stream processing dataflow engine with compression support for sliding-windows and any WA using reconfigurable hardware. Compared to the MLQ design in Chapter 4, StreamZip:

- achieves substantial reduction of aggregated data volumes by supporting both lossy and lossless compression algorithms and applied to both fixed- and floating-point numbers;
- achieves up to 7.5× and 22× higher throughput; and
- improves the effective memory capacity by up to 5× and 23×, for lossless and lossy designs, respectively.

1.3 Thesis Outline

The remainder of this thesis is organized as follows. Chapter 2 presents the design and evaluation of tuple-based single-window stream aggregation. Chapter 3 presents the design and evaluation of Time-SWAD, a dataflow engine for time-based single-window stream aggregation. Chapter 4 presents the design and evaluation of the specialized memory hierarchy for stream aggregation. Chapter 5 presents the design and evaluation of StreamZip, a dataflow engine with compressed sliding-windows for stream aggregation. Finally, the conclusions of this thesis are presented in Chapter 6.

Chapter 2

Single Window Stream Aggregation using Reconfigurable Hardware

With the recent technological advances the number of connected devices grows rapidly along with the total amount of data they produce. New emerging applications analyze unbounded streams of such *big data* in various domains (e.g. financial, transportation) to make fast, sophisticated decisions. However, real-time analytics of large data streams require *high processing throughput* to cope with massive volumes of data as well as *low latency* to respond in real-time.

Stream aggregation is one of the most challenging and computationally intensive analysis tasks in stream processing. This can be handled by applying the traditional relational database aggregation semantics to a sliding window of a particular size (Window Size - WS). Such window can then be "slid" by a particular number of elements (Window Advance - WA) as to produce new aggregated values. The result is a stream of aggregated values. Incremental aggregation – using multiple windows or panes [6–8] to compute and store partial results rather than storing all the incoming values – has been employed to improve performance and reduce memory pressure (both capacity and bandwidth). However, for some queries, especially with small WA, incremental aggregation has the opposite effect causing an excessive number of memory accesses that limit performance. This is for instance the case in streaming applications processing geo-tagged data [5], social media data [10] or manufacturing equipment data [11]. In such cases, a single window approach that explicitly stores all the incoming values in a single window before processing is more efficient. More importantly, storing values in a single window is unavoidable when computing some holistic functions; these are functions that have no constant bound on the size required to store a partial result, such as `median` [12].

Stream processing and stream aggregation systems in particular have been implemented on various compute platforms. Multicores and GPU are able to sustain high processing throughput but fall short in delivering low latency [13]. They require redundant memory accesses managed by an operating system to store incoming tuples in DRAM even before processing starts. This, besides the long latency, wastes a significant fraction of valuable memory bandwidth reducing performance. On the con-

trary, FPGAs provide both high processing throughput and low latency. Customized dataflow engines, which naturally match the stream processing characteristics, can be implemented in reconfigurable hardware achieving high throughput [13]. Furthermore, incoming tuples can be fed to an FPGA through a direct network connection with low latency, avoiding unnecessary DRAM accesses.

So far, current FPGA solutions focus on incremental aggregation approaches using multiple window or pane-based designs [6–8]. As a consequence, queries that require small WA or use holistic functions have poor performance or are not supported at all. Moreover, most existing FPGA designs do not use external DRAM and therefore support a single key or at most a handful of keys, and small window sizes, which are not practical for many real stream processing problems, such as the ones mentioned before [5, 10, 11].

This work addresses the above limitations describing, to the best of our knowledge, the first FPGA-based design for single window tuple based stream aggregation. A Maxeler N-series card is used for the design of a stream aggregation dataflow engine (DFE) [25]. The DFE is fed with incoming tuples through a direct network connection and provides direct access to DRAM through its own memory controller. The main contributions of this work are the following:

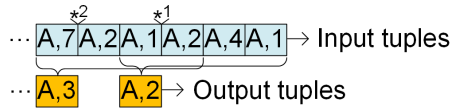
- The first FPGA-based design for single window stream aggregation, which
  - uses a Maxeler's Dataflow Engine and deep pipelining to provide processing throughput of up to 8 million tuples per second (1.1 Gbps), and
  - has a direct network connection to feed incoming tuples as well as direct access to DRAM offering ultra low end-to-end latency of up to 4 μsec.
- An implementation of the above design able to support multiple challenging realistic streaming queries with
  - holistic and arbitrary user-defined aggregation functions, as well as distributive and algebraic ones;
  - up to 1 million concurrently active keys;
  - large window sizes storing up to 6144 values per key.

The remainder of this chapter is organized as follows. Sections 2.1 and 2.2 offer background on data stream aggregation and present related work, respectively. Section 2.3 describes the proposed design for single window FPGA-based stream aggregation. Section 2.4 presents the evaluation results. Finally, Section 2.5 summarizes our conclusions.

2.1 Background - Stream Aggregation

In this section, we present the semantics of stream aggregation and provide an overview of the main implementation strategies discussed in the literature. In addition, we elaborate on the two main arguments that motivate our implementation choice among the existing implementation strategies for stream aggregation. More precisely, we count the memory accesses for each implementation strategy, showing that for some problems the algorithm we implement in hardware incurs the lowest DRAM Read/Write

(i)



(ii)

Internal states at time instance	*1	*2
Multi-Window (overlapping windows' counters)	7, 2	5, 2
Pane-Based (Panes' counters)	5, 2	3, 2
Single-Window (tuples' values)	1, 4, 2	2, 1, 2

*Figure 2.1: Example of stream aggregation over a stream of tuples composed by schema $\langle char, int \rangle$ for parameters F=`mean`, WS=4, WA=2 and K=char (i) and internal states maintained by the different implementation strategies at time instance *1 and *2 (ii).*

(R/W) operations per tuple. In addition, we explain why for some holistic functions explicitly storing all incoming values is necessary.

A stream is an unbounded sequence of tuples $t_0, t_1, \ldots$ each tuple containing n attributes $\langle A_1, \ldots, A_n \rangle$. When aggregated, tuples are fed to one (or multiple) function F such as `sum`, `mean`, `min`, `max` or `count`, among others. Since streams are unbounded, the aggregation is performed over a *sliding-window* of *size* WS and *advance* WA. Optionally, aggregation can be performed specifying a parameter K (a subset of the tuple's attributes, also referred to as *key*). In such a case, F is computed for each distinct value observed for K (*group-by*). Figure 2.1 presents an example in which a stream of tuples with schema $\langle \text{char}, \text{int} \rangle$ are aggregated using parameters F = `mean`, WS = 4, WA = 2 and K = char.

Different strategies exist to implement stream aggregation's semantics. The *Multi-Window* (MW) [6] strategy maintains the partial aggregated state of all the overlapping windows to which each tuple contributes. As shown in Figure 2.1, the partial state for F = `mean` is the sum of the values observed so far, which is then divided by WS once the window is full. When a window is full, a result is output and the window is discarded. Each tuple contributes to $\lceil \frac{\text{WS}}{\text{WA}} \rceil$ windows. Upon reception of a tuple, all the overlapping window's states are updated. When the size of the state maintained for each window in DRAM is S, the overall bytes for a key are $\lceil \frac{\text{WS}}{\text{WA}} \rceil \times \text{S}$. Windows are updated upon reception of a tuple incurring $\frac{\lceil \frac{\text{WS}}{\text{WA}} \rceil \times \text{S}}{\text{B}}$ R/W from DRAM, where B is the burst size. Furthermore, for every WA tuples, one extra R/W is required to produce the result[1]. The average R/W per tuple and per key is $\frac{1}{\text{WA}} + \frac{\lceil \frac{\text{WS}}{\text{WA}} \rceil \times \text{S}}{\text{B}}$.

The alternative *Pane-Based* (PB) strategy [7, 8], partitions the window into panes of length WA[2] and maintains partial aggregated states for the latter. As shown in Figure 2.1, the partial state for F=`mean` is the sum of the values observed for each pane of WA tuples. Upon reception of a tuple that fills a window, the result is computed by aggregating its partial states, and the stale panes of the window are then discarded (in the example, summing each pane value and dividing by WS). Finally, the contribution of the tuple is added to the partial aggregate state of the rightmost pane. In this case, when the size of the state maintained for each pane is S, the overall bytes for a key are $\frac{\text{WS}}{\text{WA}} \times \text{S}$. Upon reception of a tuple, updating the last pane incurs one R/W from

[1] This conservative estimation assumes one window's state fits in a burst.
[2] Assuming WS is a multiple of WA.

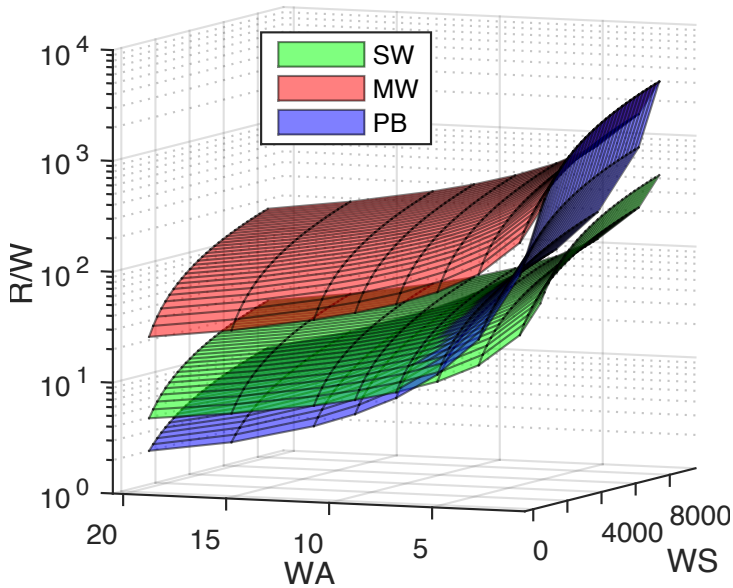


Figure 2.2: Average number of memory accesses (R/W) per tuple incurred in MW, PB and SW stream aggregation implementations when computing the average, the top-3 highest, and the top-3 lowest values of a stream of values.

DRAM[3]. Furthermore, for every WA tuples, $\frac{\lceil \frac{WS}{WA} \rceil \times S}{B}$ R/W are required to produce the result. The average R/W per tuple and per key are then $1 + \frac{1}{WA} \times \frac{\lceil \frac{WS}{WA} \rceil \times S}{B}$.

A third strategy, the *Single-Window* (SW), maintains tuples rather than partially aggregated states. With SW, WS tuples are maintained for each key (as shown in Figure 2.1) and the results are computed every time the window is full. Being A the size in bytes required to store the attributes to be aggregated, the state is equal to $WS \times A$. In the general access scheme of the DRAM memory, SW requires one R/W for the insertion a tuple and $\frac{WS \times A}{B}$ R/W for the output of a result. Since the latter happens once every WA tuples, the average R/W per tuple and per key are then $1 + \frac{1}{WA} \times \frac{WS \times A}{B}$.

Figure 2.2 shows the average memory accesses (reads and writes) per tuple incurred by the different strategies when computing the `mean`, the `top-3 highest` and the `top-3 lowest` values of a stream of integer values. For MW and PB, we maintain the sum of the incoming values (as for the example in Figure 2.1), 3 values for the `top-3 highest` and 3 values for the `top-3 lowest` values, thus incurring a state S of 28 bytes. For SW, A requires 4 bytes. We assume the burst size S is of 64 bytes. As shown, the average number of R/W is lower for SW when several functions aggregate large windows with small advance, as is the case in this work. In the example, the SW approach fits best large windows producing continuous up-to-date results (*i.e.*, with small window advance) by incurring up to $7\times$ fewer memory accesses (for WA=1 and the largest WS) compared to MW and PB (500 for SW instead of 3494 for MW and PB). As explained in Section 2.4.3, even in our SW experiments, performance is limited by DRAM bandwidth. Then, MW and PB implementations would have

[3]Assuming one pane fits in a burst.

a substantially lower performance due to their need for substantially higher DRAM bandwidth. As a consequence, SW is a faster stream aggregation choice for queries with such WA and WS parameters.

Another disadvantage of MW and PB implementation is that for holistic functions, maintaining all input tuples is necessary independently of whether the function can be computed incrementally. This is for instance, the case for the `median` function as all the tuples contributing to the window need to be kept and sorted before the median itself can be computed. This is due to the non-associative nature of the median function. For such functions, a SW implementation is more efficient. MW solutions would require storing all incoming tuples in each one of the multiple windows resulting in many duplicates (exacerbating performance bottleneck) and PB approaches would not be able to maintain partial results.

In summary, for small WA and large WS as well as for particular functions, SW is more efficient for supporting stream aggregation. This motivates our algorithmic choice, the design of which is described in Section 2.3.

2.2 Related Work

In this section, we highlight related works that use various computing platforms for stream processing. In addition, we discuss some techniques proposed for in-memory databases (IMDBs), which are closely related to ours, although they follow a store-and-process paradigm rather than on-the-fly real-time stream processing. Distributed Stream Processing Engines (SPEs) running on conventional CPUs like Apache Flink, Spark, and Storm provide generic stream processing capabilities and ease of deployment [16–18, 33]. In particular, Apache Flink is an open-source Java based state-of-the-art stream processing framework. These software-based distributed stream processing engines are easy to configure, 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 aggregating on large windows (large WS) and at high rates (small WA).

SABER is a relational stream processing system targeting heterogeneous machines equipped with CPUs and GPUs [20]. SABER achieves high throughput but at high latency of hundreds of milliseconds for aggregation queries. Moreover, it supports only incremental aggregate computations utilizing the commutative and associative property of some aggregation functions and therefore can implement only distributive (`count`, `sum`) and algebraic (`average`) functions.

Glacier is an FPGA-based streaming query to hardware compiler which supports sliding window aggregation for distributive (`count`, `sum`, `min`, `max`) and algebraic (`average`) aggregation functions [6]. This implementation relies on a MW approach, instantiating multiple aggregation compute modules, resulting in poor scalability in terms of resource utilization and performance. Oge *et al.* improves the aggregation logic in [6] using the PB approach [7], thereby making the design scalable with increasing WS/WA ratio [8]. Nevertheless, both designs use only the on-chip BRAMs for storing aggregation states and do not use DRAM. In turn, this prevents the design from

performing stream aggregations of realistic sizes (number of keys, WS). Moreover, Glacier does not support holistic functions such as `median`, as it relies on incremental aggregations. Finally, Oge *et al.* support only a single key and do not support *group-by* clauses in the query [8, 34].

A considerable number of previous works have focused on accelerating IMDB operators using FPGAs [35]. For instance, István et al. used an on DRAM hash table [36], which was subsequently improved using a Cuckoo hash table [37], similar to our design. Another interesting work [38], uses an FPGA-based Convey HC-2ex machine to support high throughput *group-by* in-memory DB aggregation, but implements only the `count` aggregation function.

In this work, we propose a single-window stream aggregation implemented on reconfigurable hardware. Our design achieves similar throughput and latency (millions tuples/sec and few μsec, respectively) as other FPGA-based solutions, but is able to produce more complex operations and with larger state. It further confirms prior art conclusions that GPUs have in general much higher stream processing latency than FPGAs [13], as our design achieves 3 orders of magnitude lower latency than related GPU approaches, although none of them has the exact same query semantics considered here. Finally, as shown in Section 2.4, our approach supports substantially higher throughput and lower latency than software.

2.3 A Data-Flow Engine for Single Window Stream Aggregation

Data-flow computing matches well the requirements of stream processing. A deep, feed-forward-only pipeline with lightweight control for a back-pressure stall mechanism is able to process fast large volumes of incoming data. We propose a reconfigurable dataflow design for single window stream aggregation. A Maxeler N-series card [25] is used to host our design, implemented as a Data-Flow Engine (DFE), further providing a direct network connection and DRAM access to minimize the processing latency. Although this Maxeler system fits well our design requirements, our approach is general and could be ported to other platforms that exhibit similar characteristics.

Figure 2.3 shows the top level block diagram of the proposed design. Incoming tuples containing a timestamp, a key, and a value, are carried by network packets and received by the receiver module (*Rx*). Their keys are hashed to index a hash table, which stores metadata per key, needed for the subsequent processing stages. After accessing the hash table, multiple concurrency control queues are used to enqueue the tuples and resolve dependencies between them. Dequeuing from the queues is performed in a round-robin fashion allowing only for a single tuple per key to be in-flight. Forwarded tuples trigger an access to DRAM to read and subsequently update the stored values (state) of the respective key. When the number of tuples per key reaches the WS threshold, DRAM accesses are issued to read all the values of the corresponding window and compute the aggregation function in the compute stage. The result of the aggregation function is finally transmitted back through the Tx module.

Note, that the flow of the data through the stages is controlled through FIFOs responsible for stalling the pipeline via a back-pressure mechanism when needed as well as for triggering a stage when valid data are available. Below, each stage of the

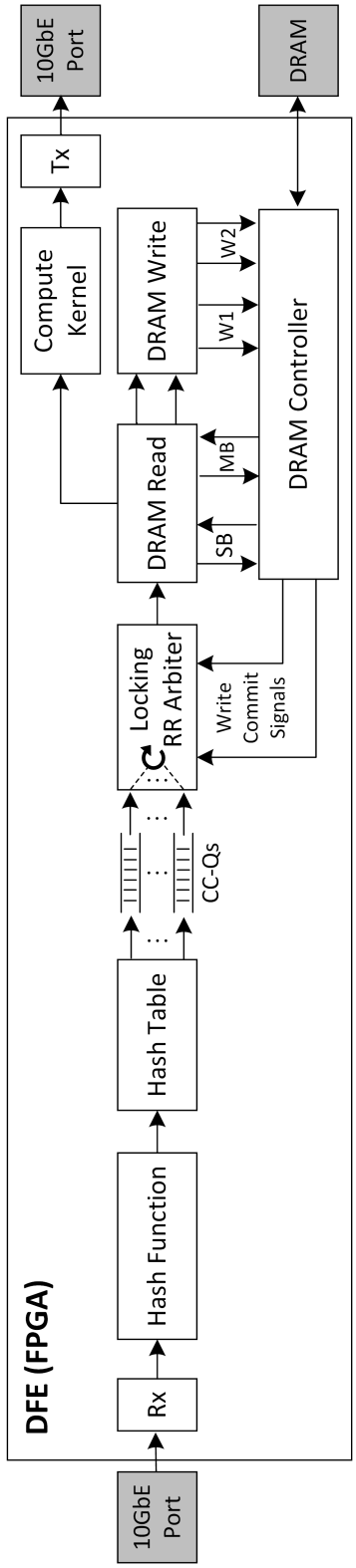


Figure 2.3: Top-level view of the Dataflow Engine for single window stream aggregation. The grey boxes indicate peripherals outside the FPGA.

stream aggregation engine is described in detail.

2.3.1 Receiver and Transmitter

The receiver *Rx* and transmitter *Tx* modules handle the incoming and outgoing network packets, respectively, supporting the network protocol processing tasks (TCP, UDP, or Ethernet). *Rx* and *Tx* receive and transmit packets from/to the 64-bit wide physical DFE link through protocol specific bi-directional streams. Each packet carries multiple tuples of the following form $\langle ts(8), key(4), value(4) \rangle$, containing in total 16 bytes, of which 8 bytes used for a timestamp, 4 bytes for the key and 4 bytes for the actual value of the tuple.

2.3.2 Hash Function

The incoming tuples are unpacked and hashed using their key to generate a hash table address (*ht_adr*). Bob Jenkin's Lookup3 hash function [39] is used for the hashing as proposed in [36]. Such functions are proven to produce reduced number of collisions due to the *Avalanche* effect, whereby, keys that differ even by a single bit produce different hash values. Two hash functions are computed in parallel, the second one used in case of collision of the first. This stage adds the *ht_adr* to the tuple and forwards it to the next stage.

2.3.3 Hash Table

The hash table, shown in Figure 2.4, stores metadata for the active keys handled in the system. In order to fit the hash table for a large number of keys using the on-chip BRAMs, our current implementation is direct mapped, having each entry corresponding to a single key. Alternatively, the handling of hash collisions could be improved by adding associativity to the hash table as described in [36]. In such setup, a table entry would offer multiple locations to store a key and a least recently used (LRU) policy would be used for replacement.

As shown in Figure 2.4, each hash table entry contains (i) a counter for the number of values (tuples) stored in DRAM for a particular key window as to determine when the key context is ready for aggregation, and (ii) a pointer to the last stored value in DRAM (*head pointer*). Optionally, a timestamp or other fields could be added as the application demands it.

As explained in Section 2.3.5, each entry in the hash table has a statically allocated DRAM memory region to store the values of the respective key. Static allocation simplifies considerably the hardware design compared to a dynamic memory management scheme. The size of this DRAM region determines the maximum window size supported. Due to this static DRAM allocation, after a hash table access, the DRAM location for accessing the stored values of the corresponding key can be calculated using (i) the hash table address storing the key, and (ii) the head pointer to the last stored value.

In general, a tuple accessing the hash table, finds the DRAM location of its key window and decides whether the computation of the aggregation function is triggered. After reading the hash table, the same hash table entry is updated with a new counter-value and head-pointer.

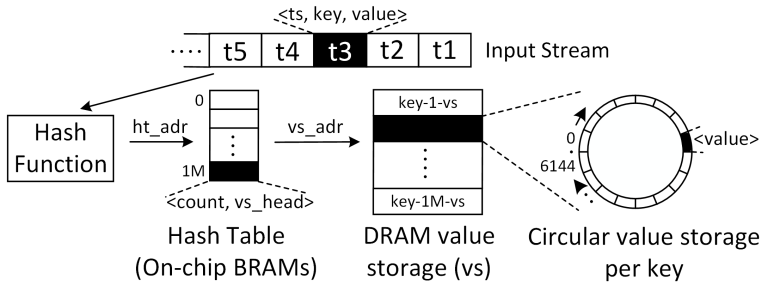


Figure 2.4: Hash table and external DRAM organization.

2.3.4 Concurrency Control Queues

After the hash table stage, the tuples are distributed among a set of concurrency control queues (*cc_Qs*) using the least significant bits of their *ht_adr*. Tuples that belong to the same key are sent to the same *cc_Q*. Each queue has a fixed time-slot to send a tuple forward and is selected in a round-robin fashion. The *cc_Q* arbiter has an additional locking functionality in which it locks the queue as soon as a tuple is pulled out. The arbiter waits for the DRAM write commit signal, indicating that the tuple has updated its key window, before unlocking again the queue. This is done to prevent read-after-write hazards in the pipeline as a result of issuing multiple requests concurrently.

Ideally, blocking of tuples would be avoided (assuming uniform random distribution of keys) when using a number of *cc_Qs* equal to the delay (in FPGA cycles) of the pipeline part that comes after the concurrency control queues (including DRAM accesses). In practice, we use 128 queues without limiting DFE performance or exhausting FPGA resources.

A similar concurrency control mechanism was proposed for an in-memory DB system in [36]. However, that work uses single registers, rather than entire queues to store tuples. As a consequence, their pipeline is stalled when an incoming tuple needs to be written to an already occupied register. On the contrary, using queues, as proposed in our design, offers higher flexibility and better load balancing, resulting in fewer stalls.

2.3.5 DRAM access

The values of each active key, forming a single window, are stored in the external DRAM accessed directly through a memory controller module in the FPGA. Next, we describe the placement of the key values in DRAM as well as the way DRAM reads and writes are performed.

2.3.5.1 Placement of key values in DRAM

The DRAM capacity is equally divided statically to the hash table entries, corresponding to the total number of (active) keys supported. For example, in our platform the total DRAM capacity is 24GB and considering that the total active keys supported is 1 million, each key is allocated a maximum single window value storage of 24 KB. We use the entire DRAM as a single monolithic block in single channel mode with

a random access pattern and a single DRAM burst of 192 bytes (*single_burst_size*). The number of DRAM bursts corresponding to the maximum single window size is $24KB/192 = 128$. So, for values of size 4 bytes (*value_size*), the maximum single window size per key is $24KB/4B = 6144$.

The values of a key in the single window are organized in a circular buffer fashion as shown in Figure 2.4. This is possible as we implement the tuple-based class of sliding window stream aggregation in which the window advance is a fixed number of tuples [40]. Consequently, this allows to overwrite the stale entries in the single window in a circular fashion[4]. The WS and WA in the continuous stream query may be parameterized at runtime. The maximum value of WS is the maximum single window size per key. WA can vary from 1...WS. Using the runtime parameterized WS, the *no_of_bursts_per_key* can be calculated as $\text{WS} \times value_size/single_burst_size$. For example, if the query has a WS of 3072, then the number of bursts required per key is 64. The granularity of a data word in the DRAM read/write data stream is a single DRAM burst of 192 bytes.

2.3.5.2 DRAM Read

As a tuple along with its control bits enters the DRAM stage, it is known whether an aggregation computation is triggered, or otherwise a new value should just be added to the window. In the former case, a multiple-burst read is issued with start address as *vs_adr*. In the latter case, a single read is issued at address $vs_adr + last_vs_block$ (*last_vs_block* is retrieved from the *head pointer*). As there are two read streams corresponding to single and multiple bursts, there are two internal queues to buffer the input data words. Once the read is issued, the tuple is placed in a queue, waiting until the memory controller returns with a response.

When DRAM responds to a single read, the tuple awaiting is dequeued and the word read from the DRAM is updated with the new key value and forwarded to the DRAM write stage along with the proper control bits specifying the DRAM location to write back the updated data. Using the above internal queue to delay the tuples with pending DRAM responses, synchronization is achieved between the incoming tuple and its values read from DRAM.

When DRAM responds to a multiple burst read request, a counter with maximum value equal to the burst length (*no_of_bursts_per_key*) is used to ensure that all the values needed for aggregation are received. Subsequently, the values are forwarded to the compute stage together with the tuple. Note that the new key value carried by the tuple that triggered the aggregation still needs to be written back to DRAM updating the key values. Consequently, a write back to DRAM is performed parallel to the computation of the aggregation.

2.3.5.3 DRAM Write

All writes to DRAM are single burst writes. This stage receives two streams coming from the DRAM Read stage, one for tuples that issued a single word DRAM read, and a second for tuples that issued a multiple burst DRAM read. These two streams bring DRAM write requests, which are sent to the memory controller and subsequently forwarded to the DRAM.

[4]For time-based windows, static memory allocation would limit the maximum number of values arriving within a time slot that defines the window. It would further require variable number of values to be updated in a WA.

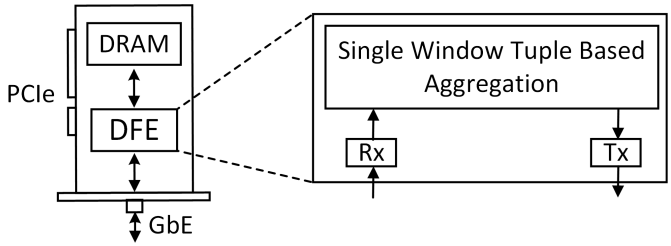


Figure 2.5: Maxeler DFE used for SW stream aggregation.

2.3.6 Compute Stage

The compute stage calculates the implemented aggregation function, using the values read from DRAM, which correspond to the window of the particular key. Depending on the function in the query (distributive, algebraic, holistic), the values may be processed gradually as they arrive from DRAM (e.g. partial aggregation), or otherwise only when all values in the window have been received. For example, algebraic aggregation functions like `average` and distributive functions such as `minimum`, `maximum`, and `sum` can be partially computed on-the-fly for each burst and then the result accumulated to find the final aggregate of the entire window. But for holistic aggregation functions such as `median`, the entire single window should be received before triggering the aggregation. Note that in our implementation of a median aggregation function, the bitonic network sorting algorithm is used, which is a parallel sorting algorithm and conforms to the dataflow paradigm [41].

After the computation of the function is completed the result is forwarded to the Tx stage. Note that the design effort for implementing different queries in our design mainly lies on modifying the compute kernel.

2.4 Evaluation

The proposed approach is evaluated in terms of performance and power consumption implementing different queries. First, the experimental setup is discussed. Then, the FPGA resource utilization of our design is presented. Finally, processing throughput, latency and power results are reported and compared to a state-of-the-art stream processing software.

2.4.1 Experimental Setup

The block diagram of the experimental platform is shown in Figure 2.5. A Maxeler N-series ISCA (MAX4AB24B) PCIe card with an Altera Stratix V (5SGXAB) was used.

The card provides direct 10 GbE network connection to the FPGA. It is further equipped with three 8 GB DDR3 DIMMs accessible directly from the FPGA (24 GB in total). The off-chip DRAM has a maximum bandwidth of 38.4 GB/s and an average latency of about 500 ns. The total on-chip BRAMs available in the DFE is 6.6 MB. Our design is implemented in MaxJ, a Java based High level Synthesis (HLS) language and compiled to FPGA bitstream using MaxCompiler.

Table 2.1: Resource utilization for the FPGA implementation.

Resource	Query 1	Query 2	Query 3
Logic (ALMs)	89853 (25.01%)	222275 (61.9%)	223364 (62.18%)
BRAMs	1856 (70.30%)	1840 (69.7%)	1906 (72.20%)

Our implementation is compared with the state-of-the-art open source stream processing software framework, Apache Flink (v1.2.1). Flink processing was set up in a workstation with an Intel® Core™ i7-4790 CPU running at 3.6 GHz and 24 GB DDR3 DRAM. Prior FPGA/GPU works cannot be directly compared with our design as they target stream aggregation queries with different semantics.

The data set is uniformly distributed and the tuple's keys and values are generated with uniform distribution. The *tcpreplay* is used to inject captured packets [42]. We experimented with WS up to 6000 and WA ranging from 1 to WS. Our experiments have used a finer grain of WA for the smaller sizes (up to 10) as this is the region of interest for the single window implementation.

The queries which we implemented can be expressed in a streaming relational model [40]. To make the queries intuitive, a subset of the LinearRoad benchmarking system [43] is used, where each vehicle has a sensor that emits tuples composed of a timestamp, a vehicle ID (key), and speed (value).

The following queries were used in our evaluation:

Query 1: Find the `average`, `minimum`, and `maximum` speed for each vehicle for the last *WS* tuples and return the aggregate every *WA* tuples.

Query 2: Find the `median` speed for each vehicle for the last *WS* tuples and return the aggregate every *WA* tuples.

Query 3: Find the `median`, `average`, `minimum`, and `maximum` speed for each vehicle for the last *WS* tuples and return the aggregate every *WA* tuples.

2.4.2 Resource Utilization and Maximum Frequency

The resource utilization of the proposed design is shown in Table 2.1 for each query. About 70% of the BRAMs are used for the hash table and the queues employed in our design. It can be further observed that the *median* operation requires more logic resources as it uses a sorting network. The reported resource utilization corresponds to the maximum WS of the queries. For Query 2 and 3, the maximum WS is determined by the largest bitonic sorting network that could fit in the available FPGA resources, which is for a WS of 144. An alternative sorting algorithm such as a merge-sort could have been used but it would still have a storage limitation as all values need to be kept in the FPGA memory to produce the final result. Query 1 operates at 150 MHz. Queries 2 and 3 have more complex implementations and operate at 100MHz.

2.4.3 Throughput

The *processing throughput*, *i.e.*, the number of input tuples processed by the system per unit of time, is measured for every query and compared to the software implementation (Flink).

Figure 2.7 and 2.6 depict the throughput for different Window Sizes (WS), Window Advance (WA) for both the FPGA-based (FPGA), in solid line, and software (Soft), in dashed line, implementations for Query 1. As the WA increases, the throughput

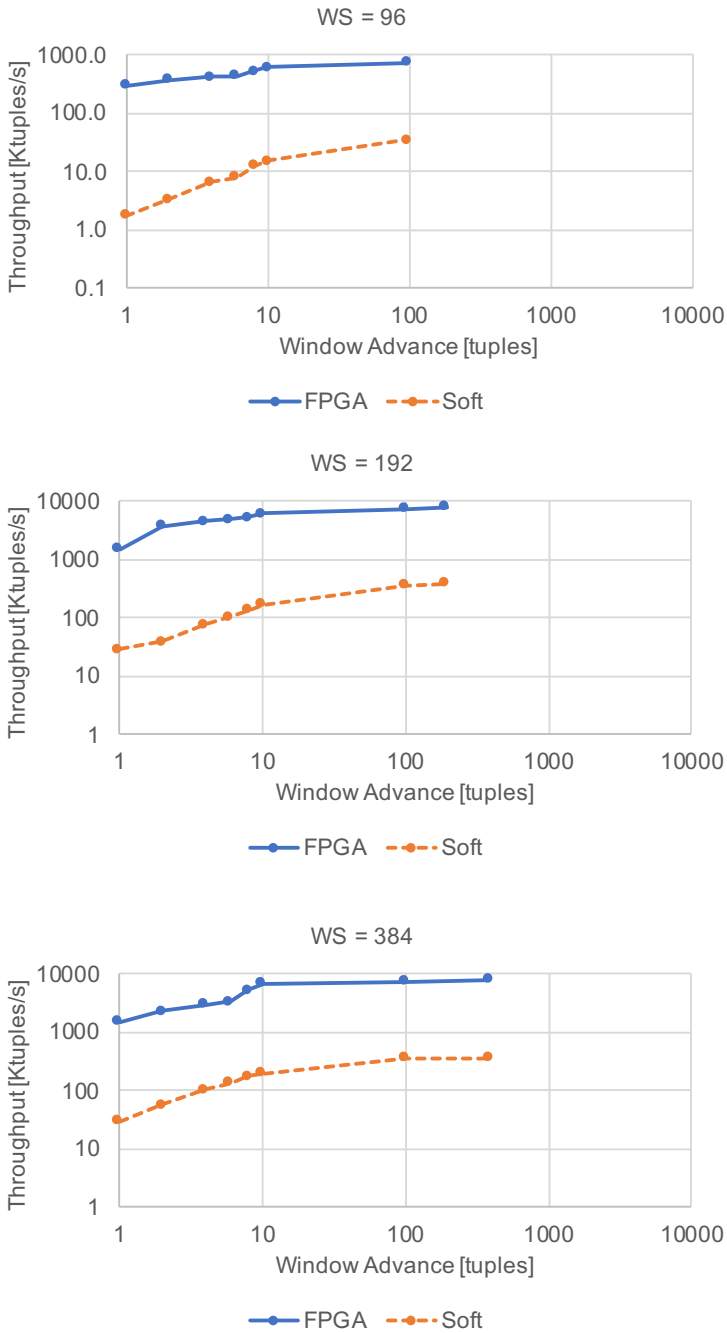


Figure 2.6: Throughput for Query 1 for Window Sizes: 96, 192 and 384 tuples.

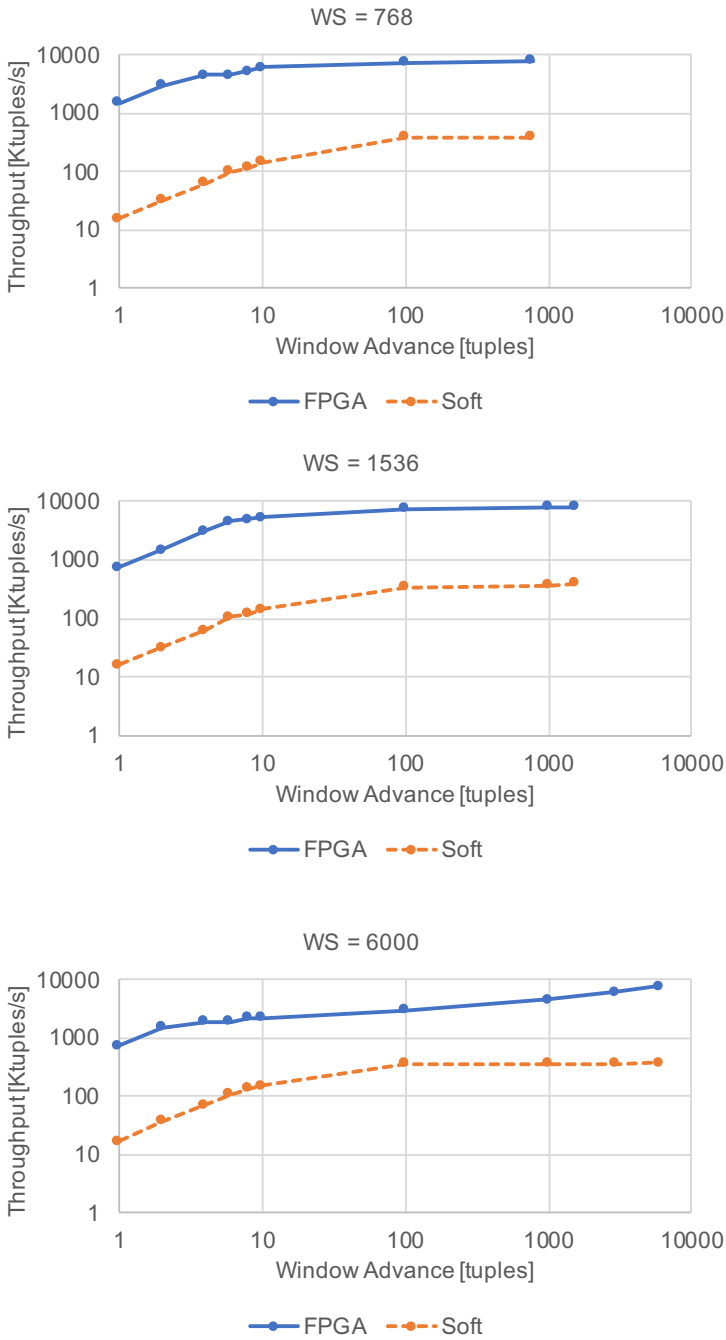


Figure 2.7: Throughput for Query 1 for Window Sizes: 768, 1536, and 6000 tuples.

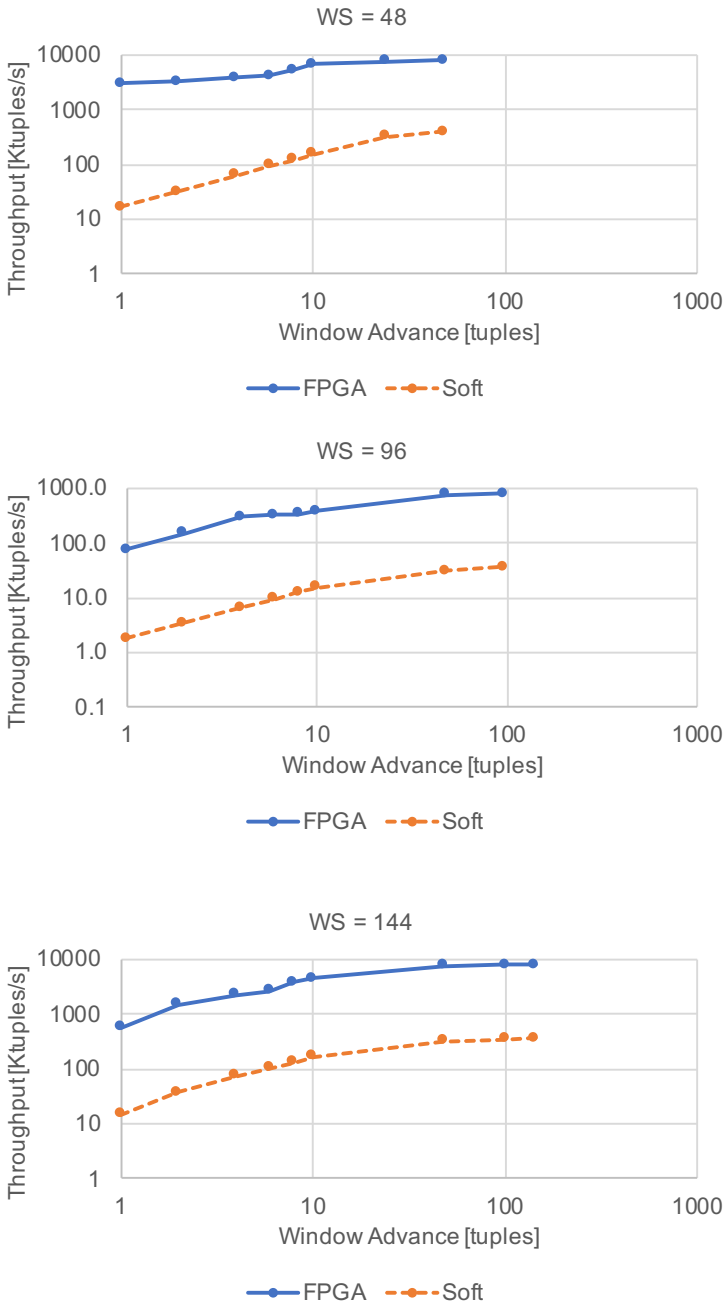


Figure 2.8: Throughput for Query 2 for Window Sizes: 48, 96, and 144 tuples.

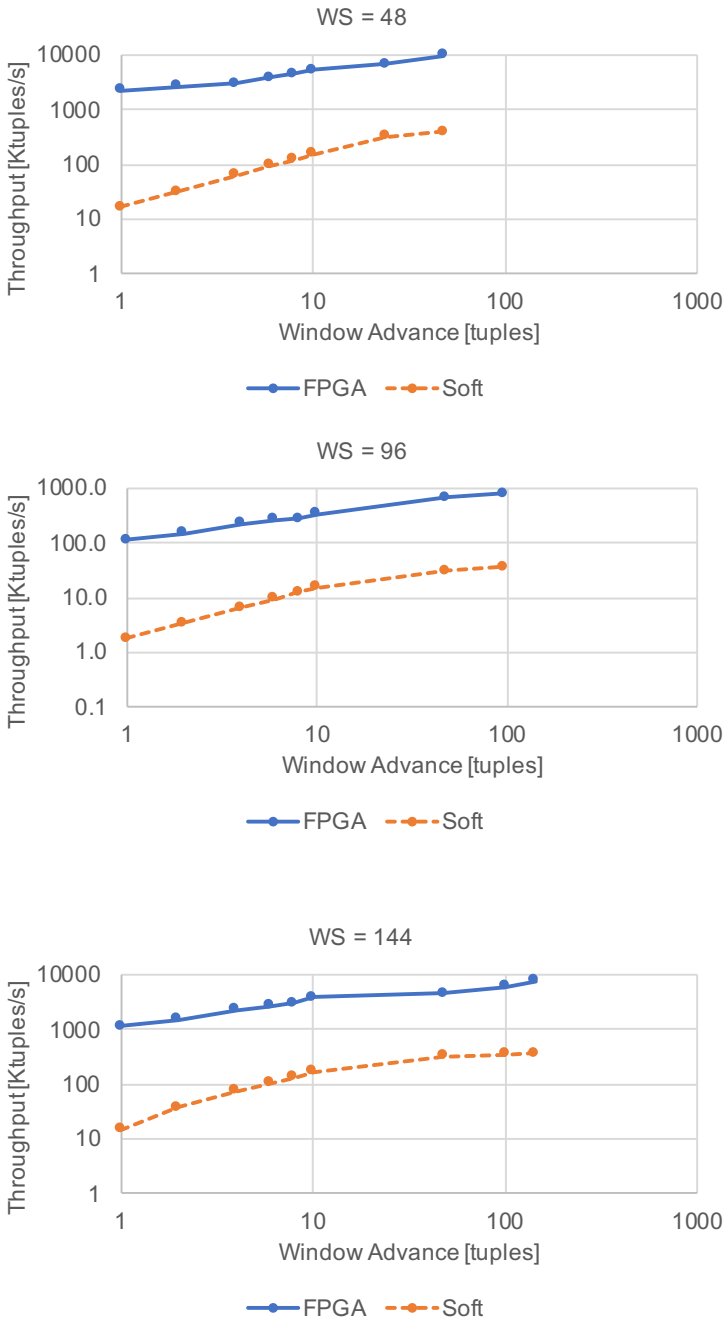


Figure 2.9: Throughput for Query 3 for Window Sizes: 48, 96, and 144 tuples.

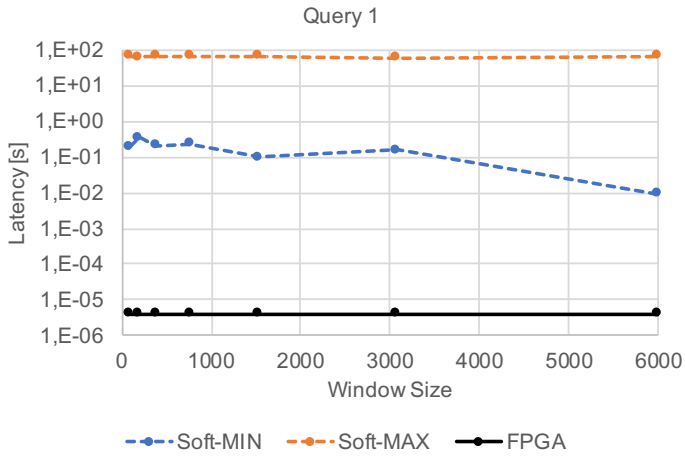


Figure 2.10: Latency for Query 1.

also increases. This is expected as the number of aggregates to be produced decreases significantly with the larger WA because the aggregation function is triggered less often. For small values of WA, the software implementation appears to suffer more than the FPGA implementation. This gap closes for larger WA values. The most important observation is that the throughput achieved by the software implementation is always significantly lower by a factor of 13× up to 173× than the FPGA-based implementation, which is able to process 8 million tuples per second.

The charts in Figures 2.8 and 2.9 show the throughput for WS of 48, 96, and 144 for Queries 2 and 3 respectively, for both the FPGA and software implementations. For these queries, the trend is the same as in Query 1: throughput increases as the WA increases, and FPGA throughput is substantially higher than software by a factor of 20× up to 185×.

Overall, stream aggregation in our platform is memory bounded. Considering the operating frequency of our design and the Ethernet bandwidth (10Gbps), DFE would be able to process about 80 Mtuples/sec if not limited by the external DRAM bandwidth. However, the throughput observed in our experiments is an order of magnitude lower.

2.4.4 Latency

For the same set of experiments we also measured the average tuple *latency*. That is the average time spend by a single tuple in the system, from the time it enters until its processing is completed. Obviously, the worst case latency occurs when the compute stage is triggered to produce a result for the aggregation function. The contribution of computing the aggregation function in the worst-case overall latency, which is about 4 µs, is 0.67 µs, 0.79 µs, and 1 µs for Queries 1, 2, and 3, respectively. In this Section, we show only the latency for Query 1 since it is the simplest and thus the one where the software-based system exhibits the lowest latency (about 1-3% lower than Queries 2 and 3). For the FPGA-based system, the latency is approximately the same across all queries.

Figure 2.10 shows average tuple latency for different window sizes of Query 1 implemented in software and FPGA. As opposed to the FPGA implementation, in software, WA has some impact in the average tuple latency. Consequently, for software we report the minimum and maximum (average) tuple latency measured for each WA. While not clear in this chart, the software latency is lower as the WA increases. This is because the aggregation function is triggered less frequently and thus the system load is not as high. For the FPGA system the difference is negligible and thus only a single line is shown.

The latency for the software system ranges from approximately 0.01 up to 70 seconds. For the FPGA system the latency is significantly lower, approximately 4 μsec. Even though the operating frequency of the FPGA-based system is much lower (100-150 MHz for the FPGA system as compared to 3.6 GHz for the software) the Maxeler N-series system provides a dedicated FPGA implementation that is substantially more efficient than the general-purpose CPU implementation. In particular, the latency for the FPGA system is 4 to 7 orders of magnitude lower than the latency of the software system.

2.4.5 Power

The *power* consumption for both the FPGA-based and software implementations is measured. For the CPU we measured the power consumption using the processor performance counters. This gives package power to which we added DRAM power. For the FPGA we used the available tool to measure the total power consumed by the ISCA board including the FPGA, DRAM and QSFP port.

While for the CPU, power consumption depends on the executed scenario, i.e. the particular query, WS and WA, for the FPGA it is relatively stable. CPU power consumption ranges between 13.7 W to 48.4 W while for the FPGA it is 23.9 W to 29.7 W. Notice that for certain cases the CPU power is actually lower than the FPGA power but for most cases it is almost double.

More relevant than the absolute power consumption is the energy efficiency achieved by each architecture. We define energy efficiency as the Performance (number of tuples processed per second) per Watt. In our experiments, energy efficiency of our FPGA-based design is 10× to 36× better than the software CPU implementation.

2.5 Conclusion

High throughput and low latency stream aggregation is critical for various emerging stream processing applications in order to process large data volumes and produce real-time responses. High-end multicores and GPUs are able to support high throughput stream processing, but fall short in delivering low latency. On the contrary, FPGA platforms can provide both. Still, previous FPGA-based solutions offer only incremental stream aggregations, which is not acceptable for certain queries. This work describes the first FPGA-based single window stream aggregation engine. It is able to implement challenging realistic queries of any holistic, distributive or algebraic function and support up to 1 million keys, storing windows of 6144 values per key. Our approach is implemented in a Maxeler N-series dataflow engine, which offers a direct network connection and access to external DRAM. The proposed design achieves 1-2 orders of magnitude higher processing throughput than a state-of-the-art stream

processing software system at up to 2× lower power cost, processing up to 8 million packets per second (1.1 Gbps) consuming 23.9 to 29.7 W. Moreover, a single tuple is processed in less than 4 µsec, at least 4 orders of magnitude faster than software.

Chapter 3

Time-SWAD: A Dataflow Engine for Time-based Single Window Stream Aggregation

The number of connected devices grows rapidly along with the amount of data they produce and exchange. Processing such *big data* brings tremendous opportunities in various domains (e.g. financial, transportation) enabling real-time sophisticated decisions that were never possible before. However, realtime analysis of unbounded streams is challenging, it requires *high processing throughput* to cope with massive volumes of data and *low latency* to respond fast.

Streaming aggregation is one of the fundamental and computationally challenging types of stream processing. Streams of values (tuples) are handled in windows of a particular size, WS, which are "slid" by a particular window advance, WA. Then, an aggregation output is produced per window, computed based on a function that uses as input the window values. The result of a Sliding Window streaming AGgregation (SWAG) is a stream of aggregated values. Often values in a stream are grouped by a key, then, values of different keys are processed separately. There are several design options for SWAG, each with different tradeoffs and challenges.

The first choice concerns the *windowing policy* and dictates the way the window is defined. The simplest way is to define the window based on the number of tuples (values) it contains. Tuple-based windows always contain (and are slid by) a fixed number of tuples. They are suitable for applications with fixed data rates and simpler to implement because they have a fixed memory footprint. Alternatively, a sliding window can be defined by a time interval. Time-based SWAG is more commonly used as it is less restrictive allowing varying data-arrival rates which naturally fits the time-series data produced by most Internet-of-Things (IoT) devices [3–5]. Nevertheless, *the number of tuples contained in a time-based window can vary making the memory and compute resources needed to produce the aggregation result unpredictable*.

The second design choice is the type of SWAG algorithm. Some aggregation functions can be computed incrementally using either multiple windows [6] or panes [7, 8]. *Incremental aggregation* computes and stores partial results, rather than storing all the incoming values of a window before computing the full function. In doing so, usually memory pressure is reduced (both capacity and bandwidth) and performance

is improved [9]. However, for some queries, especially with small WA, incremental aggregation has the opposite effect causing an excessive number of memory accesses that limit performance [9], e.g., cases of processing geo-tagged data [5], social media data [10] or manufacturing equipment data [11]. Then, a *single window*, non-incremental approach that explicitly stores all the incoming values in a single window before processing is more efficient [9]. More importantly, storing values in a single window is unavoidable when computing some holistic functions; these are functions that have no constant bound on the size required to store a partial result, such as `median` [12]. In general, non-incremental, single-window SWAG is more suitable for general data mining and machine learning functions [4]. However, *storing all incoming values before processing puts significant pressure to the memory*, which often becomes the bottleneck.

This work introduces the Time-based Single Window stream Aggregation Dataflow engine (Time-SWAD). Time-SWAD addresses the above two challenges of time-based single window stream aggregation and accelerates it using reconfigurable hardware. First, the unbounded number of tuples in a time-based sliding window is facilitated by a flexible circular buffer that stores the window values. We apply the idea of panes [7] to a single window [9] creating a circular buffer that supports bulk evictions. In addition, this buffer can be expanded dynamically with one or more unused identical buffers originally meant for other keys. Thereby, time-based windows of varying size can be stored. Second, the memory pressure of single windows, caused by their need to store all incoming data, is alleviated with a caching scheme. Time-SWAD uses external DRAM in order to support a large number of keys and sufficient volumes of stored values. However, DRAM bandwidth is limited and a caching mechanism is used to merge multiple requests to the same DRAM location. Our design is dataflow, matching well the stream processing characteristics, and is implemented in a Maxeler N-series FPGA card with direct network and DRAM interfaces [25].

Our main contributions are the following:

- The first accelerator for time-based single window stream aggregation.
- A flexible buffer for variable sized time-based windows.
- A novel caching mechanism to reduce the memory pressure of single window aggregation.
- An implementation of the above design that:
  - supports realistic streaming aggregation queries with a large number of concurrently active keys;
  - achieves high processing throughout and ultra low latency due to a deep dataflow pipeline and direct network and DRAM connections;
  - using single windows, allows to handle multiple, arbitrary (and holistic) aggregation functions; and
  - using time-based windows, offers the flexibility to support streams with variable arrival rates.

The remainder of this chapter is organized as follows. Sections 3.1 and 3.2 offer background on time-based streaming aggregation and present related work, respectively. Section 3.3 describes Time-SWAD. Section 3.4 presents the evaluation results. Finally, Section 3.5 summarizes our conclusions.

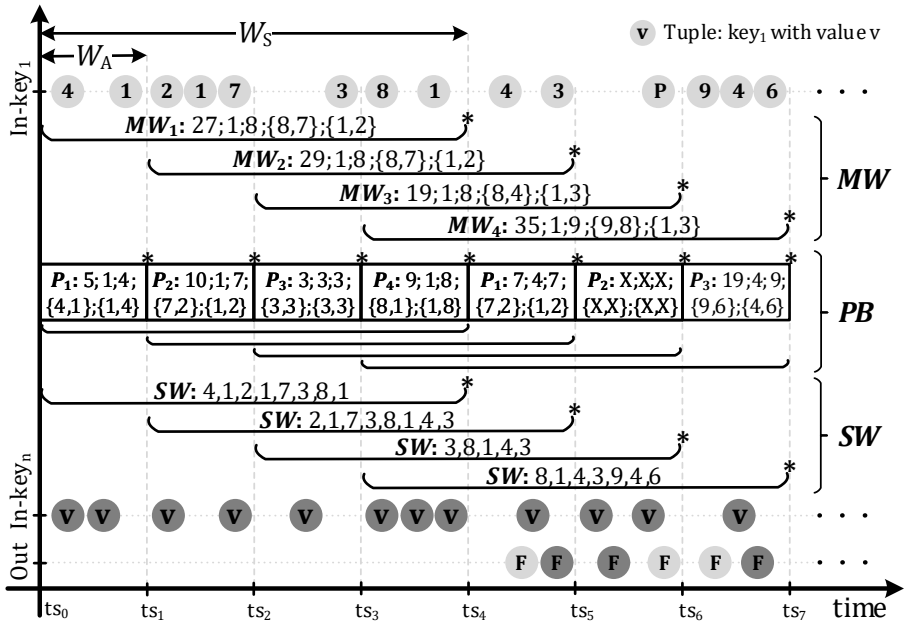


*Figure 3.1: Example of time-based SWAG over a stream of tuples composed by schema, $\langle ts, key, value \rangle$, for parameters $F = \{$`sum`, `min`, `max`, `top-2`, `bottom-2`$\}$, WS = 4 time-units, WA = 1 time-unit, and K = key and internal states/values maintained by the different implementation strategies at time instance indicated by * for In-key$_1$.*

3.1 Background - Time-based SWAG

In this section, we present the semantics of time-based SWAG and provide an overview of the main implementation strategies discussed in literature.

A stream is an unbounded sequence of tuples $t_0, t_1, \ldots$, each tuple containing $n+1$ attributes $\langle ts, A_1, \ldots, A_n \rangle$. Given a tuple t, attribute $t.ts$ represents t's event creation timestamp at the data source and $A_1, \ldots, A_n$ are application-related attributes (values). When aggregated, tuples are fed to one (or multiple) function F such as `sum`, `mean`, `min`, `max`, or `median`. Optionally, aggregation can be performed specifying a parameter K (a subset of the tuple's attributes, also referred to as *key*). In such a case, F is computed for each distinct value observed for K (*group-by*) in the input stream. Punctuation tuples of the form, $\langle ts, key, P \rangle$, where P is a reserved value to indicate that the tuple is a punctuation is used to unblock *group-by* and produce the aggregate for a particular *key* in case there were no tuples for that key during a window slide [44]. Note that the incoming tuples are timestamp-sorted.

Since streams are unbounded, the aggregation is performed over a *sliding window* of *size*, WS time-units and *advance*, WA time-units, both based on the event time in the tuple. Three operations can be performed in SWAG, namely, *insert*, *trigger*, and *eviction*. When a new tuple enters the system, it is *inserted* into the window. If the timestamp of the incoming tuple falls after the current window, aggregation is *triggered*, the oldest invalid entries *evicted* and the new tuple is *inserted*.

In general, three different strategies exist to implement SWAG's semantics, namely, *Multi-Window* (*MW*), *Pane-Based* (*PB*) and *Single-Window* (*SW*) [9]. Figure 3.1

illustrates an example of time-based SWAG showing the different implementation strategies as discussed below.

The *MW* strategy maintains the partial aggregated state of all overlapping windows to which each tuple contributes [6]. As shown in Figure 3.1, the tuple $\langle ts_3, key_1, 8\rangle$ contributes to all 4 open windows, namely, $MW_1, \ldots, MW_4$. The aggregated state is stored in MW_1 until ts_4 time-unit for F = {`sum`, `min`, `max`, `top-2`, `bottom-2`} is $27, 1, 8, \{8,7\}, \{1,2\}$. The tuple $\langle ts_4, key_1, 4\rangle$ triggers aggregation and MW_1 is evicted.

The alternative *PB* strategy [9], partitions the window into P panes, where, $P = \frac{\text{WS}}{\text{GCD(WS,WA)}}$ and maintains partial aggregated states for the latter (`GCD`: greatest common divisor). As shown in Figure 3.1, the window is separated into four 1-time-unit panes $(P_1, \ldots, P_4)$ and the partial state for F is each function's value observed for each pane of 1-time-unit. For example, pane P_4 stores the partial aggregated state, $9, 1, 8, \{8,1\}, \{1,8\}$ from ts_3 to ts_4 time-unit. Upon aggregation trigger, the result is computed by aggregating the pane's partial states (in the example, aggregating F of each pane value), and the stale panes of the window are then evicted. The contribution of the incoming tuple is added to the partial aggregate state of the corresponding pane.

The third strategy, the *SW* [9], maintains tuples, rather than partially aggregated states, in a single window. With *SW*, tuples' values for WS time-units are maintained for each key (as shown in Figure 3.1) and the results are computed on aggregation trigger. For example, until ts_4 time-unit, the SW stores the tuples' values $4, 1, 2, 1, 7, 3, 8, 1$ and on arrival of tuple $\langle ts_4, key_1, 4\rangle$, aggregation is triggered, the single window slides, and evicts the values $4, 1$.

As shown in the past, for large WS, small WA and multiple aggregation functions, *MW* and *PB* implementations would have a lower performance due to their need for higher DRAM bandwidth [9]. Another disadvantage of *MW* and *PB* implementation is that for holistic, non-associative functions maintaining all input tuples is necessary independently of whether the function can be computed incrementally [12]. This is for instance, the case for the `median` function as all the tuples contributing to the window need to be kept and sorted before the median itself can be computed.

3.2 Related Work

In this section, we briefly discuss related works on stream processing as well as on in-memory databases (IMDBs), which are similar, although they follow a store-and-process paradigm rather than on-the-fly processing. Table 3.1 offers a summary of the recent related works, indicating their platform, holistic functions support, window (data) sharing among functions, window size, number of keys, and windowing policy.

Generic software approaches, such as Apache Flink, Spark, and Storm, running on general-purpose CPU offer a wide range of stream processing capabilities, including support for time-based and holistic aggregations with ease of deployment but have limited throughput and high latency [16–18]. Other CPU-based sliding window aggregation algorithms use data structuring and algorithmic techniques such as DABA [2] and MTA [19] and improve latency of in-memory aggregation but are constrained to associative aggregation functions.

GPU systems are able to achieve high processing throughput, but similar to CPU solutions, they fall short in delivering low latency [13] as they have redundant memory accesses managed by an operating system to store incoming tuples in DRAM even before processing starts. This, besides the long latency, wastes a significant fraction

Table 3.1: Comparison of related works focusing on streaming platform and aggregation features.

Platform	Work	Aggregation Features				
		holistic functions	window sharing among functions	window size	number of keys	windowing policy
CPU	Flink [16]	✓	✗	large	large	time+tuple
	MTA [19]	✗	✗	large	single	time+tuple
GPU+CPU	SABER [20]	✗	✗	large	large	time+tuple
	Gasser [4]	✓	✗	large	single	tuple
FPGA	Glacier [6]	✗	✗	small	small	time+tuple
	Oge *et al.* [8]	✗	✗	large	single	time
	Shuntflow [45]	✗	✓	large	single	time
	SW-tuple [9]	✓	✓	large	large	tuple
	Time-SWAD	✓	✓	large	large	time

of valuable memory bandwidth, thereby reducing performance. One of the best systems that uses GPUs is SABER, a relational stream processing system targeting heterogeneous machines equipped with CPUs and GPUs [20]. SABER achieves high throughput but at high latency of hundreds of milliseconds for aggregation queries. Moreover, it supports only incremental aggregate computations utilizing the commutative and associative property of some aggregation functions and therefore can implement only distributive (`count`, `sum`) and algebraic (`average`) functions. Another work that uses GPUs is Gasser [4]. As opposed to SABER, Gasser supports holistic functions at high processing throughput by offloading batches containing, in the order of thousands of windows to the GPU, which negatively impacts the processing latency. Moreover, Gasser supports only a single key and only tuple-based windowing policy. On the contrary, our proposed design is able to support a large number of concurrently active keys for time-based stream aggregation that computes holistic functions while maintaining high throughput and ultra low latency.

On the contrary, FPGAs provide both high processing throughput, similar to that of GPUs, and orders of magnitude lower latency [9]. Customized dataflow engines, which naturally match the stream processing characteristics, can be implemented in reconfigurable hardware offering high throughput. Furthermore, incoming tuples can be fed to an FPGA through a direct network connection with low latency, avoiding unnecessary DRAM accesses. Most existing FPGA-based systems support only incremental aggregations [6, 8, 45]. In Chapter 2, the FPGA-based single window stream aggregation approach handled only tuple-based windows [9]. In that work, dependencies (read after write hazards) between tuples of the same key are resolved with concurrency control queues; this leads to poor performance in case of skewed key distributions which is common in real world data. In contrast, the caching mechanism in this work prevents read after write hazards without limiting performance in skewed key distributions.

A considerable number of previous works have focused on accelerating IMDB operators using FPGAs. One point to note here is that most of the IMDB group-by aggregation designs have some form of optimization for alleviating memory pressure and handling skewed data distributions better [35, 38]. Absalyamov *et al.* [38] performed IMDB *group-by* incremental aggregation (`count`) using CAMs to cache recently produced partial results and improve performance. On the contrary, our caching mechanism stores DRAM lines (of multiple window values per entry) that are partially updated and therefore is different. Furthermore, István *et al.* implemented an on DRAM hash table [36], which was subsequently improved using a Cuckoo hash table [37], similar to our design. They used a FIFO+CAM structure to implement a write through caching mechanism to specifically prevent read after write hazards in the DRAM read-modify-write pipleline caused due to skewed data distributions. Their caching mechanism manages to reduce only the DRAM reads as opposed to our caching scheme which reduces both the read and write accesses for skewed distributions, helping to boost performance. As the read queue allows duplicate addresses in [46], the capacity of the cache is also not efficiently utilized.

3.3 Time-SWAD Design

The proposed design for Time-SWAD is a dataflow engine, implemented on a Maxeler N-series card [25]. Dataflow computing suits the characteristics of stream processing

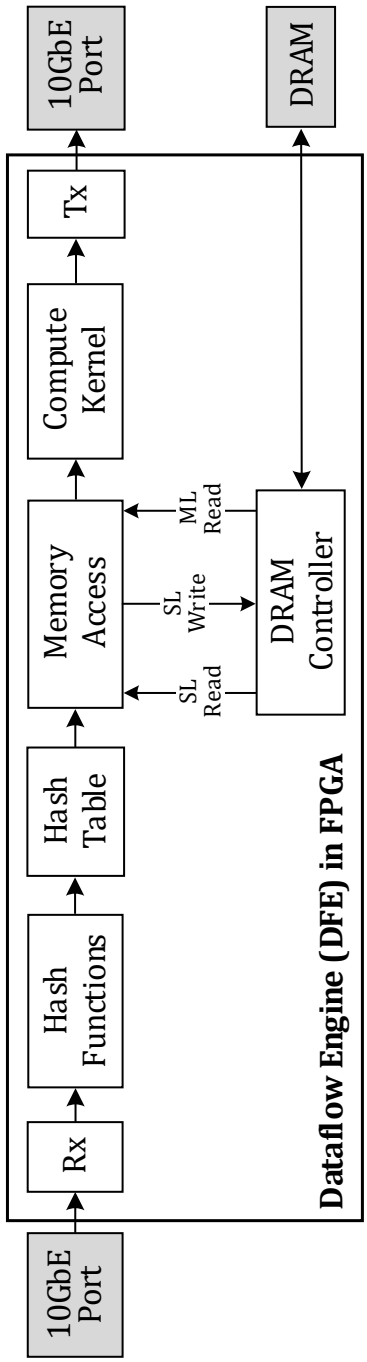


Figure 3.2: Top-level view of Time-SWAD. The grey boxes indicate peripherals outside the FPGA.

well. A deep, feed-forward-only pipeline with lightweight control for a back-pressure stall mechanism is able to process large volumes of incoming data at high throughput. The Maxeler Dataflow Engine (DFE), can facilitate minimum processing latency offering a direct network connection for receiving incoming tuples as well as a direct interface to DRAM for storing the window values.

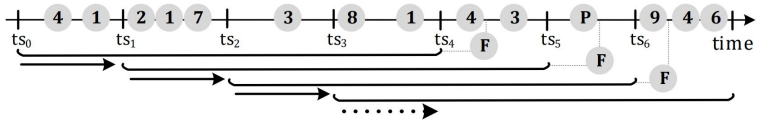
Figure 3.2 shows the top level block diagram of the proposed design. Incoming tuples of the form $\langle ts, key, value \rangle$ are carried by network packets and received by the receiver module (Rx). The key of each tuple is first hashed, as proposed in [37], to the hash table. Multiple hash functions are used for reducing collisions and for adding flexibility. Each hash table entry corresponds to a key and stores metadata needed for processing this key's incoming tuples in the subsequent stages; in particular, mainly for locating the existing window values in DRAM as well as the next available position to write the new incoming value(s). After accessing the hash table, each incoming tuple does a read-modify-write operation to a single line (SL) in DRAM in order to add the new value(s) to the window. A caching mechanism is placed before the DRAM to merge recent accesses to the same DRAM location (line). An aggregation is triggered for a particular key when one of its tuples with timestamp outside the window arrives. Then, its entire single window is read by issuing multiple-line (ML) DRAM commands. Subsequently, this single window feeds the compute kernel where the aggregation function(s) are computed. The result of the aggregation function is finally transmitted back to the network through the Tx module. Note that the flow of the data through the stages is controlled through FIFOs which stall the pipeline via a back-pressure mechanism when needed.

3.3.1 Flexible and expandable circular buffers

The window values of each key are stored in DRAM to offer sufficient capacity for high input rates and large number of keys. DRAM is statically divided equally to the number of keys supported in the system in order to avoid dynamic DRAM allocation, which would be too complex to handle in hardware. However, the amount of values stored in each time-based window is not fixed. This is accommodated in our design by implementing for each key a First-In-First-Out (FIFO) circular buffer in DRAM, which is *flexible* and *expandable*. Below we describe these two attributes of the proposed circular buffer.

The circular buffer needs to support bulk evictions of values every time the window is advancing. In time-based windows the number of evicted values is not fixed. The proposed circular buffer supports bulk evictions of variable number of values by using the pane concept, previously introduced for incremental aggregations (Section 3.1), and applying it to our single-window approach. Briefly, the single window of each key is divided in P panes, where, $P = \frac{\mathrm{WS}}{\mathtt{GCD}(\mathrm{WS},\mathrm{WA})}$. The proposed buffer is defined by a set of read pointers pointing to the beginning of each pane in the window (r_i) as well as by a write pointer w for the head of the buffer (end of last pane).

Figure 3.3 illustrates an example of such a buffer. The input stream is the same as the one in Figure 3.1 for key_1. The columns in the index table correspond to the metadata information needed to be stored in the corresponding key entry in the hash table so as to handle the circular buffer. More precisely, the timestamp ts, pointer to the head of the buffer w, and the starting pointers of each pane, $r_1 \ldots r_4$ in our example because $P = 4$. The pointer management to decide where to *insert* into the single window, what to read upon aggregation *trigger*, and how many values to *evict* in bulk



Hash Table Entry for key *k*

time

In-tuple	ts	w	r_1	r_2	r_3	r_4
Init	0	0	0	0	0	0
<ts_0,k,4>	ts_0	1	0	0	0	0
<ts_0,k,1>	ts_0	2	0	0	0	0
<ts_1,k,2>	ts_0	3	0	2	0	0
<ts_1,k,1>	ts_0	4	0	2	0	0
<ts_1,k,7>	ts_0	5	0	2	0	0
<ts_2,k,3>	ts_0	6	0	2	5	0
<ts_3,k,8>	ts_0	7	0	2	5	6
<ts_3,k,1>	ts_0	8	0	2	5	6
<ts_4,k,4>	ts_1	9	8	2	5	6
<ts_4,k,3>	ts_1	10	8	2	5	6
<ts_5,k,P>	ts_2	10	8	10	5	6
<ts_6,k,9>	ts_3	11	8	10	10	6
<ts_6,k,4>	ts_3	12	8	10	10	6
<ts_6,k,6>	ts_3	13	8	10	10	6

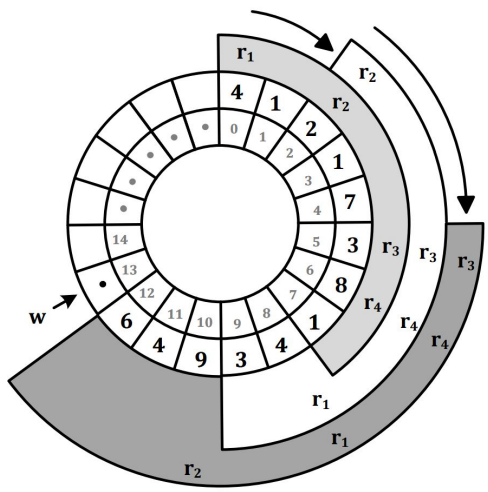


Figure 3.3: Circular buffer management. Insertion, query, and bulk eviction supported by the proposed buffer located in DRAM, which stores a single window for key k. The fields of the hash table entry for k are populated as incoming tuples arrive over time. The grey numbers in the inner circle indicate buffer indices. F-tuple indicates the output aggregate.

from the single window is controlled by this metadata information inside the hash table. The circular buffer in DRAM is just a value store. When the first tuple $\langle ts_0, k, 4\rangle$ enters the system, ts is updated with the window start timestamp ts_0 and the write pointer w is incremented by 1. This tuple contributes to P_1 and so, r_1 points to index 0. Similarly, tuple $\langle ts_1, k, 2\rangle$ contributes to P_2 and so, r_2 is updated to index 2. On arrival of tuple $\langle ts_4, k, 4\rangle$, aggregation is triggered as the event time ts_4 falls outside the window calculated from $ts = ts_0$. This tuple contributes to pane P_1 and the interesting part to note here is that the pane number has wrapped around and this enables us to use the existing read pointer index stored in r_1 as the starting index of the circular buffer for aggregation. So, the circular buffer region of interest for aggregation is starting from $r_1 = 0$ to $w - 1 = 7$. Once the aggregation indices are queried, the single window starting timestamp ts gets updated to ts_1, w to 9, and r_1 to 8 which corresponds to the *insertion* into the sliding window. The *eviction* happens in bulk and comes for free on insertion. Another interesting point to note here is the support for punctuation tuple [44], $\langle ts_5, k, P\rangle$, which is a special tuple used to unblock the aggregation for key k as k did not have any tuple for the time interval, $[ts_5, ts_6)$. This tuple does not increment the indices as it does not contain any valid value.

Although the buffer described so far is flexible, it has limited capacity. Consequently, it cannot accommodate within the window time an amount of incoming values that exceeds its capacity. We address this problem by allowing the buffer to be expanded using the DRAM space of another (or multiple other) inactive, unused keys. This way, we trade the maximum number of keys supported simultaneously in the system for increasing the DRAM space of a single key. Buffers of other inactive keys can be found both within the same set of the hash table (if there is associativity) and across sets by using multiple hash functions. Inactive keys are keys that have not received any tuples for an entire window and therefore do not store any valid value. A key that claimed DRAM buffer space of other keys returns this space when its window shrinks back to its original statically allocated size. In our implementation, each key can expand its buffer space occupying the storage of up to one more key. The victim key is found in another entry (set) of the hash table using a second hash function.

3.3.2 Hash Table

The hash table stores metadata for the active keys handled in the system. Depending on the SWAG parameters (WS, WA, number of keys), the hash table may have very long entries and/or very large number of entries. In order to allow our design to operate at high frequency, the hash table is organised in a tiled fashion as shown in Figure 3.4. In doing so, the implementation tools can map it more efficiently to the BRAM resources. Hash table tiles are organised in a 2D structure, with r rows and $c+1$ columns configured at build time, each column being a different pipeline stage. By default, a tile T has two ports, width T_w of 80 bits and depth T_d of 1024 entries, hence, each tile consumes 4 M20K BRAMs (80 kb).

The cell in the first tile column is special and it stores the following metadata: (i) valid bit: boolean to indicate whether the hash table entry (row) is valid; (ii) window full bit: boolean to indicate whether the SW circular buffer is full; (iii) key (24 b) which is hashed to this row for identification; (iv) window start timestamp (24 b) to determine when the key is ready for aggregation; (v) write pointer (20 b), pointing to the next DRAM index in the key's single window where to *insert*. The remaining fields in a hash table entry that span across the remaining tile columns are the pointers

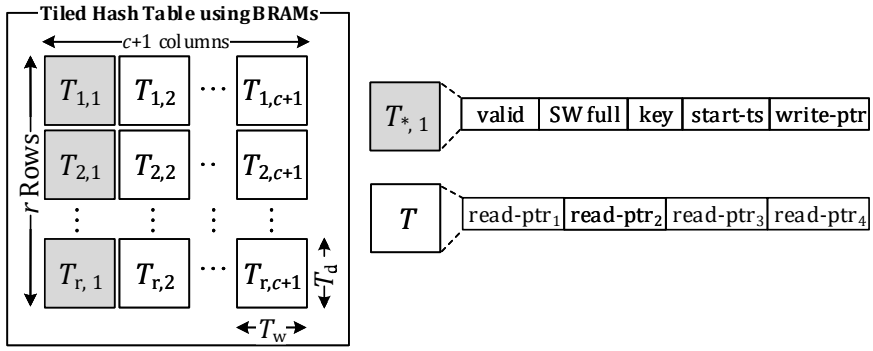


Figure 3.4: Tiled organisation of the hash table with the corresponding fields.

to the beginning of each pane r_i as described in Section 3.3.1.

Before accessing the hash table and in parallel to computing the hash functions for generating the hash table address, the following information is also generated: (i) the pane number to which the current tuple contributes, $P_n = \lfloor \frac{ts \bmod \mathrm{WS}}{\mathrm{GCD(WS,WA)}} \rfloor$, where ts is the timestamp in the current tuple; (ii) the hash table tile column number corresponding to the above pane, given by, $1 + \lfloor \frac{P_n}{N} \rfloor$, where N is the number of pane read indices present in each tile entry of width T_w; (iii) the pane read pointer slot in the above hash table tile entry, given by, $P_n \bmod N$; and (iv) the starting pane number of the single window, $P_\mathrm{W} = (P_n + 1) \bmod P$ and the corresponding tile column number and slot in the tile entry as above, used to retrieve the key's window-start-index to check if it is full.

Then, the hash table is accessed in a pipelined fashion as follows. First, a tile in column 0 is read using the hash table addresses obtained from previous stage. Here it is checked if the requested key exists in either of the entries identified by the hash functions (or in multiple entries if the key has expanded to multiple buffers). A hit in the key determines the subsequent row of the remaining table to be accessed. In case of missing the key in the hash table, a new entry is made, evicting the oldest key of the accessed entries based on their timestamp. If the evicted key is still active a flag is raised indicating collision and the information is sent to software. In a complete system, a software process could handle keys that do not fit in hardware due to collisions or capacity issues. Accessing the first column of the hash table determines whether an aggregation is triggered or the incoming value is just added to the window and provides the address and type of the subsequent memory access. In addition, the values of the circular buffer pointers are updated and written back. Note that in case the buffer space allocated for the key is exhausted, then the buffer is expanded if possible following the above eviction process. In case of collision or reaching the maximum buffer space, a flag is raised again, providing the respective information to software. Finally, unused buffers previously occupied by an already expanded key are detected and released.

3.3.3 Memory subsystem

There are two types of memory accesses in our system. The first one updates the circular buffer that stores the window of a key adding an incoming value. This requires a read-modify-write on a single DRAM line. The second type of memory accesses

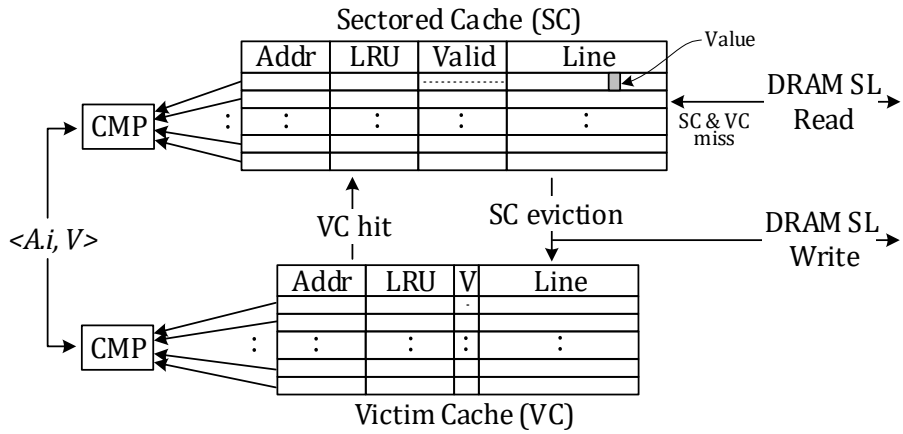


Figure 3.5: Caching mechanism before DRAM single line (SL) read-modify-write.

takes place when an aggregation is triggered. Then, the entire window of a key is read and used to compute the aggregation function(s). This second case requires one or multiple multiple-line DRAM read commands. We discuss both types of memory accesses below in detail below.

3.3.3.1 DRAM Single-Line (SL) Read-Modify-Write

DRAM SL read-modify-write requests are pipelined to improve throughput, but may also cause read after write hazards. As a consequence, our design needs to ensure that a write caused by an earlier tuple has been committed to the DRAM before a read of a subsequent tuple to the same address. Another reason for the read after write hazard in our DFE is due to the fact that there are separate queues to the DRAM controller for DRAM reads and writes. The DRAM controller does not provide guarantees on the ordering between commands in different queues.

We address this problem by adding a caching mechanism before the DRAM. The same mechanism also improves performance reducing DRAM accesses in skewed distributions of keys which is common in real world data sets. In such cases, the same key produces bursts of tuples (temporal locality). Then, adding a cache merges consecutive DRAM accesses to the same address caused by such bursts.

The proposed caching scheme illustrated in Figure 3.5 needs to support the following two points. First, incoming values need to wait in the cache for the DRAM line to arrive before they can be added to it; consider that read-modify-write to the cache can be handled in a single cycle. Second, the updated DRAM line need to be kept in (another victim) cache to be available for reuse until it is confirmed that it has been written to DRAM. We use two almost identical cache blocks to support each of above points. They are both fully associative, their lines are equal to a DRAM line (64 B that fit 32 values), and have 32 entries in order to cover for the maximum DRAM (read or write) access latency which is 32 cycles in our system. The replacement policy is Least Recently Used (LRU) in order to ensure that a cache line stays in the cache for at least 32 cycles since its last use before it gets evicted. The first cache needs to be sectored in order to keep track of the validity of each value in the line separately; in other words it needs a separate valid bit for each of the 32 values in a cache line. This

is the only difference between the two cache blocks henceforth called Sectored Cache (SC) and Victim Cache (VC).

The proposed caching mechanism works as follows. Consider an incoming value V to be added on address $A.i$, where A is the DRAM line address and i is the offset of the value in the line. In case address A misses in both the SC and the VC, a new entry for A is added in the SC, V is added in the offset i (the rest of the line is invalid), and a DRAM read request is sent for address A. In case A hits on either of the two caches, it updates the value of the line at offset i; if it was a hit in the VC, that VC entry is removed and written to SC. Finally, a line evicted from SC is written back to DRAM and also in the VC; a line evicted from VC is not written back.

3.3.3.2 DRAM Multiple-Line (ML) Read

When an aggregation is triggered, the entire window of a key needs to be read and send to the compute kernel. This is performed with one or multiple multiple-line read requests to DRAM. In our system, the memory controller allows reading up to 128 DRAM lines in one command. For example, to read a single window of 32 kB, 4 read commands need to be issued. In case a requested line hits in the cache, it overwrites the one read from DRAM. It is worth noting that the order of the read lines per key is maintained in order to support non-commutative aggregation functions like `rank` which are order sensitive.

3.4 Evaluation

3.4.1 Experimental Setup

We evaluate Time-SWAD on a Maxeler N-series ISCA (MAX4AB24B) PCIe card with an Altera Stratix V (5SGXAB). This card provides a 10 Gb/s direct network connection to the FPGA and is equipped with 3× 8 GB DDR3 off-chip DRAM DIMMs. The memory is accessible directly by the FPGA via three independent channels and has a maximum bandwidth of 38.4 GB/s. The FPGA has also 6.6 MB of on-chip memory (BRAM). Our design is implemented in MaxJ, a Java based High level Synthesis (HLS) language and compiled to a bitstream using MaxCompiler.

Two standard real-world datasets are used as the input streaming data: the Google compute cluster monitoring (CM) [47] and smart grid (SG) [48]. The tuples fed into the system are composed of three fields, a timestamp (24 bits), key (24 bits) and value (16 bits). These are the only required fields for the queries. Truncation of the fields is performed to match the tuple size to the network interface and does not affect the contents of the fields. The *tcpreplay* tool [42] is used to inject the captured packets at varying injection rates to determine the highest sustainable throughput of the system. The queries used are the following:

CM_1: Find the aggregation functions, F of the CPU usage for each *category* for the last 60 seconds (WS = 60), returning a result every WA seconds. CM_1 represents queries with aggregation on a small number of keys (4 *categories*), exploiting the benefits of the caching system.

CM_2: Find the aggregation functions, F of the CPU usage for each *jobID* for the last 60 seconds (WS = 60), returning a result every WA seconds. CM_2 represents

aggregation queries on a large number of *jobIDs*, exploiting the support for large number of concurrently active keys.

SG: Find the aggregation functions, F of the power usage for each *house* for the last 3600 seconds (WS = 3600), returning a result every WA seconds. SG represents aggregation queries on a large window size.

For each query, WA is varied from 1 to WS time-units. The aggregation functions F include both associative and non-associative functions such as `average`, `minimum`, `maximum` and `median`. Histogram-based median filtering is implemented for computing the `median` function [49].

Besides throughput and latency, power consumption for Time-SWAD was measured using the available tool to measure the total power consumed by the ISCA board including the FPGA, DRAM and QSFP port.

3.4.2 Resource Utilization and DFE Frequency

For the implementation of Time-SWAD, 132779 (37.0%) logic (ALMs) and 2195 (83.1%) BRAMs of the available FPGA resources are used. The logic resource utilization is mainly constituted of hash table stage (20%), caching system (25%), and the compute kernel (40%) implementing the aggregation functions. Based on the number of BRAMs that can be allocated in the platform for building the hash table, the tiling mechanism as described in Section 3.3.2, is used to maximise the number of keys supported for a given number of panes. For example, in our FPGA platform, we allocate a budget of 1500 M20K BRAMs for the hash table which is around 68% of the total BRAMs utilised. The key cardinalities supported for this BRAM budget range from 9 to 18 bits as the number of panes vary from 3600 to 1, respectively. The remaining BRAMs are mostly utilized by the queues between the pipeline stages. This BRAM budget was chosen for the hash table in order to keep the total BRAM utilization around 80% making timing closure for the desired frequency easier. As the resources occupied by the hash table, caching system and the compute kernel are the same for the aggregation queries, the resource utilization is similar across queries. The DFE operates at 156.25 MHz enabling to achieve a maximum of 10 Gb/s (156.25 million tuples per second) line rate through the direct 64 bit network interface connected to the FPGA.

3.4.3 Throughput and Latency

The chart in Figure 3.6 depicts the throughput and latency for the three queries (CM_1, CM_2 and SG) for Time-SWAD. For each query we show the results for different values of WA. Throughput represents the number of tuples processed by the system performing the queries on the input stream per unit of time. Latency represents the average time it takes from the moment a tuple enters the system until it is processed.

The results in Figure 3.6 show that for all queries the throughput increases as the WA values increase. This is a consequence of having fewer calls to the compute kernel as WA increases, which in turn result in fewer ML DRAM accesses and thus a reduction of the overall waiting time.

The impact of the queries' characteristics on the throughput is also clear from the results. The higher throughput is achieved for the query CM_1 that imposes less pressure on the memory sub-system, since the aggregation function is on a smaller set of keys and also the window size is relatively small. As there are only 4 concurrently

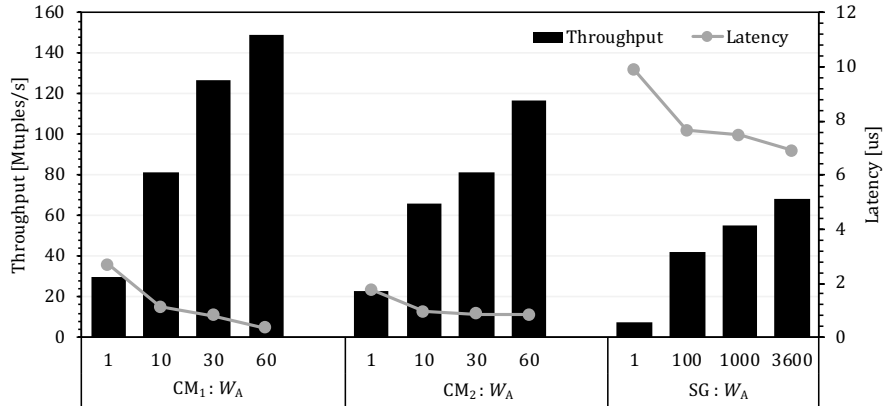


Figure 3.6: Throughput and latency for Time-SWAD.

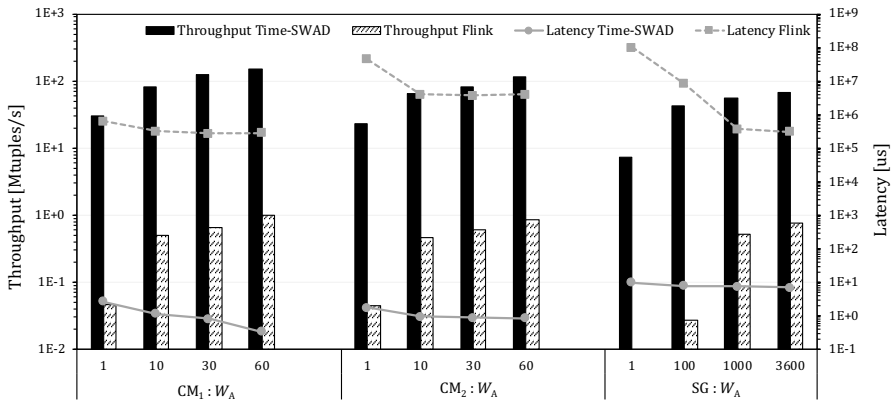


Figure 3.7: Throughput and latency: Time-SWAD vs Flink.

active keys in this workload, the DRAM SL reads and writes are largely reduced due to the caching layer. For CM_2, we observe that the throughput is reduced and this can be attributed to the fact that in this query the aggregation function is performed on a larger number of keys. Throughput is further reduced for SG, which has large window size, thus resulting in a large number of DRAM ML reads when data is collected by the compute kernel.

When compared to the throughput supported by the network interface (*line rate*), the throughput for the queries ranges from 5% (SG: WS = 3600; WA = 1) to 100% (CM_1: WS = 60; WA = 60) of the maximum throughput.

Latency, as expected, follows the opposite trend to the throughput, showing lower values as WA increases. The latency for CM_1 ranges from 0.3 to 2.68 µs. The latency for CM_2 ranges from 0.83 to 1.75 µs. The latency for SG ranges from 6.89 to 9.88 µs. The highest latency is for SG due to the DRAM pressure caused by the large window size.

3.4.4 Comparison with alternative systems

As mentioned before, Time-SWAD is the first hardware accelerator to perform time-based stream aggregation for non-associative functions. As such, a direct comparison with our system can only be done with a software-based system such as Flink [16]. Flink (Apache Flink v1.5.1) is evaluated on a workstation with an Intel® Core™ i7-4790 CPU running at 3.6 GHz and 24 GB DDR3 DRAM. The non-incremental API provided by Flink is used to implement the non-associative aggregation queries. The power consumption for this system is measured using the processor performance counters for the package power to which we added the DRAM power. The throughput and latency for both Time-SWAD and Flink are depicted in Figure 3.7.

It can be observed that the trends for both throughput and latency are very similar for both systems. The major difference is the large gap between the absolute values. Notice that both the throughput and latency scales are represented in logarithmic scale. The throughput achieved by Time-SWAD is 1 to 2 orders of magnitude higher and the latency is 4 to 7 orders of magnitude lower than Flink.

In terms of power consumption, depending on the query, the FPGA board for the Time-SWAD consumes between 24.9 W to 27.1 W while the CPU for the Flink system consumes between 30.1 W to 48.2 W. More important than power consumption is the energy efficiency, measured in throughput per watt, where Time-SWAD is 2 to 3 orders of magnitude more energy efficient than Flink.

All previously proposed hardware-based systems that perform time-based stream aggregation are limited to using associative functions as they do not support non-incremental aggregation [20]. In order to present a comparison with alternative state-of-the-art hardware-based systems, we have tested our system with the same queries as before but excluding the non-associative function `median` and compared it with the results reported for SABER [20], which is the fastest among previous works. The results are depicted in Figure 3.8. Note that since Time-SWAD is focused on non-associative functions, the implementation of it's aggregation functions is non-incremental. Nevertheless, associative functions can be implemented more effectively if implemented in an incremental way. Therefore, we also present results of an incremental implementation of Time-SWAD, called Time-SWAD-Inc, simplifying the original design presented in Section 3.3. This is achieved by using the hash table to store partial results and disabling all subsequent memory access and compute stages; this is possible because the incremental values for the functions can be calculated in the hash insert stage and their values can be added to the hash entries themselves.

The results in Figure 3.8 show that the non-incremental implementation of the aggregation achieves limited throughput. In contrast, the emulated incremental implementation for our system achieves the maximum possible throughput for the tested network interface. This throughput is even larger than what is achieved by SABER. The difference is small for both CM_1 and CM_2 but quite large for SG_2. We attribute this larger difference from the fact that SG_2 produces the aggregation value over a very large window size and that for SABER the execution of the aggregation function is offloaded to a GPU, requiring large amount of data to be transferred from the main system to the memory of the GPU.

In terms of latency, SABER reports a sub-second processing latency. In contrast, Time-SWAD has a latency that is less then a few tens of a microsecond, which is about 5 orders of magnitude shorter than SABER.

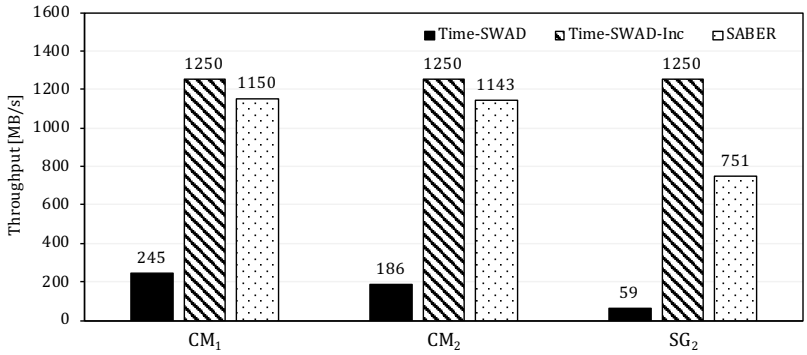


Figure 3.8: Throughput for associative aggregation functions: Time-SWAD vs SABER. CM_1, CM_2 and SG_2 represent queries taken from [20].

3.5 Conclusion

This chapter describes Time-SWAD, the first FPGA-based time-based single window streaming aggregation engine. The proposed flexible and expandable circular buffers are able to cope with the variable size of time-based windows. Time-SWAD achieves high throughput, up to 150 Mtuples/s (10 Gb/s) by employing dataflow processing and a novel caching mechanism which alleviates memory pressure. Time-SWAD matches the throughput of related GPU systems, which however do not offer both time-based and single window aggregation. In addition, the proposed design exploits the direct network and DRAM connections of the employed platform to offer ultra low latency of 1-10 µs, which is at least 4 orders of magnitude lower than CPU and GPU solutions. In summary, our approach is able to support realistic streaming group-by aggregation queries, with multiple holistic and arbitrary user-defined aggregation functions, large number of concurrently active keys, large window sizes and small window advances which trigger frequent aggregations.

Chapter 4

A Specialized Memory Hierarchy for Stream Aggregation

The rapidly increasing rates at which data are produced globally have enabled a large number of emerging stream processing applications [50]. Such applications are employed in various domains, e.g., financial, transportation, to analyze large unbounded streams of data and make fast, sophisticated decisions. However, consuming massive data volumes at line rates requires *high processing throughput* and in case real-time response is expected, it also needs *low latency*.

Stream aggregation is one of the most challenging tasks in stream processing. It can be described by applying the traditional relational database aggregation semantics to a sliding window. Such 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 stream of aggregated values is 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 a 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 window policy is the time-based one where the size and slide are defined by time intervals [1]. A typical tuple-based SWAG example is depicted in Figure 4.4(a).

For some problems, the aggregations can be simplified by computing them incrementally using multiple windows/panes [6–8]. However, for many others with *non-associative* aggregation functions which cannot be computed incrementally, e.g., `median` [12], or even if they can, it is more expensive than explicitly storing all incoming values in a *single window*, e.g., for processing geo-tagged data [5], social media data [10] or manufacturing equipment data [11].

Single window stream aggregation is a memory-intensive problem [13]. For each incoming tuple, the sliding window of the respective key is updated with the newly arrived value(s). In addition, every time the window advances, its entire

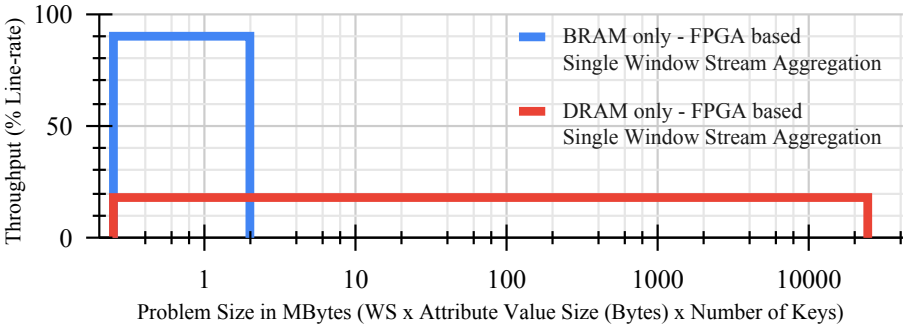


Figure 4.1: Processing throughput vs. problem size for an FPGA-based single window SWAG at 156.25 MHz using only BRAMs (2 MB) or only DRAM (24 GB). WS= 2-96K values; WA=WS; 128K keys, value size 2 bytes.

contents need to be read and fed to the aggregation function(s) to produce a result. Different stream aggregation platforms handle memory in different ways. Multicore CPU and GPU based systems, although able to sustain high processing throughput [4, 20], have wasteful memory management [13, 51]. They require redundant memory accesses to store incoming tuples from the network to DRAM even before processing starts. This, besides the latency overhead, wastes valuable memory bandwidth and hence limits performance. On the contrary, FPGAs use their memory resources more efficiently [9, 26]. They can offer a direct network connection to receive incoming tuples and support dataflow processing, delivering both high processing throughput and low latency. However, existing FPGA approaches use either only on-chip memory (i.e. BRAMs) [6–8] or only off-chip memory (i.e. DRAM) to store the values for aggregation [9, 26]. As illustrated in Figure 4.1 for a particular stream aggregation problem, the higher bandwidth and limited capacity of on-chip BRAM enables line-rate processing on an FPGA based stream aggregation system but supports only small problem sizes. The larger but lower bandwidth off-chip DRAM can handle larger problems, but with limited performance.

This chapter introduces Multi-level Queues (MLQ), the first memory hierarchy specialized for stream aggregation systems aiming to alleviate their memory bottleneck. The proposed memory system offers a higher and better utilized bandwidth as well as off-chip DRAM capacity to enable higher processing throughput for larger problem sizes, i.e., WS $\times$ number of keys. As shown in Figure 4.2, multiple memory levels are used to form logical queue buffers, each buffer storing the contents of a sliding window for a particular key. Each multi-level logical queue needs to support (i) *single element write* and (ii) *all elements read* operations for window updates and window aggregations, respectively. More precisely, for a window update, a new value needs to be enqueued. The head of the MLQ can be at any memory level, but the tail is always at the fastest (and smaller) first level which is the on-chip SRAM. This ensures that the window is always updated at the highest speed. Then, when the window advances, the contents of the entire window are read utilizing the aggregate bandwidth of all memory levels and subsequently, WA number of elements are discarded by just updating the head pointer. Compared to a BRAM-only design, MLQ offers higher capacity. Compared to a DRAM-only design, it offers faster window updates at on-chip SRAM speed as well as faster aggregation using the aggregate bandwidth of all

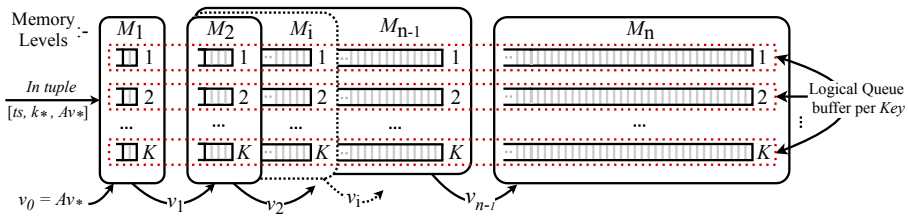


Figure 4.2: Memory hierarchical model with n levels with each level having K queues (one per key) and v_i denoting the flushed values from level i. Incoming tuple of key k_* is at timestamp ts with attribute value Av_*.

levels, rather than only the DRAM one.

Another alternative for improving the memory bandwidth of a SWAG could be the use of high bandwidth 3D-stacked DRAM. However, even 3D-stacked DRAM bandwidth is at least an order of magnitude lower than on-chip SRAM, while its size is significantly smaller than off-chip DRAM. In addition, 3D-stacked DRAMs are expensive, especially for systems deployed near the edge. Nevertheless, they could be part of the proposed hierarchy.

Queue buffers composed of two memory types have been designed in the past. More precisely, about two decades ago, 2-level hybrid SRAM/DRAM packet buffers were introduced for network processing [27–31] offering SRAM speed and DRAM capacity. Although our approach is in the same direction, there are several fundamental differences. Firstly, the SRAM/DRAM packet buffers implement queues that support read and write operations at the granularity of a single element and this is less bandwidth demanding compared to the stream aggregation. In addition, a network packet size is at least equal to a DRAM line (64Bytes) and therefore fits DRAM better than the stream aggregation accesses which are often finer and hence require expensive read-modify-write operations. Finally, the hybrid packet buffers were limited to two levels, while the proposed memory system can use more levels to exploit a higher aggregate bandwidth.

The contributions of this work are the following:

- MLQ, a specialized memory hierarchy for stream aggregation;
- An analytical model of MLQ and a method to automatically generate its configuration for a problem at hand; and
- An MLQ implementation, in a FPGA-based SWAG system and its evaluation and comparison with related work.

The remainder of this chapter is organized as follows. Section 4.1 offers an analytical model of MLQ. Section 4.2 describes our FPGA-based stream aggregation design with MLQs. Section 4.3 presents the evaluation and comparison with related work. Finally, Section 4.4 summarizes our conclusions.

4.1 MLQ Analytical Model

In this section, a generic description of MLQs and an analytical model for estimating system's performance are presented. This model is then used by a heuristic to identify

an efficient configuration of the memory hierarchy, i.e., selecting block size per level, for the SWAG problem at hand considering the given memory characteristics while aiming to maximize the processing throughput of the system.

We consider n levels in the memory hierarchy, $M_1, \ldots, M_n$ as shown in Figure 4.2.

The higher the level the longer the access time and the larger its capacity, which is shared equally between the keys (K) supported in the system. Let C_i denote the capacity of the i^{th} memory level and $Ck_i = C_i/K$, denote the capacity available per key in each level. The access granularity of a memory level is defined as the minimum number of bytes that can be read or written (R/W) in a memory access. Let Gw_i and Gr_i denote the write and read access granularity of the i^{th} level, respectively. For example, on-chip SRAM can be configured to support R/W access granularity of just a single bit, while DRAM has a R/W access granularity is 64B lines. The ideal write access granularity (en-queue) required for window updates is equal to size of the attribute value(s) (As) carried by a tuple. In case the write access granularity is larger than that, e.g. values of 4B written directly to DRAM, then a more expensive read-modify-write operation is needed rather than a simple write.

The average R/W memory access time T_i of level i is measured in number of cycles of the processing chip (i.e., FPGA). For simplicity it is considered that all memory ports are R/W with the same access time for both access types. T_i is the inverse of the average access rate (throughput) of a level. The number of available channels offering for independent parallel R/W access in level i is denoted as Ch_i.

We define the input system throughput TPC_{in}, i.e., the line rate at which the system receives incoming tuples. Considering that one tuple is received per cycle, $TPC_{in} = 1$ tuple per cycle.

When a tuple enters the system, its value gets enqueued to the tail of the corresponding key's MLQ in the fastest first level of the memory hierarchy M_1. When a level gets full, the complete block of values in that level is subsequently flushed to the next level. Let F_i denote the flush rate of the i^{th} memory level, which is the inverse of the block of values stored in that level. v_i denotes the number of values stored in the i^{th} level per key before it is flushed to the next level. The incoming tuple's attribute value inserted in M_1 is denoted as $v_0 = 1$. So, $F_i = 1/v_i$, for $1 \leq i \leq n-1$. It is worth noting that the n^{th} level should be able to hold the entire window ($v_n = \mathrm{WS}$).

Based on the above, we model the number of memory accesses per incoming tuple required for window updates and aggregation and then the throughput sustained by each memory level. For simplicity, a uniform random distribution of tuple arrival per key is considered, but different arrival rates would follow the same methodology.

4.1.1 Window Updates

There are three types of memory accesses that may occur during window updates. First, a write access of the incoming value(s) of size As to M_1. Second, a read access to any memory level that is full and needs to be flushed. Third, a write access to level i to store the flushed values from level $i-1$. Let Wu_i and Ru_i be the average number of write and read accesses per tuple, respectively, due to window updates to the i^{th} memory level. Every incoming tuple has to be written to M_1, so, $Wu_1 = 1$. For the successive levels, the number of writes in level i depends on the flushing rate per tuple of the previous level F_{i-1} and on the number of write accesses needed to write the v_{i-1} flushed values which is $\lceil \frac{v_{i-1} \times As}{Gw_i} \rceil$. In case a read-modify-write (rmw) operation is needed, an equal amount of read accesses needs to be accounted: $R_{rmw_i} = Wu_i$, if

$(v_{i-1} \times As) < Gw_i$; considering values will be aligned and do not span across two M_i lines. Note, that there is the option to underutilize the capacity of memory to avoid read-modify-writes. In the event of a flush, the number of read accesses on level i is $\lceil \frac{v_i \times As}{Gr_i} \rceil$ and on average, these reads happen at a rate of F_i per tuple. Then, the number of write and read accesses for window updates at each level are:

$$Wu_i = \begin{cases} 1, & \text{for } i = 1 \\ F_{i-1} \times \lceil \frac{v_{i-1} \times As}{Gw_i} \rceil, & \text{for } 2 \leq i \leq n \end{cases}$$

$$Ru_i = \begin{cases} F_i \times \lceil \frac{v_i \times As}{Gr_i} \rceil + R_{rmw_i}, & \text{for } 1 \leq i \leq n-1, n > 1 \\ R_{rmw_i}, & \text{for } i = n \end{cases}$$

4.1.2 Aggregation

Upon aggregation, the entire window of a key is read, which may spread across all MLQ levels. The number of read accesses in level i for one aggregation is $\lceil \frac{v_i \times As}{Gr_i} \rceil$. Since these reads happen once every WA arriving tuples, then the average number of read accesses per tuple is $Ra_i = \frac{1}{\text{WA}} \times \lceil \frac{v_i \times As}{Gr_i} \rceil$. This is the worst case Ra_i as it assumes the entire key space in the level (v_i) is needed.

4.1.3 Average Throughput

The average throughput sustained by level i is measured in tuples per cycle, TPC_i. This is the inverse of the average number of cycles per tuple CPT_i required to complete the accesses per tuple in the level. The CPT_i required per tuple for window update writes and reads as well as for aggregation reads is $[(Wu_i + Ru_i + Ra_i)/Ch_i] \times T_i$, where T_i is the access time (in cycles) and Ch_i the number of parallel channels at the level. Thus, the tuples per cycle for level i is $TPC_i = 1/CPT_i$.

The overall system throughput TPC_{all} is the minimum between the input throughput (TPC_{in}) and the throughput of each memory level. The average cycles per tuple consumed per level are not summed as all memory levels are working in parallel to perform the accesses for window updates and aggregation in a dataflow fashion. So, the throughput supported by the system $TPC_{all} = min(TPC_{in}, TPC_1, \ldots, TPC_n)$. As we focus on the memory system, we consider that the aggregated values delivered by the memory system can be consumed at the same rate by the subsequent stage which computes the aggregation functions, otherwise the throughput of the compute stage needs to be considered in the above equation.

4.1.4 Automatic memory configuration generation

Based on the above performance modeling, we seek the optimal memory configuration, that is the number of values stored per level $V = \{v_1, \ldots v_n\}$ based on the aggregation parameters (number of keys, WS, WA) and memory characteristics in order to maximize the processing throughput of the entire system and meet the memory capacity constraints.

We developed a heuristic, described in Algorithm 1, to find the partition set V that maximizes system throughput. Input i encapsulates the memory characteristics of level i such as capacity(Ck), access-granularity (G), and access-time (T). Starting from level-1, the heuristic finds for each level i, the set of solutions for v_i that can

ALGORITHM 1: Partitioning algorithm.

```
   Input: i, isBackward                          // memory level, back-propagation flag
   Output: V, TPC_all
1  Function Partition(i, isBackward):
2      if isBackward then
3          if i == 1 then
4              update v_1, TPC_1, TPC_in
5              Partition(i + 1, false)
6          else
7              update v_i, TPC_i based on v_{i+1}, TPC_{i+1}
8              Partition(i − 1, true)
9          end
10     else
11         if hasVisited_i then
12             Partition(i + 1, false)
13         else
14             find v_i set to maximise TPC_i within available Ck_i
15             if i == n and v_n < WS then
16                 return Not enough capacity in MLQ
17             end
18             if TPC_i == TPC_{i−1} and v_i == WS then
19                 print V, TPC_all
20                 return                                  // partitioning done
21             end
22             if TPC_i < TPC_{i−1} then
23                 Partition(i − 1, true)
24             else
25                 hasVisited_i = true                  // visit flag - level i
26                 Partition(i + 1, false)
27             end
28         end
29     end
```

support its requested input throughput TPC_{i-1} within the available capacity Ck_i (line 13) using the formulas of our analytical model. In case there is a set of v_i solutions that can support 100% of TPC_{i-1}, then the output throughput of the level TPC_i is calculated and given to the level $i+1$ as input (line 23). Otherwise, the heuristic would call the function for the previous $i-1$ level (line 20) to reconsider its solutions for the newly adjusted TPC_{i-1} (lines 2-8). The heuristic exits when it finds a solution for the last memory level that satisfies its given input throughput and capacity constraint (lines 16-18). The output of the heuristic is V and the (possibly adjusted) TPC_{all} that is satisfied by V, which is the maximum input throughput TPC_{in} that can be processed by the system. Compared to exhaustive search which would need to search the entire WS space for each memory level ($O(\mathrm{WS}^n)$), our heuristic has substantially lower complexity of $O(n^2)$ as in the worst case, each recursive call has to traverse and fit only from the current level to level-1.

4.2 Reconfigurable Single Window SWAG with MLQ

A reconfigurable single window SWAG dataflow engine (DFE) that uses the proposed MLQ memory hierarchy is designed to exploit the offered high bandwidth and capacity. The top level block diagram of our engine is shown in Figure 4.3. Incoming tuples 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 for reducing collisions and for adding flexibility [37, 52]. Each hash table entry corresponds to a key and stores metadata needed for processing this key's incoming tuples in the subsequent stages; in particular, for managing the memory accesses for window updates and aggregation at each memory level. After accessing the hash table, the memory commands are generated to each level based on the metadata state. The data collector acts as a buffer to synchronise the dataflow of the single window from the various memory levels. Subsequently, aggregated values of a window fetched from the memory are fed to the compute kernel(s) where the aggregation function(s) are computed. The result of the aggregation function is finally transmitted back to the network through the Tx module. Note that the flow of the data through the stages is controlled through FIFOs which stall the pipeline via a back-pressure mechanism when needed. The design is implemented in a platform that offers three memory types to be used in MLQ, in particular besides the FPGA on-chip BRAMs, an off-chip DRAM, and off-chip SRAM are used. Table 4.1 shows their characteristics.

From the end users' perspective, based on the stream aggregation parameters provided and the specifications of the memories, the heuristic discussed in Section 4.1 generates the partitioning of the design-time reconfigurable memory hierarchy that maximizes throughput. The APIs provided by MLQ are: a) *insert* (en-queue) a single value upon tuple arrival; b) *flush* a block of values from a full memory level to the next; c) *aggregate* by reading the complete single window queue and d) bulk *evict* values upon window slide corresponding to the WA from the memory levels. These APIs translate to one or more read or write access micro-commands.

4.2.1 Hash Table

The hash table stores the metadata for the active keys in the system and is implemented using BRAMs. A hash table entry points to the MLQ space allocated for storing the window of a single key. The metadata contains: (i) a valid bit; (ii) the key assigned to the entry; (iii) window start timestamp for key replacement in case of collision and to trigger aggregation in time-based windows; (iv) read pointers to each memory level, r_i pointing to the head index in level i; and (v) write pointers to each memory level, w_i pointing to the tail index of each memory level i.

The hash table is accessed in a pipelined fashion using multiple hash functions. 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. If the evicted key is still active, a flag is raised indicating collision and the information is sent to software.The hash table can be extended using the memory hierarchy and/or a software process could handle keys that do not fit in hardware due to collisions or capacity issues. A hit in the hash table fetches the metadata of the associated key which determine: i) the index in M_1 to *insert* the incoming tuple's value for window update; ii) whether any memory is full

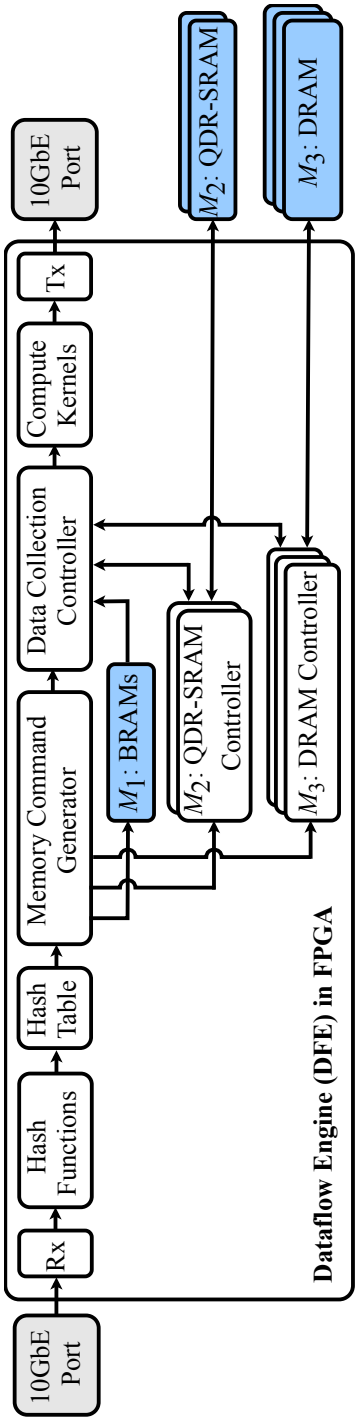
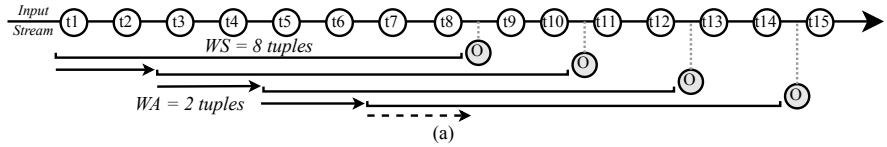


Figure 4.3: Top-level view of a Reconfigurable single window SWAG with MLQ.



(a)

Tuple	r_1	w_1	r_2	w_2	r_3	w_3	Operation	M_1	M_2	M_3
Init	0	0	0	0	0	0	-	0 1	0 1 2 3	0 1 2 3 4 5 6 7
1	0	1	0	0	0	0	M_1-*Wu* (*Insert* t1 in M_1)	t1 -	- - - -	- - - - - - - -
2	0	2	0	0	0	0	M_1-*Wu* (*Insert* t2 in M_1)	t1 t2	- - - -	- - - - - - - -
3	0	1	0	2	0	0	M_1-*Ru*; M_2-*Wu*; M_1-*Wu* (*Flush* M_1 to M_2; *Insert* t3 in M_1)	t3 -	t1 t2 - -	- - - - - - - -
4	0	2	0	2	0	0	M_1-*Wu* (*Insert* t4 in M_1)	t3 t4	t1 t2 - -	- - - - - - - -
5	0	1	0	4	0	0	M_1-*Ru*; M_2-*Wu*; M_1-*Wu* (*Flush* M_1 to M_2; *Insert* t5 in M_1)	t5 -	t1 t2 t3 t4	- - - - - - - -
6	0	2	0	0	0	4	M_1-*Wu*; M_2-*Ru*; M_3-*Wu* (*Insert* t6 in M_1; *Flush* M_2 to M_3)	t5 t6	- - - -	t1 t2 t3 t4 - - - -
7	0	1	0	2	0	4	M_1-*Ru*; M_2-*Wu*; M_1-*Wu*; (*Flush* M_1 to M_2; *Insert* t7 in M_1)	t7 -	t5 t6 - -	t1 t2 t3 t4 - - - -
8	0	2	0	2	2	4	M_1-*Ra*; M_2-*Ra*; M_3-*Ra*; M_1-*Wu*; (*Aggregate* t1:8; *Evict* t1:2; *Insert* t8 in M_1)	t7 t8	t5 t6 - -	-t1 -t2 t3 t4 - - - -
9	0	1	0	4	2	4	M_1-*Ru*; M_2-*Wu*; M_1-*Wu*; (*Flush* M_1 to M_2; *Insert* t9 in M_1)	t9 -	t5 t6 t7 t8	- - t3 t4 - - - -
10	0	2	0	0	4	0	M_1-*Ra*; M_2-*Ra*; M_3-*Ra*; M_3-*Wu*; M_1-*Wu* (*Aggregate* t3:10; *Evict* t3:4; *Flush* M_2 to M_3; *Insert* t10 in M_1)	t9 t10	-t5 -t6 -t7 -t8	- - -t3 -t4 t5 t6 t7 t8
11	0	1	0	2	4	0	M_1-*Ru*; M_2-*Wu*; M_1-*Wu* (*Flush* M_1 to M_2; *Insert* t11 in M_1)	t11 -	t9 t10 - -	- - - - t5 t6 t7 t8
12	0	2	0	2	6	0	M_1-*Ra*; M_2-*Ra*; M_3-*Ra*; M_1-*Wu* (*Aggregate* t5:12; *Evict* t5:6; *Insert* t12 in M_1)	t11 t12	t9 t10 - -	- - - - -t5 -t6 t7 t8
13	0	1	0	4	6	0	M_1-*Ru*; M_2-*Wu*; M_1-*Wu* (*Flush* M_1 to M_2; *Insert* t13 in M_1)	t13 -	t9 t10 t11 t12	- - - - - - t7 t8
14	0	2	0	0	0	4	M_1-*Ra*; M_2-*Ra*; M_3-*Ra*; M_3-*Wu*; M_1-*Wu* (*Aggregate* t7:14; *Evict* t7:8; *Flush* M_2 to M_3; *Insert* t14 in M_1)	t13 t14	-t9 -t10 -t11 -t12	t9 t10 t11 t12 - - -t7 -t8
15	0	1	0	2	0	4	M_1-*Ru*; M_2-*Wu*; M_1-*Wu*; (*Flush* M_1 to M_2; *Insert* t15 in M_1)	t15 -	t13 t14 - -	t9 t10 t11 t12 - - - -

(b)

Figure 4.4: (a) A stream of tuples t1, t2, . . . , of a key aggregated with WS=8 and WA=2 tuples. (b) Hash Table metadata logic and dataflow through the memory hierarchy for the above stream. Memories configured with: v_1=2; v_2=4; v_3=8. M_i-Wu, M_i-Ru, and M_i-Ra denotes write operation due to window update, read operation due to window update, and read operation due to aggregation in memory level i, respectively. The red numbers in the Init row indicate the indices of each memory cell. The grey and blue boxes indicate window updates and aggregation reads, respectively.

and needs to be *flushed* to the next level; iii) the index of the successive level to *insert* the *flushed* block of values from the predecessor; and iv) whether the key is ready for *aggregation* and perform *eviction* of invalid entries on a window-slide. Based on the above cases, the entry is updated and written back to the hash table.

Figure 4.4 illustrates an example of the index management for tuple-based stream aggregation using the hash table. When the first tuple t1 enters the system, the write pointer w_1 gets incremented by 1 as the tuple is to be inserted in M_1. Similarly, for t2, w_1 becomes 2. On arrival of t3, as $w_1 = 2$ (equal to v_1, the capacity available per key in M_1), M_1 has to be flushed to M_2. Then, t3 is inserted in M1 and w_1 gets

updated to 1 pointing to the index in M_1 where the next tuple has to be inserted. Similarly, on arrival of t5, as M_2 is full ($w_2 = v_2$), M_2 is to be flushed to M_3. This goes on until t8 when the total count of tuples in the single window becomes equal to WS ($(w_1 - r_1) + (w_2 - r_2) + (w_3 - r_3) = 8$ tuples) and triggers aggregation. On aggregation, the entire single window spanning across the memory hierarchy has to be read, and then the invalid tuples evicted based on the WA (2 tuples). The eviction is marked by incrementing r_3 by 2 which reduces the total tuple count to 6. An interesting point to note is that on arrival of t10, the buffer in M_3 wraps around and on t14, M_2 gets flushed to the first line in M_3. This maintains the circular buffer per key which is statically allocated in the memory hierarchy. In this example, the read pointers of M_1 and M_2 remain to zero, because on a window-slide (WA=2) the evictions did not span to M_1 or M_2. In a case where the evictions span multiple levels (large WA), the MLQ parts of the upper levels will be emptied and the read pointer of the lower affected level will be updated.

For a tuple-based windowing policy, the read and write pointers memory can be reused to manage evictions. For a time-based case, more metadata are required to manage the number of evicted tuples upon aggregation as described in [26], because the number of tuples per slide (WA) is time-dependent. This is orthogonal to the metadata used for supporting MLQ. Another note is that the MLQ storage in each level $i < n$ can be reduced from v_i to $v_i - v_{i-1}$ because the last write access of v_{i-1} elements to level i triggers a flush to level $i+1$ and therefore it can be forwarded without storing it in i.

After the hash table access, the tuple along with its hash address, read/write indices, full flag per memory level, and a "ready for aggregation" flag are pushed to the command generator.

4.2.2 Memory Command Generator

The command generator controls the memory levels and converts the window update (insert, flush) and aggregation operations into access commands to each memory level. It converts the received indices to actual physical addresses to each memory level. As the available memory space of each level is statically allocated equally to a block per key, the physical address is created using the key block offset. Then, the address of the memory line(s) to be accessed within the key block is generated based on the received index, the value size, and the number of values to be accessed. In case, multiple memory lines need to be accessed (i.e. due to flushing, aggregation), then multiple commands are generated. Finally, in case of a write access at a granularity smaller than a memory line, multiple commands are generated to implement a read-modify-write. Figure 4.4 describes for the given example the commands to the different memory levels generated by the memory command generator.

After this stage, the tuple along with the number of lines read based on the read commands generated, the full flag per level as well as the aggregation ready flag are passed to the data collection stage.

4.2.3 Data Collection Controller

This module synchronizes the data read from memory levels due to flushes and aggregations each of the two cases using a separate state machine. The flushing state machine checks the full-flag of each memory level and writes the flushed data to the

Table 4.1: Platform memory specifications

Type	Capacity	R/W Access Granularity (bytes)	Theoretical Bandwidth	Avg. Access Time in FPGA cycles (156.25 MHz)
On-chip BRAM	2 MB	4 in our design	14.4 TB/s	1
Off-chip SRAM	72 MB	R: 18 / W: 1	9.9 x2 = 19.8 GB/s	1.2
Off-chip DRAM	24 GB	64	12.8 x3 = 38.4 GB/s	7 (<4 lines), 2 (>4 lines)

next level. In case of a read-modify-write, the lines are modified before flushing to the next level. The aggregation state machine maintains separate internal queues for the tuple's metadata and for the data flowing in from each memory level. It then synchronises the contents of the different queues to deliver in order the aggregated values to the subsequent compute kernel. Value reordering is needed because the aggregation read operations are performed in parallel for all memory levels that contain valid data. This is important for supporting non-commutative aggregation functions like `rank`. Finally, the data are pushed to the compute kernel for computing the functions.

4.2.4 Computer Kernels

Depending on the aggregation function in the query, the values can be processed on the fly, gradually as they arrive, e.g. for algebraic (i.e.,`average`) or distributive functions (i.e., `minimum`, `maximum`, and `sum`), or otherwise only when all values have been received, e.g., for holistic functions, such as `median`. In our implementation, the compute kernel is pipelined and for `median` a Histogram-based median filtering is implemented [49]. Our design is implemented for a particular (worst-case) WS and WA, but one can choose dynamically at runtime to use a lower WS and/or a different WA, within the same memory configuration. The selected WS and WA is the same for all keys. Multiple queries can be supported by implementing multiple parallel compute kernels. Should these compute kernels be implemented in a way that supports dynamic partial reconfiguration, then these queries could be updated at runtime. After the function computation, the result is forwarded to the Tx stage.

4.3 Evaluation

The performance of the proposed approach 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 against existing approaches.

4.3.1 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. As shown in Table 4.1, besides on-chip BRAMs and 24 GB DDR3 DRAM, the board offers off-chip QDR-SRAM memory [53]. The designs are implemented in MaxJ, a Java-based High level Synthesis (HLS) language, and compiled using MaxCompiler.

Table 4.2: Resource utilization for the FPGA implementation.

Resource	DFE(D)	DFE(B+D)	DFE(B+Q+D)
Logic (ALMs)	95998 (26.73%)	98137 (27.32%)	102300 (28.48%)
BRAMs	1429 (54.14%)	1705 (64.57%)	1798 (68.09%)

Three different types of FPGA-based single window SWAG dataflow engines (DFEs) are implemented. A SWAG engine that uses only DDR3 DRAM, denoted as DFE(D), that follows the designs described in the current state-of-the-art [9, 26]. A SWAG engine with BRAM and DRAM, denoted as DFE(B+D) and follows the general principles of 2-level SRAM/DRAM hybrid memories used in network processing [27–31]. A SWAG engine with a 3-level MLQ memory system using BRAM, off-chip QDR-SRAM and DRAM, denoted as DFE(B+Q+D) and best captures the principles of our approach, although as explained below for some SWAG problems using fewer levels may suffice. The configuration of each design (partitioning V per level) was generated using the proposed heuristic. It is worth noting that none of the SWAG problems sizes used could be supported by a design that uses only BRAM. Finally, a software baseline is used, implemented in Apache Flink v1.5.1 [16] running on an Intel Core i7-4790 CPU at 3.6 GHz using 24 GB DDR3 DRAM.

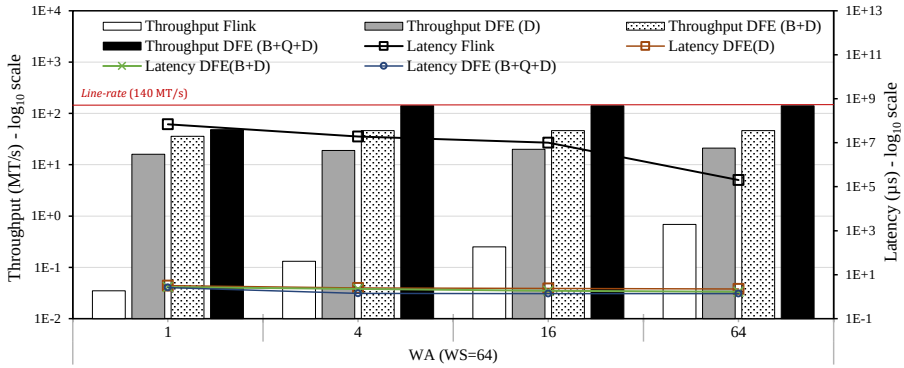
As a stream aggregation application, a subset of the LinearRoad benchmark [43] is used, where each vehicle has a sensor that emits tuples composed of a timestamp (24b), a vehicle ID (key, 24b), and speed (value, 16b). The data set is uniformly distributed, and the tuple's keys and values are generated with uniform probability. The *tcpreplay* tool [42] is used to inject the captured packets at varying injection rates to determine the highest sustainable system throughput.

The implemented query, comprising of algebraic, distributive, and holistic aggregation functions, is the following: "Find the `average`, `minimum`, `maximum`, and `median` speed for each vehicle for the last WS tuples and return the aggregate every WA tuples." The WS ranges from 64 to 4K tuples, the WA varies from 1 to WS tuples and the number of vehicles (keys) is 128K which is in line with our goal of supporting a wide range of *group-by* stream aggregation queries with large window sizes, frequent aggregations, generic aggregation functions, and a large number of keys.

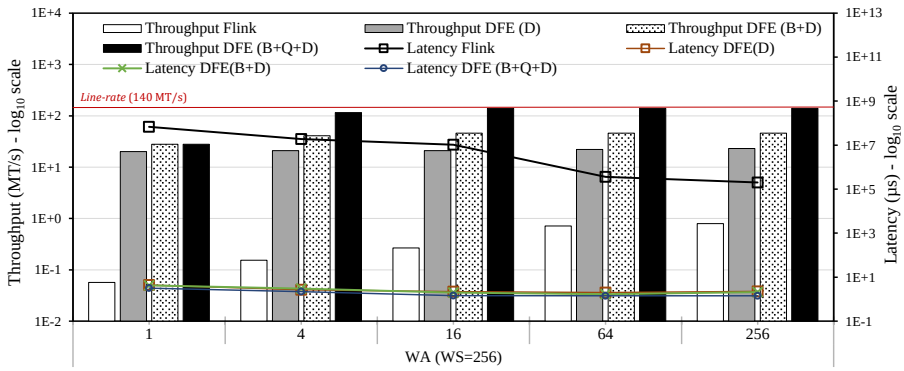
4.3.2 Implementation Results

The resource utilization of the proposed design is as depicted in Table 4.2. The logic utilization is mainly constituted of the hash table stage and the compute kernel (40%) implementing the aggregation functions. The increase in logic utilization across designs is mainly attributed to the increase in pointer arithmetic for managing the extra levels in the hash table stage, command generator, and data collector. The BRAM utilization also increases with more memory levels as more read/write pointers are required in the hash table. For value storage, 512KB of BRAMs are allocated providing 4B per key for DFE(B+D) and DFE(B+Q+D) designs. The remaining BRAMs are mostly utilized by FIFOs between the pipeline stages.

All DFE designs operate at 156.25 MHz allowing to receive one incoming tuple/-cycle using the full 10 Gb/s bandwidth of the network interface. This translates to a theoretical line rate of 156.25 million tuples/sec, but in practice, the actual highest rate of incoming tuples measured on the board is about 140 million tuples/sec, so about



(a) WS=64 tuples



(b) WS=256 tuples

Figure 4.5: 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.

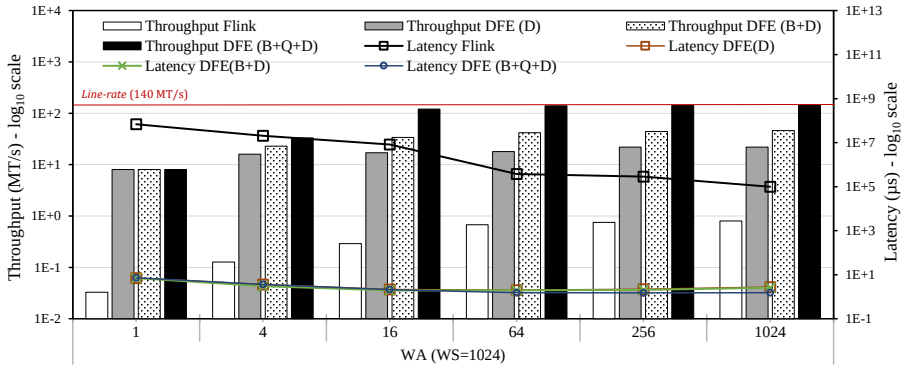
90% of the theoretical. Finally, QDR-SRAM and DDR3 DRAM are clocked at 350 MHz and 800 MHz, respectively.

4.3.3 Processing throughput and latency

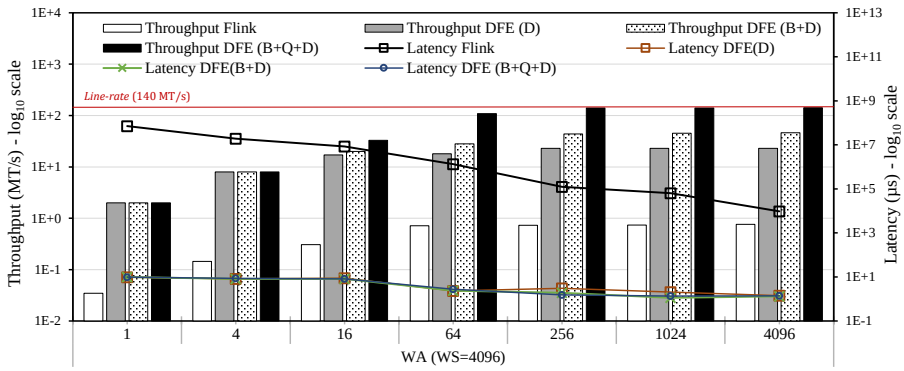
Figures 4.5 and 4.6 show the processing throughput of the alternative SWAG designs for different WS and WA, measured in number of tuples processed per unit of time. It also depicts the average (per aggregation) latency, which is interesting for systems with real time constraints.

A general observation is that the throughput of all designs is reduced for small window advance (WA), especially for large window sizes (WS). This is because smaller WA trigger aggregations more frequently and in addition, the larger windows aggregate more data, hence the problem becomes more bandwidth demanding.

The performance of the reference Flink software confirms the claim that CPU memory management does not suit the considered SWAG requirements as it offers 2-3 orders of magnitude lower throughput and 4-7 orders of magnitude higher latency compared to DFEs.



(a) WS=1024 tuples



(b) WS=4096 tuples

Figure 4.6: 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.

DFEs exhibit similar latency, as DFE(B+Q+D) has on average 90% and 99% the latency of DFE(D) and DFE(B+D), respectively.

The DRAM-only design, DFE(D), which follows the design principles of [9, 26], achieves the lowest throughput out of the three DFEs, supporting up to 15% of the line rate because it handles inefficiently the window update accesses. More precisely, it requires slow and bandwidth wasteful read-modify-writes, since value size (2B) is smaller than DRAM line (64B). However, for small WA (1) and large WS (1024-4096) it offers about the same throughput as the other two DFE designs as shown in Figures 4.6a and 4.6b. This is due to the following reasons. First, in these cases the aggregation traffic dominates and therefore the inefficient handling of window updates has negligible effect. Second, DFE(B+D) and DFE(B+Q+D) cannot take advantage of the added aggregation bandwidth offered by BRAM and BRAM+QDR-SRAM, respectively, because for large WS the dominant portion of the window is stored in DRAM which becomes the bottleneck.

Adding BRAM to form a BRAM+DRAM DFE, DFE(B+D), improves throughput offering up to 30% of the line rate. Although better than DFE(D), the DFE(B+D) memory system is still inefficient. For the given large number of keys, BRAM capacity is too small to fit an entire DRAM line per key, and so cannot completely eliminate

read-modify-writes. It can store only two values per key in BRAM, so compared to DFE(D) it reduces the expensive read-modify-write DRAM operations to half, offering better but still limited throughput.

Adding another memory level between BRAM and DRAM solves the above problem and supports up to 90% of the theoretical line rate, which in practice matches the actual maximum rate of incoming tuples on the board. The off-chip QDR-SRAM employed in DFE(B+Q+D) offers the capacity required for storing an entire DRAM line of key-values before flushing to DRAM, completely eliminating read-modify-writes. Moreover, it offers higher aggregate memory bandwidth. This is mostly evident in problems with small WA and WS, because in small WAs, aggregation traffic dominates and in small WS, significant part of the window is not in DRAM, e.g., for WS=64, half of the window is in the lower memory levels.

4.3.4 Accuracy of the analytical performance model

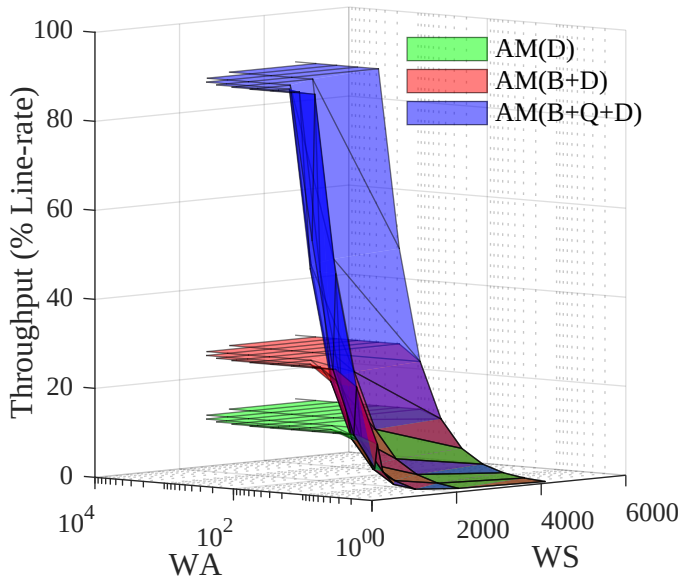
Figures 4.7a and 4.7b show for various WS and WA the system throughput estimated by our analytical model (AM) and measured in our actual implemented DFEs, respectively. The mean absolute percentage error (MAPE) is 12%, but as it can be observed the trend is estimated correctly which is sufficient to guide well our heuristic.

4.3.5 Comparison with related work

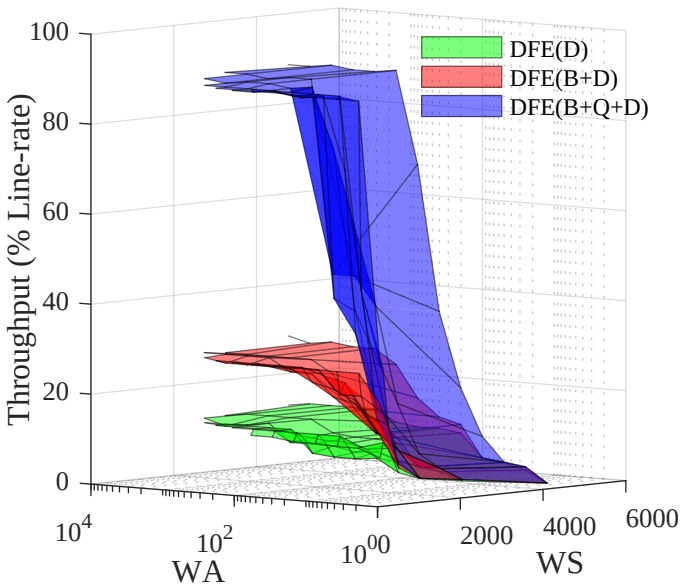
Compared to existing works, our approach offers higher performance for larger problems due to the use of MLQs.

The DFE(D) implementation follows the design of existing DRAM-only FPGA approaches [9, 26] and is 6-8$\times$ slower than our 3-level MLQ design. Another advantage of our approach compared to DRAM-only designs is that it handles skewed key distributions without additional support. The design in [9] would suffer from consecutive tuples of the same key as it would create read after write hazards in the pipeline. The design presented in [26] uses an additional cache structure before the DRAM controller to deal with consecutive DRAM accesses by tuples of the same key. On the contrary, our designs perform all window updates in the fastest level-1 which offers single cycle access and therefore there are no read-after-write-hazards. Moreover, the more recently received values of each key are at the lower levels supporting better performance. We experimented with skewed distribution traffic (from real-world datasets [47, 48]) with the same set of associative and non-associative aggregation functions, and confirmed that the throughput of our design is agnostic to the key-distribution. Finally, BRAM-only FPGA designs such as the ones presented in [6–8] can support only 4 orders of magnitude smaller stream aggregation problems; that is up to WS=8 for the considered query and number of keys.

To the best of our knowledge, the only GPU-based stream processing hardware accelerator that supports non-associative functions (i.e., `median`), therefore using a non-incremental aggregation approach is Gasser [4]. However, Gasser supports queries with only a single key, hence small problem sizes, and also does not capture tuples from the network. We experimented with a single key query and DFE(B+Q+D) was able to achieve similar throughput (line rate) comparable to Gasser for varying WS/WA. The throughput readings are similar to the experiment with uniform key distribution traffic. However, DFE achieves this at a much lower latency, which is 3-4 orders of magnitude lower than Gasser.



(a) Throughput estimated by the anaytical model (AM).



(b) Throughput measured for the actual DFE implementations.

Figure 4.7: System throughput as percentage of line-rate: (a) Estimated by the analytical model (AM); and (b) Measured in the hardware implemention using Dataflow Engines (DFE) for the various memory configurations, namely, with only DRAM (D), Block-RAMs + DRAM (B+D) and the 3-level MLQ system with Block-RAMs + QDR-SRAM + DRAM (B+Q+D).

4.4 Conclusion

This chapter introduced Multi-level Queues (MLQs), a specialized memory hierarchy for stream aggregation. MLQs use multiple memory levels to form logical queues that offer on-chip SRAM (BRAM) bandwidth for window updates and DRAM capacity. In addition, they employ the aggregate bandwidth of all levels in the hierarchy, offering higher aggregation throughput. Compared to BRAM-only stream aggregation designs, MLQ supports 4 orders of magnitude larger problems. Compared to DRAM-only designs, it achieves up to 8× higher throughput. Even compared to hybrid BRAM+DRAM designs, MLQ has up to 4× higher throughput. Finally, MLQ offers the same throughput as GPUs, but in addition supports group-by operations –up to 128K keys rather than one key offered by the competing GPU systems– and 4 orders of magnitude lower aggregation latency.

Chapter 5

StreamZip: Compressed Sliding-Windows for Stream Aggregation

The massive volumes of data produced globally enable a large number of emerging stream processing applications [50]. Such applications are used in various domains, e.g., financial, transportation, to analyze large unbounded streams of data and make fast, sophisticated decisions. However, consuming large data volumes at line rates require *high processing throughput* and sometimes, e.g., in financing, also *low latency*.

Stream aggregation is one of the most challenging tasks in stream processing. An example is depicted in Figure 5.1. 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 [6–8]. However, many others need to follow the *single window* stream aggregation (Single-SWAG) approach [9,26], which is the focus of this paper. 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 [5], social-media data [10] or manufacturing-equipment data [11].

Single Sliding-Window stream AGgregation (Single-SWAG) is a memory-intensive problem [13]. For each incoming tuple, the memory needs to be accessed to update the window and, when ready for aggregation, the entire window should be read. These two

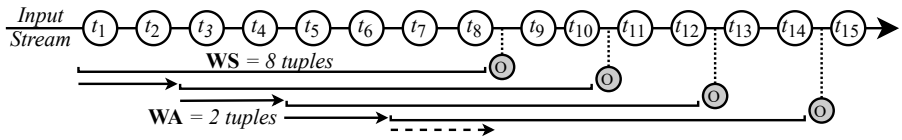


Figure 5.1: Sliding-window stream aggregation with Window Size (WS) = 8 tuples and Window Advance (WA) = 2 tuples for a input data stream, e.g. a vehicular sensor emitting tuples $t_1, t_2, \ldots$ with each tuple containing a timestamp, vehicle-ID and speed. For simplicity, only tuples from a single vehicle (key) is shown. The grey tuples indicate the aggregate output generated when the sliding-window gets full. (e.g. `top-3`*,* `average`*, and* `median` *speed).*

Single-SWAG steps, *window update* and *window aggregation*, generate tremendous memory pressure, the latter especially for queries with frequent aggregations (small WA). In Chapter 4, we discussed the memory bandwidth bottleneck due to window updates being mitigated using multi-level queues (MLQ) [15]. 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 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 counters, sensor, geolocation and other time-series data have significant redundancy that can be exploited through data compression [14]. As an example, Figure 5.2 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 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 [14]. However, TerseCades only supports batch-processing of separate non-overlapping tumbling windows, i.e., WA=WS, rather than true stream processing with WA $\leq$ 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.

This work introduces StreamZip, the first true stream processing engine with compression support for sliding windows and WA $\leq$ WS. StreamZip is based on previous FPGA-based multi-level queue (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.

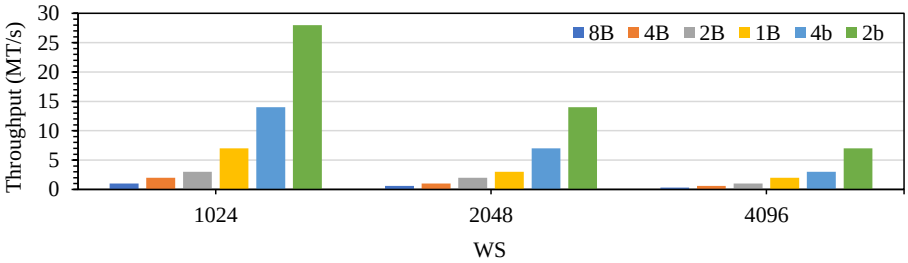


Figure 5.2: Processing throughput in million tuples per second vs. WS in tuples for an FPGA-based MLQ Single-SWAG dataflow engine [15] at 156.25 MHz with varying value sizes and WA = 1 tuple, showing the potential for data-compression.

Table 5.1: Data-types and Compression schemes in StreamZip

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

The contributions of this work are the following: StreamZip, a novel dataflow, true stream processing engine with compression support for sliding windows with any WA, which:

- supports diverse compression algorithms;
- devises efficient concurrency control mechanisms to mitigate the data and control dependencies in the pipeline;
- 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 remainder of this chapter is organized as follows. Section 5.1 offers background about compression and discusses related work. Section 5.2 describes the StreamZip design. Section 5.3 presents the evaluation and compares StreamZip with related works. Finally, Section 5.4 summarizes our conclusions.

5.1 Background and Related Work

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

5.1.1 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 little data dependencies as possible. Adding compression to such a system introduces a number of challenges. Briefly, the challenges pertaining to the choice of compression

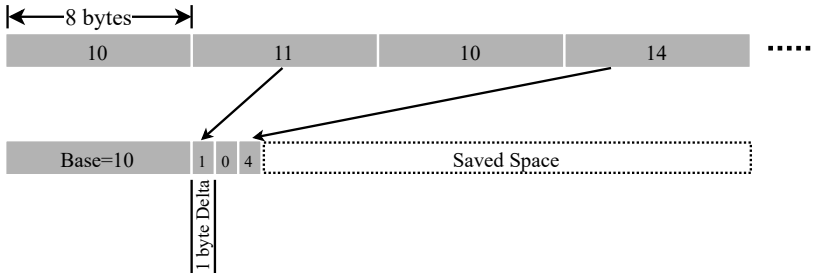


Figure 5.3: Base-Delta compression of a stream of 8 byte values to 1-byte deltas.

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 5.1, 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 used for decades in computing systems [54, 55] and offers competitive compression ratio for a wide range of fixed-point integer data, ranging from performance counters, 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 5.3 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 the XOR lossless compression scheme for floating-point numbers [56, 57]. Base-delta lossless encoding does not work well for floating-point numbers as it does not reduce the number of bits sufficiently and hence results in low compression ratios [14]. 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 would have a substantial number of leading zeros. Hence, it can be encoded and compressed by a leading zero count that is followed by the remaining bits. The incoming tuple's double-precision floating-point values are variable length encoded using XOR compression [57] 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 is calculated and a '1' bit is stored followed by either:
  - Control bit '0': If the block of meaningful bits falls within the block of previous meaningful bits, i.e., the number of leading zeros and trailing zeros is at least as many as in the previous XORed value, that information

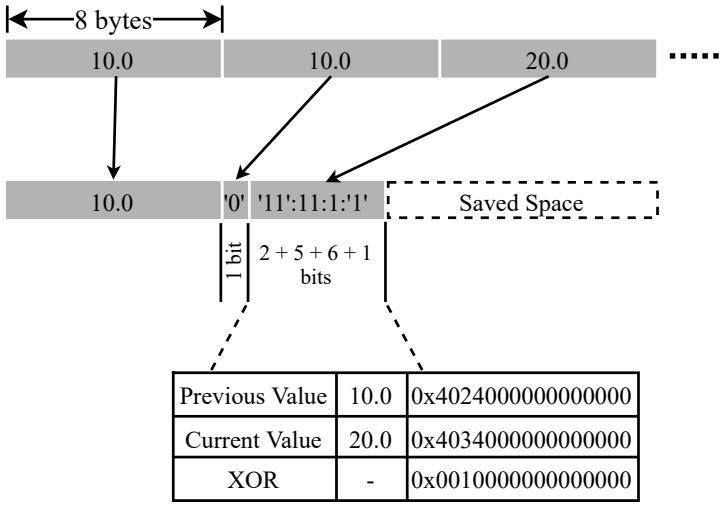


Figure 5.4: XOR compression of a stream of 8 byte values.

is used for the block position and just the meaningful XORed value is stored.

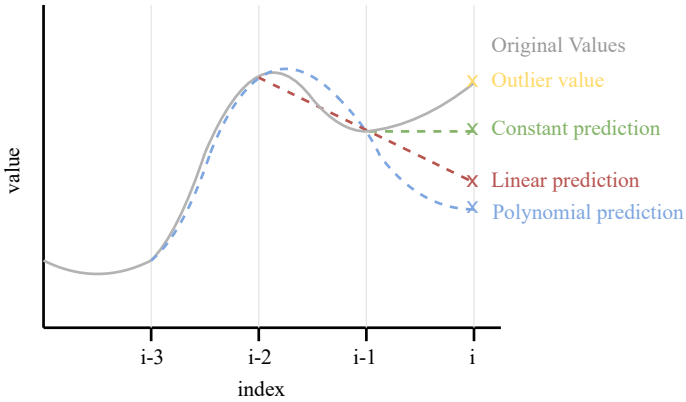
– Control bit '1': The length of the number of leading zeros is stored in the next 5 bits, then the length of the meaningful XORed value in the next 6 bits. Finally, the meaningful bits of the XORed value is stored.

Figure 5.4 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 a sequence of XORed values in time-series data often have a very similar number of leading and trailing zeros [57].

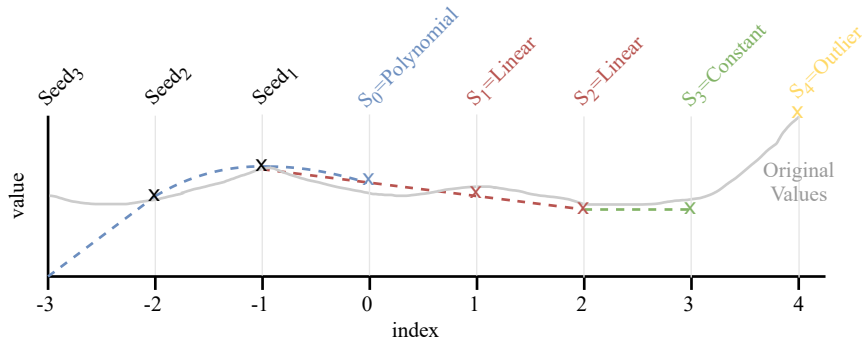
The third algorithm used in StreamZip is the lossy Squeeze (SZ). SZ compresses a sequence of numeric values by describing each value X_i as a function of the three preceding values $[X_{i-3}, X_{i-2}, X_{i-1}]$, according to a predefined function model [58]. A typical SZ supports four such function models (to be encoded with 2-bits per value), namely:

- a *constant* value is approximated as equal to the nearest preceding one,
- a *linear* value is extrapolated from the preceding two values,
- a *polynomial* value fits on the cubic curve described by the preceding three values, 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*).

Figure 5.5a shows the four function models used in SZ. It is clear that computing a value depends on the computations of its previous three, hence the computing complexity is important. In case computing a value takes more than a cycle, then



(a) Four function models in SZ compression.



(b) Reconstruction of SZ value stream.

Figure 5.5: SZ Compression

throughput can be limited. As shown by Eldstål et al., computing a value can be reduced for the most complex function (polynomial) to two pipeline stages, each with a 3-operand addition [59]. 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 offers roughly 4× higher compressibility than lossless compression schemes. Figure 5.5b shows the decompression of a SZ value sequence.

In the past, SZ has been used in the GhostSZ FPGA-based system for I/O compression [60]. GhostSZ breaks the dependencies between computing consecutive values by splitting them to multiple parallel sequences that are compressed separately, thereby increasing throughput.

5.1.2 Stream processing platforms

Various computing alternatives have been used for stream processing and stream aggregation in particular, each having different functionality and performance potential.

Generic software platforms running on general-purpose CPU offer a wide range of stream processing capabilities, but have limited throughput and high latency [16–18].

Multicore CPU and GPU based systems are able to sustain high processing throughput, but have wasteful memory management [13] as they require redundant memory accesses to store incoming tuples from the network to DRAM even before processing starts. Some CPU-based approaches propose algorithmic modifications to reduce latency, but are constrained to only associative aggregation functions [2, 19]. StreamBox-HBM exploits high bandwidth HBM memories to improve the processing throughput [61]. TerseCades achieves the same goal with compression [14]. However, both StreamBox-HBM and TerseCades only support aggregation queries with tumbling windows (WS = WA). The two fastest GPU stream aggregation systems are SABER, which performs only incremental aggregations [20], and Gasser, which supports non-associative functions too [4].

FPGAs can offer direct network connection to receive incoming tuples and support dataflow processing. This enables FPGAs to deliver both high processing throughput, on par with the fastest GPU and CPU systems, as well as low latency, at least 3 orders of magnitude lower than CPUs and GPUs [9].

FPGA designs based on incremental aggregation algorithms does not support *non-associative* functions [7, 8]. Recently, there has been work on FPGA-accelerated approximable query processing using sketches [62] 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 [26] windowing policies, explicitly store values and support generic aggregation functions (including non-associative), varying window-slides, large window sizes and large number of keys, as discussed in Chapters 2 and 3. However, these works use only DRAM for maintaining the single window state which require slow and bandwidth wasteful read-modify-writes for window-updates. As discussed in Chapter 4, a multi-level queue (MLQ) approach for single window aggregation was proposed to offer faster window updates [15]. 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.

5.2 StreamZip Design

StreamZip is a reconfigurable, stream aggregation dataflow engine (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 [15], StreamZip windows are stored in multi-level queues, but in this case, they are compressed. Figure 5.6 illustrates the StreamZip pipeline. Incoming tuples 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 [52]. Each hash table entry corresponds to a key and stores metadata needed for *compression*, *multi-level memory management*,

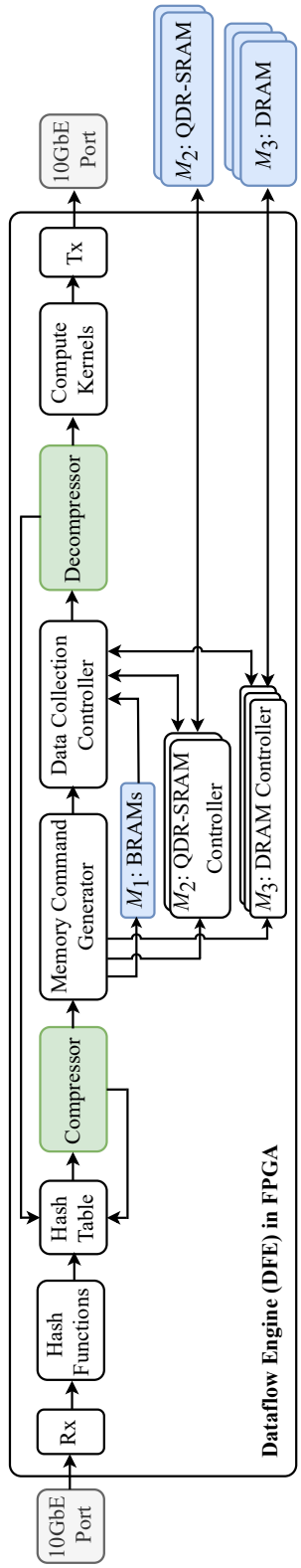


Figure 5.6: Top-level view of the StreamZip compression pipeline for dataflow-based stream aggregation.

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 synchronise 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.

5.2.1 Hash table organisation

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 [52]. 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.

5.2.1.1 Window management bank

The window management bank contains: (i) a valid bit; (ii) the key assigned to the entry; (iii) tuple counter, t_c, to determine when the key is ready for aggregation for tuple-based windows; (iv) window start timestamp for key replacement in case of collision and to trigger aggregation in time-based windows; and (v) optionally, starting stream number, 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 5.2.3.1.

5.2.1.2 Compressor bank

The compressor bank contains: (i) current seed(s), denoted by S_C, used to compress the incoming tuple's value; (ii) write pointers to each memory level, w_i, pointing to the tail index of each memory level for inserting the upcoming compressed value; and (iii) 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 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 5.2.2. 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 predictions based on the function models are stored.

5.2.1.3 Decompressor bank

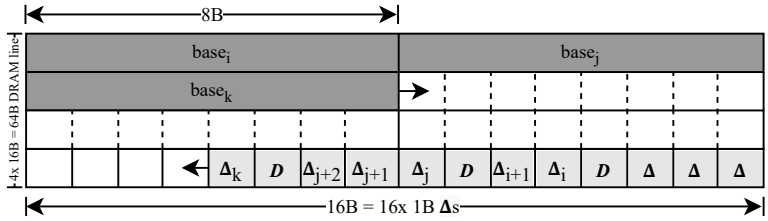
The decompressor bank contains: (i) starting seed(s), S_D, for decompressing the window, used to start the decompression of a window on aggregation trigger and (ii) read pointers to each memory level, r_i, pointing to the head index in level i from which to start reading the compressed values. The seeds 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 at 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 [52]. 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.

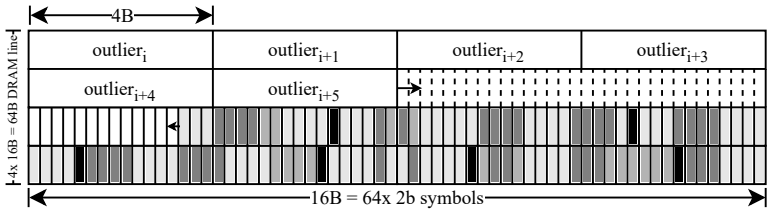
A hit in the hash table enables fetching the remaining metadata fields of the associated key from the compressor and decompressor banks. The compressor metadata determine: (i) the current seed(s) needed for the compression of the incoming tuple's value; (ii) the write address to insert the compressed value upon compression in M_1; (iii) 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; (iv) indicate whether a memory level is full and needs to be flushed to the next level; and (v) 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: (i) the starting seed(s) needed to decompress the window; and (ii) 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.

5.2.2 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 5.6. 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 64 bytes.



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

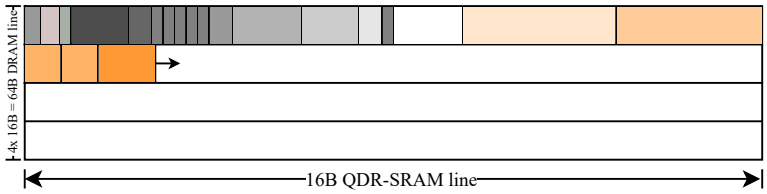


(b) SZ block format of a StreamZip compressed sliding window

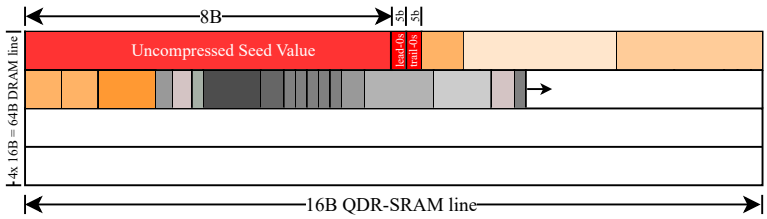
Figure 5.7: Block format of a compressed sliding window for two different compression algorithms: (a) Base-Delta fixed point lossless compression. Bases, denoted by base$_*$ and deltas denoted by Δ_* are written from the most and least significant addresses respectively. D represents the demarcation in the compressed sequence to denote a new base to be picked up from the most significant side; and (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 3 grey shades indicate constant, linear, and polynomial prediction compressed symbols.

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 few compressed values. Then, M_2 is configured to store one compressed block of 64 bytes per key matching the access granularity of M_3. The access granularity of M_2 is 16 bytes and so, four M_2 lines compose one CB that can be stored in one M_3 line.

Figure 5.7 shows the format of a CB for BD and SZ compression algorithms used in StreamZip. The size of an uncompressed value in a tuple is 8 bytes. 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,



(a) XOR Compressed block format with dependent DRAM lines



(b) XOR Compressed Block format with independent DRAM line

Figure 5.8: 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.

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 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 5.8a. 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.

5.2.3 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.

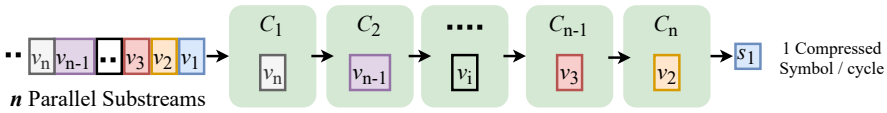


Figure 5.9: Value Interleaving to support multiple substreams in StreamZip. $C_1, \ldots, C_n$ denote the n pipeline stages of the compressor. The various color coded uncompressed values denoted by $v_1, \ldots, v_n$ represent the n substreams that can be interleaved and fed to the compressor to process one value every FPGA cycle to generate a compressed symbol, s_i, every cycle.

Besides making the pipeline longer, adding compression introduces the following challenges. Firstly, 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 creates 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.

5.2.3.1 Intra-(De)Compressor Pipeline Dependency

There are two inefficiencies within the (de)compressor pipeline affecting the processing throughput of StreamZip that needs 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 [59, 60]. The incoming values of each key are divided into *multiple independent sub-streams* as shown in Figure 5.9. Then, a pipelined version of the SZ computations can handle these sub-streams without data hazards. This is performed by storing the initial seeds 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. Although this is sufficient for maintaining correctness and high processing throughput, it complicates window aggregation and may affect compression quality.

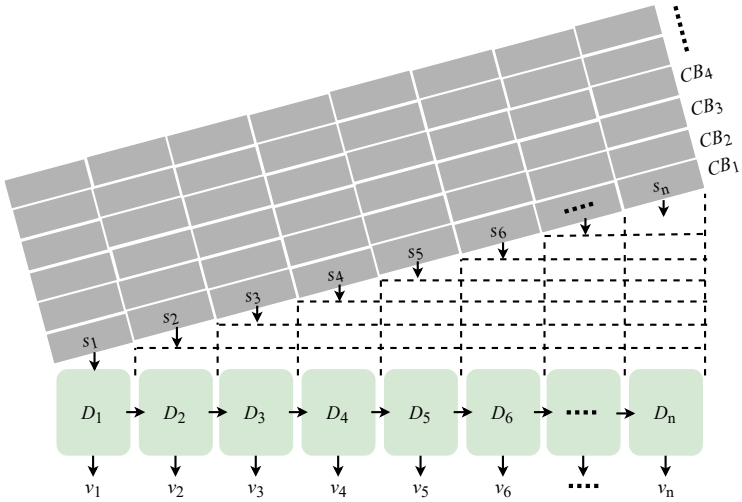


Figure 5.10: Compressed Blocks denoted by $CB_1 \ldots CB_n$ processed once every cycle when there are no dependencies between consecutive CBs. n denotes the number of compressed symbols in a block which are fed to the unrolled decompressor stages, $D_1, \ldots, D_n$. $s_1, \ldots, s_n$ denote the various symbols which are buffered and fed to the decompressor in consecutive cycles.

StreamZip removes some of this complexity by allowing all sub-streams to share the same memory block for aggregating values of the same key. Moreover, multiple decompressor pipelines are added, each handling a window of a different key, to cope with the processing rates.

(ii) Sequential dependency between values in compressed blocks: At the decompressor, the *sequential dependency* between consecutive values 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 and CBs. 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 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

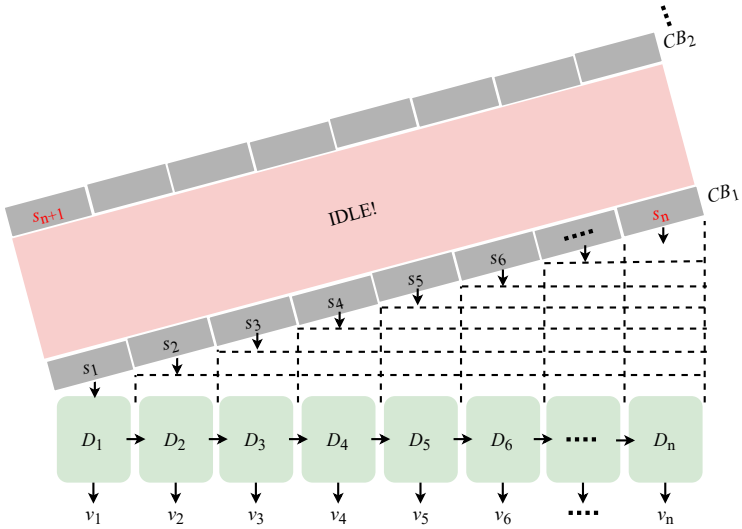


Figure 5.11: Idling due to sequential dependency between compressed blocks costing processing throughput. Here CB_2 can be fed to the decompressor only after n cycles after processing CB_1 as the decompression of symbol s_{n+1} in CB_2 requires the decompressed value from the symbol s_n in CB_1. $s_1, \ldots, s_n$ denote the various symbols which are buffered and fed to the decompressor in consecutive cycles.

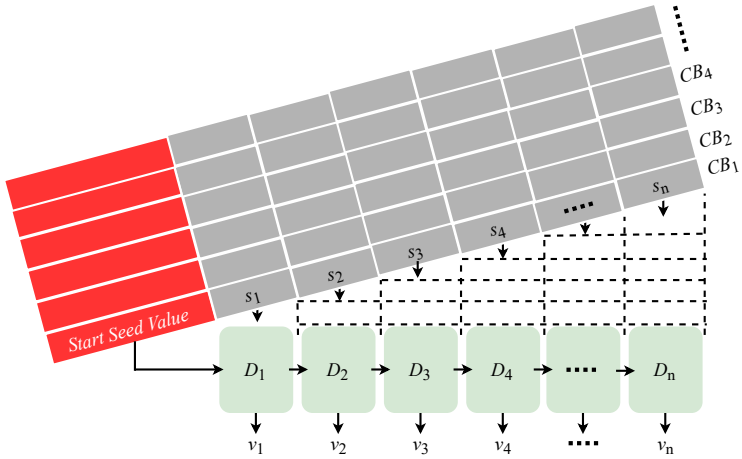
throughput can also be achieved for XOR compression by using pipelined parallel decompressors as there are no sequential dependency between CBs. However, for larger windows and keys triggering aggregation in close succession, the processing throughput is severely penalised due to the idle time between consecutive CBs as shown in Figure 5.11.

One way to solve this problem is by making the compressed blocks independent as shown in Figure 5.12a using the block format shown in Figure 5.8b. However, making compressed blocks *independent* by storing the starting uncompressed seed per CB can lead to lower compression ratios.

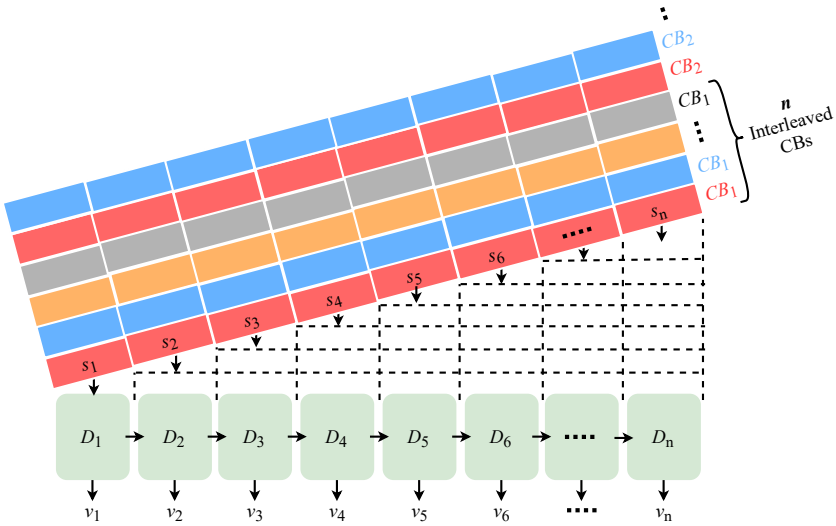
Another approach pursued in StreamZip is to perform *compressed block interleaving* across keys so that the each block entering the decompressor pipeline is from a different key in consecutive cycles as shown in Figure 5.12b. This would require buffering per key after the decompressor to store the intermediate decompressed values for computation. However, for skewed key distributions, it would not be possible to interleave compressed blocks which can be solved using caching the decompressed values as discussed in Section 5.2.3.5.

5.2.3.2 Inter-Compressor-Decompressor Dependency

When an aggregation is triggered before all tuples of a key already in flight have been compressed, the decompressor pipeline will have to stall until the compression is done, limiting processing throughput. This is pronounced when compression has higher latency. StreamZip addresses this by buffering the values under compression



(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.

Figure 5.12: Solutions for sequential dependency in compressed blocks.

until they are processed and if needed, forwarding them, instead of their compressed counterparts, to the compute kernel using a separate bypassing path.

5.2.3.3 Dependencies between Consecutive Aggregations

An input data stream with skewed key distribution and 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 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. 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.

5.2.3.4 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 5.2.3.2. 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 synchronises 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`.

5.2.3.5 Caching Optimization for skewed-key distribution

For skewed-key distributions and especially for large WS and frequent aggregations, redundant memory reads for the same single window of a key wastes valuable memory bandwidth and thereby, becomes 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 which triggered aggregation at the decompressor DFE pipeline stage using on-chip memory as shown in Figure 5.13. Moreover, reusing the single-window of a 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

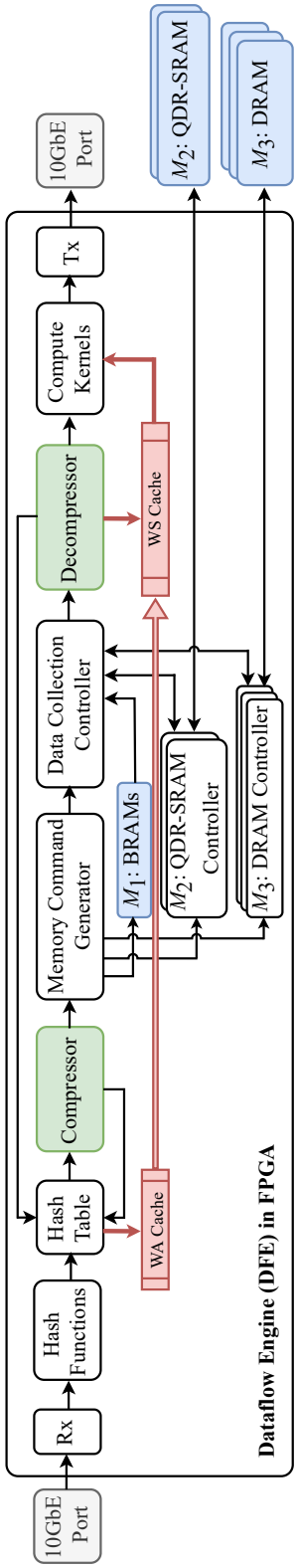


Figure 5.13: Caching Optimization for skewed-key distribution in StreamZip.

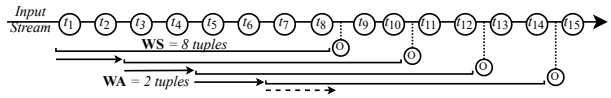
higher processing throughput for skewed-key distributions. The size of a cached block per key at the decompressor side is equal to WS, and the oldest tuples corresponding to WA will be evicted upon window-slide during aggregation. For tuple-based windows, the size is deterministic as the WS and WA are 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. 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 in a cycle.

At the compressor side, the recently seen keys which 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 and used in conjunction with the cached decompressed values at the decompressor. The replacement policy of the caches follow a LRU policy based on the incoming keys similar to the cache structure in Time-SWAD [26].

5.2.4 Memory Management and Dataflow

StreamZip memory management and dataflow is illustrated by the example in Figure 5.14. Here, lossless base-delta compression is described, but the dataflow is similar for other compression choices, too. 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 2 write pointers to indicate the base side written from the most significant memory cell, denoted by w_{2B} 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 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 5.2.2. 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 remains 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



(a)

		Compressor					Decompressor						
tup: value	t_e	S_C	w_1	w_{2S}	w_2	w_3	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	0 4 8 12 1 5 9 13 2 6 10 14 3 7 11 15
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)	-	2 - - - 1 - - -
t_4: 53	4	50	1	0	1	0	50	0	3	3	M_1-*Wu* (Insert Δ_4 in M_1)	3	2 - - - 1 - - -
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)	-	4 - - - 3 - - - 2 - - - 1 - - -
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	- 4 - - - 3 - - - 2 - - - 1 - -
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)	-	4 - - - 3 - - - D - - - 5 - - -
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	320 4 - - D 3 - - 5 2 - - - 1 - -
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)	-	- 4 - - - 3 - - 11 2 - - 10 - - -
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	- 4 - - (3) 320 - - 11 (2) - - 10 - 5 - -
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)	-	13 4 - - 12 320 - - 11 D - - 10 - 5 - -
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	- (4) 3 - - 320 12 - - D 11 - - (5) 10 -
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)	-	580 13 - - 20 320 12 - D - D 11 - - - 10 -
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	- - 320 13 580 - - - 12 20 - - - 11 D - - - (10) -
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)	-	- - 13 580 - - 12 20 22 - 11 D 21 - - -

(b)

Figure 5.14: (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 denotes the base and delta corresponding to t_i. Δ requires only 1 memory cell and a base (value) requires 2 cells. M_1, M_2, and M_3 configured to store up to 1 Δ, 4 Δs, and 8 values; M_i-Wu, M_i-Ru, and M_i-Ra denotes 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.

the compressed sequence for decompression, S_C=320 is written to M_2 leading to w_{2S} to point to 2 as a base occupies 2 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 to 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_1. 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 $\times$ number of keys).

5.3 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.

5.3.1 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-ship QDR-SRAM, and 24 GB off-chip DDR3 DRAM. The 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 [9,26]. Secondly, a SWAG engine with a 3-level MLQ memory system using BRAM, off-chip QDR-SRAM and DRAM, denoted as MLQ, that follows the best performing previous design [15]. 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 3-level MLQ memory system. These designs are denoted by, StreamZip-Lossless-Fixed, StreamZip-Lossless-Float, and StreamZip-Lossy-Float respectively, and, aptly capturing the principles of our approach.

The memory configuration of each design (partitioning per memory level) was generated using the partitioning algorithm in [15] with MLQ configured to store 1 and 8 values in the M_1 and M_2 levels, respectively. StreamZip-Lossless-Fixed using BD compression scheme is configured with 1 and up to 64 Δs, StreamZip-Lossless-Float using XOR compression is configured with 8 bytes and 64 bytes, and StreamZip-Lossy-Float design with 4 and up to 256 symbols per key in the M_1 and M_2 levels,

Table 5.2: Resource utilization for the FPGA implementations.

Design / Resource	Logic (ALMs)	BRAMs	DSP
DFE-Base	86208 (24%)	1136 (43%)	0
MLQ	93392 (26%)	1347 (51%)	0
Streamzip-Lossless-Fixed	104168 (29%)	1584 (60%)	0
Streamzip-Lossless-Float	219112 (61%)	1796 (68%)	0
Streamzip-Lossy-Float	308912 (86%)	2139 (81%)	165 (47%)

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) [47] real world dataset is used as the input streaming data with the tuples composed of timestamp (32b), job ID (key, 32b), and CPU usage (value, 64b). The *tcpreplay* tool [42] is used to inject the captured packets at varying injection rates to determine the highest sustainable system throughput.

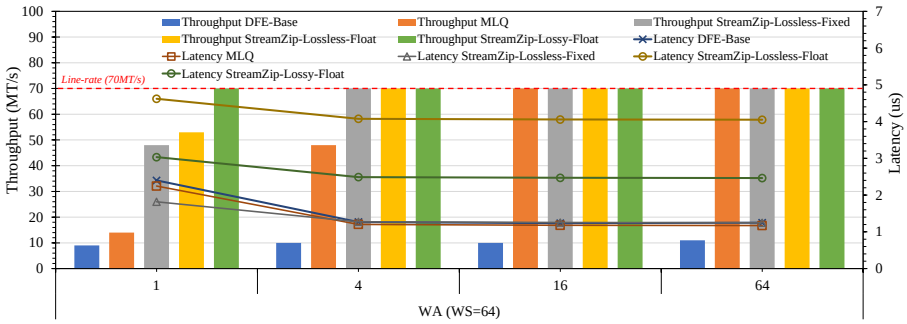
The implemented query, comprising of algebraic, distributive, and holistic aggregation functions, is the following: "Find the `average`, `minimum`, `maximum`, and `median` CPU usage for each job ID for the last WS tuples and return the aggregate every WA tuples" [20]. The WS ranges from 64 to 4K tuples, the WA varies from 1 to WS tuples with support for up to 16K concurrently active keys.

5.3.2 Implementation Results

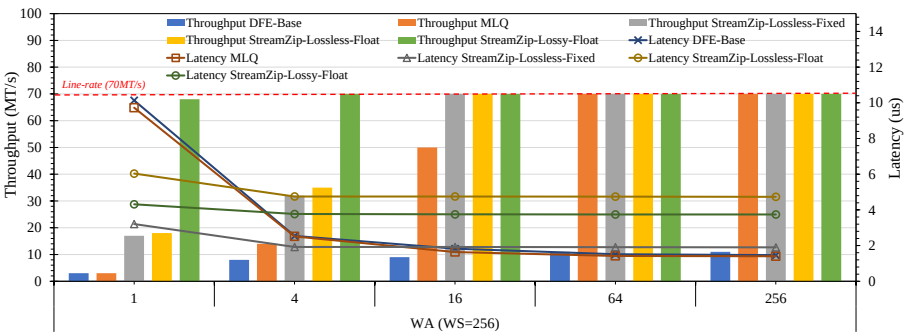
The resource utilization of the evaluated designs is as shown in Table 5.2. For StreamZip-Lossless-Fixed, the increase in logic utilization over MLQ is attributed to the extra (de)compressor metadata management and encoding. StreamZip-Lossless-Float requires a deeper decompressor pipeline due to the 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 attributes 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 3 (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



(a) WS=64 tuples



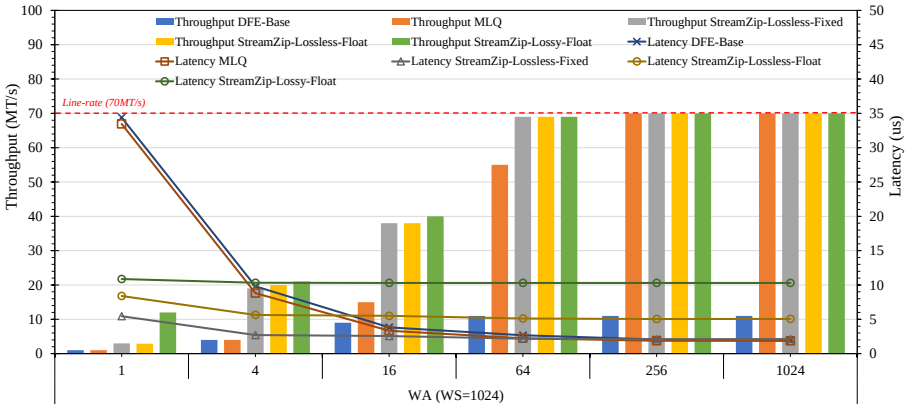
(b) WS=256 tuples

Figure 5.15: 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.

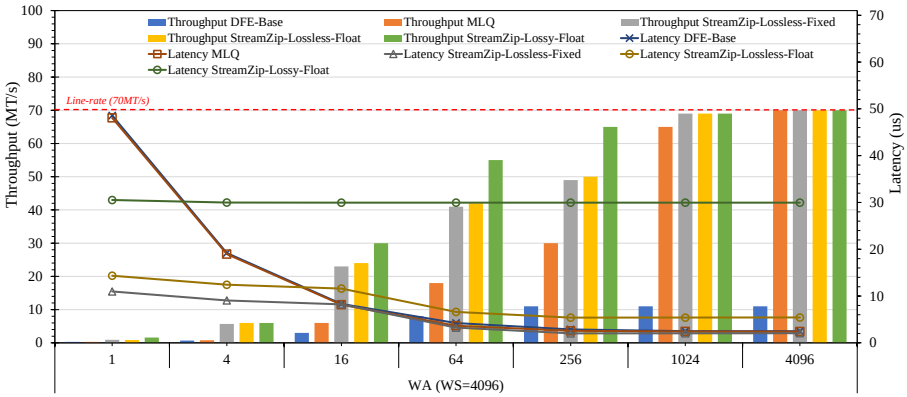
tuples/sec, so about 90% of the theoretical. Finally, off-chip SRAM and DRAM are clocked at 350 MHz and 800 MHz, respectively.

5.3.3 Performance results

A general observation is that the throughput of all designs is reduced for small window advance (WA), especially for large window sizes (WS). This is in line with the trend observed in the previous work [9, 15, 26] 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 line reads upon aggregation is smaller compared to DFE-Base and MLQ by a factor of the overall compression ratio achieved for the input data set. 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. 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



(a) WS=1024 tuples



(b) WS=4096 tuples

Figure 5.16: 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.

the number of DRAM lines read upon aggregation. The graphs in Figures 5.15a, 5.15b, 5.16a and 5.16b 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.

The latency follows an opposite trend as, 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 it 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 [9,26], achieves the lowest throughput out of the three designs, supporting up to 15% of the line rate. This is primarily 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, the dominant portion of the window is stored in DRAM which becomes the bottleneck.

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$\times$, leading to proportional improvement in processing throughput as shown in Figures 5.15b. StreamZip-Lossless-Float (XOR compression) achieves a compression ratio of 6$\times$ for this design point.

In addition, StreamZip-Lossless-Float design mitigates the performance bottleneck due to the sequential dependency between the compressed blocks (CBs) using *independent CBs* as discussed in Section 5.2.3.1. However, the trade-off here is the reduction in compression ratio as a result of adding the extra starting seeds to the beginning of each CB versus the benefit of processing up to CB every FPGA cycle. In our experiments, we see that this benefit outweighs the reduction in compression ratio 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 on average a compression ratio of 6.5$\times$ which gets reduced to 5.6$\times$ 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 two bits, offering better compressibility and hence higher throughput. Another interesting design point is WS=64, 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 lossless 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$\times$ and 23$\times$, respectively.

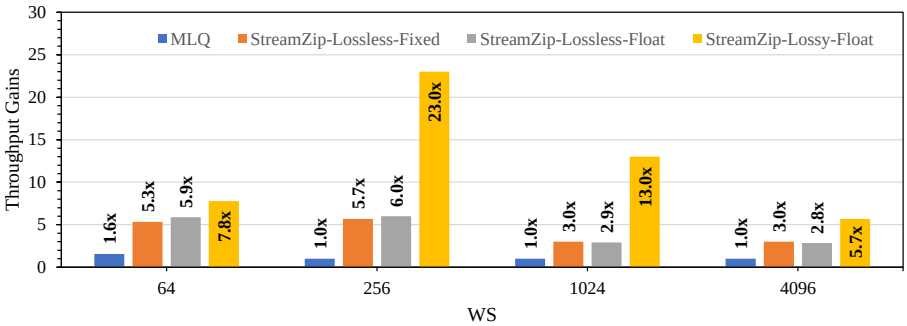


Figure 5.17: Throughput gains for various designs normalized to DFE-Base design for WA=1 tuple

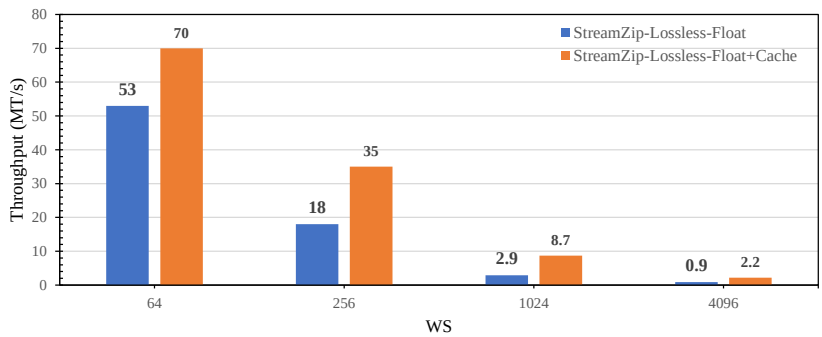


Figure 5.18: Throughput of StreamZip-Lossless-Float with Caching. A synthetic benchmark is used with a single key triggering aggregation in every cycle with WA=1 tuple.

5.3.3.1 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 5.17 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 achieves processing throughput gains of up to 5.7×, 6×, and 23×, respectively.

5.3.3.2 Caching results for skewed key distribution

Figure 5.18 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 data set with a single key triggering extremely frequent aggregations with WA=1. This causes each tuple entering StreamZip to trigger aggregation. 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 up to a window size of 128, meaning, the compute kernel is

fed with a window of 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.

5.3.3.3 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 [15], 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 offers up to 6×, 4×, and 3× lower latency than MLQ. However, for larger WS, StreamZip-Lossy-Float has up to 12× higher latency. This is due to the the deep pipeline required during aggregation for decompressing a large number of values packed per read DRAM line. Should more resources be allocated to implement multiple parallel decompressors, the latency cost of StreamZip-Lossy-Float would be reduced. 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 [4]. 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) comparable to Gasser for varying WS/WA. However, StreamZip achieves this at a much lower latency, which is 3-4 orders of magnitude lower than Gasser.

5.4 Conclusion

This chapter introduced 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 floating point 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.

Chapter 6

Conclusions

This thesis considered the design of FPGA-based accelerators for Single Sliding-Window Aggregation (*Single-SWAG*) supporting holistic aggregation functions. Multi-core CPU and GPU based stream processing systems, although able to sustain high processing throughput have wasteful memory management. They require redundant memory accesses to store incoming tuples from the network to DRAM even before processing starts. This, besides the latency overhead, wastes valuable memory bandwidth and hence limits performance. On the contrary, FPGAs use their memory resources more efficiently. They can offer a direct network connection to receive incoming tuples and support dataflow processing, delivering both high processing throughput and low latency. However, existing FPGA-based solutions offer only incremental stream aggregations, which is not applicable for queries with holistic aggregation functions which require keeping explicitly all incoming values in a single window before computing an aggregation function. Moreover, existing FPGA solutions only relied on on-ship memories and so could support only smaller problem sizes, i.e., Window-Size $\times$ Number of Keys. This thesis bridges this gap in existing literature by proposing Single-SWAG using FPGAs for supporting holistic aggregation functions and achieving high processing throughput and low latency. This thesis deals with the tremendous memory pressure in the Single-SWAG approach, and proposes efficient memory management techniques to alleviate this bottleneck.

6.1 Summary

Chapter 2 describes the first *tuple-based* Single-SWAG engine using FPGA [9]. Tuple-based windows always contain (and are slid by) a fixed number of tuples. They are suitable for applications with fixed data rates and have a fixed memory footprint. Our approach is implemented in a Maxeler N-series Dataflow Engine (DFE) and uses deep pipelining to provide high processing throughput. The DFE has a direct network connection to feed incoming tuples as well as direct access to DRAM offering ultra low end-to-end latency and large problem sizes. It is able to implement challenging realistic queries of any holistic, distributive or algebraic function and support large window sizes and number of keys.

Chapter 3 describes Time-SWAD, the first FPGA-based *time-based* Single-SWAG engine [26]. Time-based SWAG allows for varying data-arrival rates which naturally fits the time-series data produced by most Internet-of-Things (IoT) devices. Never-

theless, the number of tuples contained in a time-based window can vary, making the memory and compute resources needed to produce the aggregation result unpredictable. This unbounded number of tuples in a time-based sliding window is facilitated by a flexible circular buffer that stores the window values. In addition, this buffer can be expanded dynamically with one or more unused identical buffers originally meant for other keys. Thereby, time-based windows of varying size can be stored. Second, the memory pressure of single windows for skewed-key distributions, caused by their need to store all incoming data, is alleviated with a caching scheme. Similar to the tuple-based Single-SWAG DFE in Chapter 2, Time-SWAD design is dataflow, matching well the stream processing characteristics, and is implemented in a Maxeler N-series FPGA card. The DFE has direct network and DRAM interfaces supporting holistic functions, large number of keys and sufficient volumes of stored values. However, DRAM bandwidth is limited and the caching mechanism enables to merge multiple requests to the same DRAM location, especially for skewed-key distributions, enabling high throughput.

Chapter 4 describes Multi-level Queues (MLQs), a specialized memory hierarchy for stream aggregation [15]. Instead of using only DRAM as in the previous chapters to accommodate the single window per key, MLQs employ multiple memory levels available in the FPGA platform to form logical queues that offer on-chip SRAM (BRAM) bandwidth for window updates and DRAM capacity. This ensures that the window is always updated at the highest speed and mitigates the need for expensive read-modify-write operations as in DRAM-only designs. Then, when the window advances, the contents of the entire window are read utilizing the aggregate bandwidth of all memory levels. This chapter describes an analytical model of MLQ and a method to automatically generate its configuration for a problem at hand. A 3-level MLQ design is implemented in a Maxeler DFE to support Single-SWAG and is evaluated and compared with related work.

Chapter 5 describes Streamzip, a dataflow stream aggregation engine that is able to compress the sliding windows [32]. The MLQ system in Chapter 4 mainly mitigates the memory pressure due to the window-update step in Single-SWAG. StreamZip, built on top of MLQs, improves the processing throughput of Single-SWAG by mitigating the memory bandwidth bottleneck posed by the large volume of window-aggregation traffic using compression. Streamzip addresses a number of concurrency control challenges to integrate a compressor in the stream aggregation pipeline and supports both lossy and lossless compression algorithms with diverse characteristics, and is applied to both fixed- and floating- point numbers.

6.2 Contributions

This thesis proposes Single Sliding-Window Stream Aggregation (Single-SWAG) at up to line-rate throughput and low latency supporting any arbitrary functions and large problem sizes. This is achieved using dataflow processing in reconfigurable hardware with smart window management for both *tuple-* and *time-* based windowing policies and by mitigating the memory pressure of the data-intensive Single-SWAG using *multi-level queues* and *compressed* sliding-windows. To this end, this thesis describes the first FPGA-based *Single-SWAG Dataflow Engine (DFE)*, which:

- Supports both windowing policies, namely, *tuple-based* and *time-based*;

- Supports multiple challenging realistic streaming queries with holistic and arbitrary user-defined aggregation functions, as well as distributive and algebraic ones;
- Supports large number of concurrently active keys and large window sizes;
- Utilizes deep pipelining to achieve 1-2 orders of magnitude higher processing throughput than a state-of-the-art stream processing software system at up to 2× lower power cost; and
- Utilizes direct network connection to feed incoming tuples as well as direct access to DRAM offering ultra low latency, which is at least 4 orders of magnitude lower than CPU and GPU solutions.

For improving the performance of the window-update step in Single-SWAG DFE, the thesis proposes *Multi-level Queues (MLQs)*, a specialized memory hierarchy for stream aggregation, which:

- Utilize multiple memory levels to form logical queues that offer on-chip SRAM (BRAM) bandwidth for window updates and DRAM capacity;
- Compared to BRAM-only stream aggregation designs, MLQ supports 4 orders of magnitude larger problems;
- Compared to DRAM-only DFE designs, it achieves up to 8× higher throughput; and
- Offers 4 orders of magnitude lower aggregation latency, compared to competing GPU stream processing systems.

Finally, the overall performance of the Single-SWAG DFE is improved by reducing the volume of window-aggregation traffic using compression. To this end, this thesis proposes, *Streamzip*, a dataflow stream aggregation engine built on top of the MLQ design, which:

- Alleviates the memory pressure posed by window aggregation traffic, and improves performance as well as effective memory capacity by compressing sliding-windows;
- Supports both lossy and lossless compression algorithms with diverse characteristics, applied to both fixed- and floating- point numbers;
- Compared to designs without compression, achieves up to 7.5× higher throughput while reducing effective memory capacity by up to 5× for the Lossless deisgn; and
- Achieves up to 22× higher throughput while reducing effective memory capacity by up to 23× for the Lossy design.

6.3 Future Work

There are several directions for future research which can improve and complement the work presented here. In the following, we identify and list some of them:

Building Stream Processing Data Management System using FPGA-based Dataflow Engine: It would be interesting to build generic stream processing data management system with support for other stateless and stateful operators using FPGA-based Dataflow Engines. Examples of stateless operators include Map (the data streaming counterpart of the relational projection function), Filter (the data streaming counterpart of the relational select function) and Union operators. Examples of other stateful operators include the Join operator [16, 18, 33]. It would be interesting to evaluate the impact of the designs and ideas proposed for stream aggregation in this thesis on other resource intensive stateful operators like stream-joins [22, 63–67]. To this end, modular libraries for various stream processing operators can be implemented using similar dataflow-based FPGA designs. The exposed interfaces of the accelerator libraries would be used to combine and build stream processing pipelines and accelerate deployments in production.

Utilizing High Bandwidth Memory Subsystem: Another possible research direction is to evaluate the impact of the designs and ideas proposed in this thesis using other memory technologies like High Bandwidth Memories (HBM) [68, 69]. In contrast to traditional DRAM, 3D die-stacking in form of HBM compensates its lower clock frequency with wide busses and a high number of separate channels. However, this also requires data to be spread out over all channels to reach the full throughput [68]. As HBMs are becoming mainstream and less expensive, they can be part of the proposed MLQ hierarchy. It will be interesting to show the tradeoffs arising from HBM integration related to data movement and partitioning across the memory hierarchy and it's impact on processing throughput and latency of the stream processing pipeline.

Hardware-Software Co-design: All the designs proposed in this thesis have the FPGA working mostly in standalone mode with direct network and memory interfaces. However, for corner cases like hash table collisions and congestion management, it would be interesting to explore the impact of involving the software on the host side to provide dynamic bookkeeping capabilities and to work in tandem with the FPGA. This can also open up the possibility of the CPU taking the unconventional role of a specialized accelerator and the FPGA as the general purpose engine which can be especially useful for stream processing engines [70].

Bibliography

[1] H. C. Andrade, B. Gedik, and D. S. Turaga, *Fundamentals of stream processing: application design, systems, and analytics*. Cambridge University Press, 2014.

[2] K. Tangwongsan, M. Hirzel, and S. Schneider, "Low-latency sliding-window aggregation in worst-case constant time," in *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*. ACM, 2017, pp. 66–77.

[3] V. Gulisano, Y. Nikolakopoulos, D. Cederman, M. Papatriantafilou, and P. Tsigas, "Efficient data streaming multiway aggregation through concurrent algorithmic designs and new abstract data types," *ACM Transactions on Parallel Computing (TOPC)*, 2017.

[4] T. De Matteis, G. Mencagli, D. De Sensi, M. Torquati, and M. Danelutto, "Gasser: An auto-tunable system for general sliding-window streaming operators on gpus," *IEEE Access*, vol. 7, pp. 48 753–48 769, 2019.

[5] V. Gulisano, Y. Nikolakopoulos, I. Walulya, M. Papatriantafilou, and P. Tsigas, "Deterministic real-time analytics of geospatial data streams through scalegate objects," in *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '15. ACM, 2015, pp. 316–317.

[6] R. Mueller, J. Teubner, and G. Alonso, "Streams on wires: a query compiler for fpgas," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 229–240, 2009.

[7] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker, "No pane, no gain: efficient evaluation of sliding-window aggregates over data streams," *ACM SIGMOD*, vol. 34, no. 1, pp. 39–44, 2005.

[8] Y. Oge, M. Yoshimi, T. Miyoshi, H. Kawashima, H. Irie, and T. Yoshinaga, "An efficient and scalable implementation of sliding-window aggregate operator on fpga," in *International Symposium on Computing and Networking (CANDAR)*. IEEE, 2013, pp. 112–121.

[9] P. R. Geethakumari, V. Gulisano, B. J. Svensson, P. Trancoso, and I. Sourdis, "Single window stream aggregation using reconfigurable hardware," in *2017 International Conference on Field Programmable Technology (ICFPT)*, 2017, pp. 112–119.

[10] V. Gulisano, Z. Jerzak, S. Voulgaris, and H. Ziekow, "The debs 2016 grand challenge," in *ACM International Conference on Distributed and Event-based Systems*, ser. DEBS '16. ACM, 2016, pp. 289–292.

[11] V. Gulisano, Z. Jerzak, R. Katerinenko, M. Strohbach, and H. Ziekow, "The debs 2017 grand challenge," in *ACM International Conference on Distributed and Event-based Systems*, ser. DEBS '17. ACM, 2017, pp. 271–273.

[12] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh, "Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals," *Data mining and knowledge discovery*, vol. 1, no. 1, pp. 29–53, 1997.

[13] M. Najafi, K. Zhang, M. Sadoghi, and H.-A. Jacobsen, "Hardware acceleration landscape for distributed real-time analytics: Virtues and limitations," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2017, pp. 1938–1948.

[14] G. Pekhimenko, C. Guo, M. Jeon, P. Huang, and L. Zhou, "Tersecades: Efficient data compression in stream processing," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 307–320.

[15] P. R. Geethakumari and I. Sourdis, "A specialized memory hierarchy for stream aggregation," in *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, 2021, pp. 204–210.

[16] "Apache flink," https://flink.apache.org/, accessed: January 17, 2022.

[17] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *ACM symposium on operating systems principles*, 2013, pp. 423–438.

[18] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham *et al.*, "Storm@ twitter," in *ACM SIGMOD International Conference on Management of data*. ACM, 2014, pp. 147–156.

[19] A. Villalba, J. L. Berral, and D. Carrera, "Constant-time sliding window framework with reduced memory footprint and efficient bulk evictions," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 3, pp. 486–500, 2019.

[20] A. Koliousis, M. Weidlich, R. Castro Fernandez, A. L. Wolf, P. Costa, and P. Pietzuch, "Saber: Window-based hybrid stream processing for heterogeneous architectures," in *International Conference on Management of Data*. ACM, 2016, pp. 555–569.

[21] R. Mueller, J. Teubner, and G. Alonso, "Data processing on fpgas," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 910–921, 2009.

[22] Y. Oge, T. Miyoshi, H. Kawashima, and T. Yoshinaga, "A fast handshake join implementation on fpga with adaptive merging network," in *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, 2013, pp. 1–4.

[23] Y. Oge, M. Yoshimi, T. Miyoshi, H. Kawashima, H. Irie, and T. Yoshinaga, "Wire-speed implementation of sliding-window aggregate operator over out-of-order data streams," in *2013 IEEE 7th International Symposium on Embedded Multicore Socs*. IEEE, 2013, pp. 55–60.

[24] Y. Oge, M. Yoshimi, H. Kawashima, T. Miyoshi, H. Irie, and T. Yoshinaga, "Design and evaluation of a configurable query processing hardware for data streams," *IEICE TRANSACTIONS on Information and Systems*, vol. 98, no. 12, pp. 2207–2217, 2015.

[25] "Maxeler-n-series," http://bit.ly/2v1APVK, accessed: January 17, 2022.

[26] P. Ramakrishnan Geethakumari, V. Gulisano, P. Trancoso, and I. Sourdis, "Timeswad: A dataflow engine for time-based single window stream aggregation," in *2019 International Conference on Field-Programmable Technology (ICFPT)*, 2019, pp. 72–80.

[27] S. Iyer, R. R. Kompella, and N. McKeowa, "Analysis of a memory architecture for fast packet buffers," in *2001 IEEE Workshop on High Performance Switching and Routing*. IEEE, 2001, pp. 368–373.

[28] J. Garcia, J. Corbal, L. Cerda, and M. Valero, "Design and implementation of high-performance memory systems for future packet buffers," in *IEEE/ACM MICRO*, 2003.

[29] J. Garcia-Vidal, M. March, L. Cerda, J. Corbal, and M. Valero, "A dram/sram memory scheme for fast packet buffers," *IEEE Transactions on Computers*, vol. 55, no. 5, pp. 588–602, 2006.

[30] S. Iyer, R. R. Kompella, and N. McKeown, "Designing packet buffers for router linecards," *IEEE/ACM Transactions On Networking*, vol. 16, no. 3, p. 705–717, Jun. 2008.

[31] A. Mutter, "A novel hybrid sram/dram memory architecture for fast packet buffers," in *ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2009, p. 183–184.

[32] P. R. Geethakumari and I. Sourdis, "Streamzip: Compressed sliding-windows for stream aggregation," in *2021 International Conference on Field-Programmable Technology (ICFPT)*, 2021, pp. 203–211.

[33] V. Gulisano, R. Jiménez-Peris, M. Patiño-Martínez, C. Soriente, and P. Valduriez, "Streamcloud: An elastic and scalable data streaming system," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 12, pp. 2351–2365, 2012.

[34] S. Zhang, F. Zhang, Y. Wu, B. He, and P. Johns, "Hardware-conscious stream processing: A survey," *ACM SIGMOD Record*, vol. 48, no. 4, pp. 18–29, 2020.

[35] J. Fang, Y. T. Mulder, J. Hidders, J. Lee, and H. P. Hofstee, "In-memory database acceleration on fpgas: a survey," *The VLDB Journal*, vol. 29, no. 1, pp. 33–59, 2020.

[36] Z. István, G. Alonso, M. Blott, and K. Vissers, "A hash table for line-rate data processing," *ACM TRETS*, vol. 8, no. 2, p. 13, 2015.

[37] Z. István, D. Sidler, and G. Alonso, "Caribou: Intelligent distributed storage," *VLDB Endowment*, vol. 10, no. 11, 2017.

[38] I. Absalyamov, P. Budhkar, S. Windh, R. J. Halstead, W. A. Najjar, and V. J. Tsotras, "Fpga-accelerated group-by aggregation using synchronizing caches," in *Proceedings of the 12th International Workshop on Data Management on New Hardware (DaMoN)*, 2016, pp. 1–9.

[39] "Bob jenkins lookup3 hash function," http://bit.ly/2vU4cpm, accessed: January 17, 2022.

[40] A. Arasu, S. Babu, and J. Widom, "The cql continuous query language: semantic foundations and query execution," *The VLDB Journal*, vol. 15, no. 2, pp. 121–142, 2006.

[41] A. Kos, V. Ranković, and S. Tomažič, "Chapter four-sorting networks on maxeler dataflow supercomputing systems," *Advances in computers*, vol. 96, pp. 139–186, 2015.

[42] "Tcpreplay," http://tcpreplay.appneta.com/, accessed: January 17, 2022.

[43] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts, "Linear road: a stream data management benchmark," in *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*. VLDB Endowment, 2004, pp. 480–491.

[44] P. Tucker and D. Maier, "Exploiting punctuation semantics in data streams," in *Proceedings 18th International Conference on Data Engineering*. IEEE, 2002, p. 279.

[45] S. Gong, J. Li, W. Lu, G. Yan, and X. Li, "Shuntflow: An efficient and scalable dataflow accelerator architecture for streaming applications." in *Design Automation Conference (DAC)*, 2019, pp. 194–1.

[46] L. Woods, Z. István, and G. Alonso, "Ibex: An intelligent storage engine with support for advanced sql offloading," *VLDB Endowment*, vol. 7, no. 11, pp. 963–974, 2014.

[47] J. Wilkes, "Google cluster data," http://bit.ly/1A38mfR, last accessed: January 17, 2022.

[48] H. Ziekow and Z. Jerzak, "The debs 2014 grand challenge," in *8th ACM DEBS*, vol. 14, 2014.

[49] S. A. Fahmy, P. Y. Cheung, and W. Luk, "Novel fpga-based implementation of median and weighted median filters for image processing," in *Int'l Conf. on FPL*. IEEE, 2005, pp. 142–147.

[50] J. Gantz and D. Reinsel, "The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east," December 2012.

[51] J. Hoozemans, J. Peltenburg, F. Nonnemacher, A. Hadnagy, Z. Al-Ars, and H. P. Hofstee, "Fpga acceleration for big data analytics: Challenges and opportunities," *IEEE Circuits and Systems Magazine*, vol. 21, no. 2, pp. 30–47, 2021.

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

[53] "Qdr sram," https://intel.ly/2GxJl6m.

[54] W. F. Tichy, "Rcs—a system for version control," *Software: Practice and Experience*, vol. 15, no. 7, pp. 637–654, 1985.

[55] M. Farrens and A. Park, "Dynamic base register caching: A technique for reducing address bus width," in *Proceedings of the 18th Annual International Symposium on Computer Architecture*, ser. ISCA '91, 1991, p. 128–137.

[56] P. Ratanaworabhan, J. Ke, and M. Burtscher, "Fast lossless compression of scientific floating-point data," in *Data Compression Conference (DCC'06)*. IEEE, 2006, pp. 133–142.

[57] T. Pelkonen, S. Franklin, J. Teller, P. Cavallaro, Q. Huang, J. Meza, and K. Veeraraghavan, "Gorilla: A fast, scalable, in-memory time series database," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1816–1827, 2015.

[58] S. Di and F. Cappello, "Fast error-bounded lossy hpc data compression with sz," in *2016 ieee International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2016, pp. 730–739.

[59] A. Eldstål-Ahrens and I. Sourdis, "Memsz: Squeezing memory traffic with lossy compression," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 17, no. 4, pp. 1–25, 2020.

[60] Q. Xiong, R. Patel, C. Yang, T. Geng, A. Skjellum, and M. C. Herbordt, "Ghostsz: A transparent fpga-accelerated lossy compression framework," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2019, pp. 258–266.

[61] H. Miao, M. Jeon, G. Pekhimenko, K. S. McKinley, and F. X. Lin, "Streambox-hbm: Stream analytics on high bandwidth hybrid memory," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 167–181.

[62] G. Chrysos, O. Papapetrou, D. Pnevmatikatos, A. Dollas, and M. Garofalakis, "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)*, 2019, pp. 278–285.

[63] J. Teubner and R. Mueller, "How soccer players would do stream joins," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, 2011, pp. 625–636.

[64] R. J. Halstead, I. Absalyamov, W. A. Najjar, and V. J. Tsotras, "Fpga-based multithreading for in-memory hash joins." in *7th Biennial Conference on Innovative Data Systems Research (CIDR)*. Citeseer, 2015.

[65] C. Kritikakis, G. Chrysos, A. Dollas, and D. N. Pnevmatikatos, "An fpga-based high-throughput stream join architecture," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2016, pp. 1–4.

[66] Z. F. Eryilmaz, A. Kakaraparthy, J. M. Patel, R. Sen, and K. Park, "Fpga for aggregate processing: The good, the bad, and the ugly," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2021, pp. 1044–1055.

[67] M. Najafi, M. Sadoghi, and H.-A. Jacobsen, "Scalable multiway stream joins in hardware," *IEEE Transactions on Knowledge and Data Engineering*, vol. 32, no. 12, pp. 2438–2452, 2019.

[68] P. Holzinger, D. Reiser, T. Hahn, and M. Reichenbach, "Fast hbm access with fpgas: Analysis, architectures, and applications," in *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2021, pp. 152–159.

[69] K. Kara, C. Hagleitner, D. Diamantopoulos, D. Syrivelis, and G. Alonso, "High bandwidth memory on fpgas: A data analytics perspective," in *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2020, pp. 1–8.

[70] G. Alonso, T. Roscoe, D. Cock, M. Owaida, K. Kara, D. Korolija, Z. Wang *et al.*, "Tackling hardware/software co-design from a database perspective," in *Proceedings of the 6th biennial Conference on Innovative Data Systems Research (CIDR), Amsterdam, Netherlands, January 2020.*, 2020.