Wire-Speed Implementation of Sliding-Window Aggregate Operator over Out-of-Order Data Streams

journal or publication title	IEEE 7th International Symposium on Embedded Multicore/Many-core SoCs (MCSoC-13)
page range	55-60
year	2013-09
URL	http://id.nii.ac.jp/1438/00001920/

Wire-Speed Implementation of Sliding-Window Aggregate Operator over Out-of-Order Data Streams

Yasin Oge*, Masato Yoshimi*, Takefumi Miyoshi†, Hideyuki Kawashima‡, Hidetsugu Irie*, and Tsutomu Yoshinaga*
* Graduate School of Information Systems, The University of Electro-Communications, Japan
1-5-1 Chofugaoka, Chofu-shi, Tokyo 182-8585, Japan
E-mail: oge@comp.is.uec.ac.jp, {yoshimi,irie,yosinaga}@is.uec.ac.jp
† e-trees.Japan, Inc.,
Daiwa Building 3F, 2-9-2 Oowada-cho, Hachioji, Tokyo, 192-0045
E-mail: miyoshi@e-trees.jp
‡ University of Tsukuba, Japan
1-1-1 Tennodai, Tsukuba, Ibaraki 305-8577, Japan
E-mail: kawasima@cs.tsukuba.ac.jp

***Abstract*—This paper shows the design and evaluation of an FPGA-based accelerator for sliding-window aggregation over data streams with out-of-order data arrival. We propose an order-agnostic hardware implementation technique for windowing operators based on a one-pass query evaluation strategy called Window-ID, which is originally proposed for software implementation. The proposed implementation succeeds to process out-of-order data items, or tuples, at wire speed due to the simultaneous evaluations of overlapping sliding-windows. In order to verify the effectiveness of the proposed approach, we have also implemented an experimental system as a case study. Our experiments demonstrate that the proposed accelerator with a network interface achieves an effective throughput around 760 Mbps or equivalently nearly 6 million tuples per second, by fully utilizing the available bandwidth of the network interface.**

I. Introduction

Nowadays, an increasing number of applications deal with continuous data streams. For example, many data processing tasks, such as financial analysis and high-speed network monitoring, are required to handle a huge amount of data with certain time restrictions. Real-time processing is essential for systems such as *Data Stream Management Systems* (DSMSs) which process unbounded and continuous input streams, executing *continuous queries* over data streams [1]–[3].

One of the key challenges for DSMSs is an efficient support for *window aggregation*. It is a common approach for a DSMS that subsequence of data stream elements (hereafter *tuples*) is defined as a *window*. In other words, windows decompose a data stream into possibly overlapping subsets of tuples (*i.e.,* each tuple belongs to multiple windows). After that, according to a given query, window aggregate operators repeatedly calculate aggregate functions such as COUNT, SUM, AVERAGE, MIN, and MAX for each window.

Motivating Issue. Mueller *et al.* [7] propose an implementation technique for a sliding-window aggregate query on an FPGA. Their implementation relies on an implicit assumption about the physical order of incoming tuples, that is to say, tuples arrive in correct order at the windowing operator. Obviously, this assumption simplifies the definition and implementation of sliding windows; however, it does not always fit into a realistic setting where some degree of disorder (*i.e.,* out-of-order arrival of tuples) might be expected.

It is mentioned in [6] that previous works on data streams commonly model a data stream as an unbounded sequence of tuples arriving in order of some timestamp-like attribute; however, disorder naturally occurs in real-world stream systems. This means that, in reality, we cannot always assume all tuples to be ordered by their timestamp values when they arrive in a DSMS. For example, input tuples arriving over a network from remote sources may take different paths with different delays. As a result, some tuples may arrive out of sequence according to their timestamp values.

Unfortunately, Mueller *et al.* [7] do not address or discuss the issue regarding out-of-order arrival of tuples. To the best of our knowledge, it is still an open question how to design and implement an efficient hardware accelerator for sliding-window aggregate operator that can handle out-of-order arrival of tuples. In this paper, we address the problem and present an improved implementation for a sliding-window aggregate query on an FPGA.

Our Contributions. This paper proposes an order-agnostic implementation of a sliding-window aggregate query on an FPGA, based on a one-pass query evaluation strategy called the *Window-ID* (WID) [5]. With the proposed method, we can process out-of-order tuples at wire speed due to the one-pass query evaluation strategy and simultaneous evaluations of overlapping sliding-windows by taking advantage of the hardware parallelism. The proposed implementation can handle disorder by utilizing punctuations [9]. It is stated in [5] that WID does not require a specific type of assumption about the physical order of tuples in a data stream and can process out-of-order tuples as they arrive without sorting them into the "correct" order. Since the proposed implementation is based on WID approach, it can also process input tuples on the fly without reordering them into the correct order.

The rest of the paper is organized as follows. Section II briefly reviews related work. Section III provides design concepts underlying the proposed approach. Section IV describes a motivating example. Section V introduces the proposed hardware implementation. Section VI evaluates the proposed implementation with some experimental results. Finally, section VII concludes the paper by summarizing the results.

II. Related Work

An important work related to the present study is *Glacier* [7], which is a compiler that provides a library of components and transforms continuous queries into logic circuits to be implemented on an FPGA. Glacier supports selection, aggregation, grouping, and windowing operators. It can compile sliding-window aggregate queries since both windowing and aggregation operators are provided by the library, and logic circuits are generated by composing library components on an operator-level basis. Glacier, however, does not address the issue regarding disordered data.

Another important work is *W*indow-*ID* (WID) [5]. WID is proposed for a software-based implementation of order-agnostic window aggregation. It is stated in [6] that WID provides a method to implement window aggregate queries in an order-agnostic way, using punctuations that indicate the end of the windows.

Informally, a punctuation is a kind of tuple which contains control information, and it is embedded in a data stream. The punctuation can be used to indicate that no more tuples having certain attribute values will be seen in the stream [5]. Therefore, punctuations are useful to unblock some blocking operators such as group-by and aggregation. The formal definition and further details of the punctuation are found in [9].

Abadi *et al.* [1] classify types of operators as *order-agnostic* or *order-sensitive*. Order-agnostic operators can always process tuples in the order in which they arrive whereas order-sensitive operators can only be guaranteed to execute with finite buffer space if they can assume some ordering over their input streams [1].

Slack [1] defines an upper bound on the degree of disorder that can be handled by an order-sensitive operator. Aurora [1] assumes some ordering (potentially with bounded disorder) over input streams. Any tuple arriving after its corresponding period specified by a slack parameter is discarded. In Aurora, the slack parameter is used to specify the number of tuples to be stored and sorted before an order-sensitive operator processes input tuples. Aurora classifies window aggregation as an order-sensitive operation. In Aurora, therefore, aggregate operators require buffering and reordering of tuples before computation to handle disorder.

III. Design Concept

Glacier implements a windowing operator as an order-sensitive operator and does not discuss the issue regarding out-of-order arrival of tuples. In order to address the problem, this paper proposes an alternative implementation technique for windowing operators. The proposed implementation follows the same approach as WID [5] to handle disorder. In other words, aggregation operation is order-agnostic, and punctuations are used to unblock window-aggregate operators.

On the other hand, WID is proposed for a software-based implementation. The main interests of WID [5] are to calculate window aggregates with the one-pass evaluation strategy and to handle disorder by using punctuations. However, hardware-based implementation of order-agnostic window aggregation is neither provided nor discussed in [5].

Contrary to the software-based implementation proposed in [5], this paper presents hardware-based implementation which handles multiple windows with a single clock cycle. The proposed implementation instantiates multiple window-aggregation modules by taking advantage of hardware parallelism. Upon arrival of a new tuple, each of the window-aggregation modules can simultaneously evaluate the tuple within the same clock cycle. This is the main difference between software-based WID [5] and our proposed approach.

The number of the window-aggregation modules is determined by using window parameters ($RANGE$ and $SLIDE$) and a slack [1] specification. As mentioned before, Aurora [1] uses a slack parameter to determine the number of tuples to be buffered and reordered before aggregation. The proposed approach, however, relies on punctuations to handle disorder, and the slack parameter is used to calculate the number of the window-aggregation modules required to be instantiated. This is a significant difference between the approach adopted in this paper and that of Aurora.

IV. Motivating Application

Glacier [7] demonstrates how to implement a window aggregate query on an FPGA. The implementation of the query (Query Q_3 of [7]) includes a windowing operator which implicitly relies on the arrival sequence of input tuples. Contrary to Glacier, the proposed approach permits windowing on any attribute, allowing a bounded disorder of the tuples. This work focuses on the same query as a case study and shows how to implement the query in an order-agnostic manner. It should be emphasized that the proposed approach is general enough to apply a wide range of window aggregate queries which include the algebraic aggregate functions [4] considered in [7] (*i.e.,* COUNT, SUM, AVERAGE, MIN, and MAX).

We assume the same financial application as [7]. Our approach, however, requires an explicit timestamp attribute to define windows over an input stream. Instead of a sequence number attribute, a timestamp attribute has been added. The schema of the stream Trades [7] is redefined as follows:

```
CREATE INPUT STREAM Trades (
  Symbol string(4),  -- valor symbol
  Price  int,        -- stock price
  Volume int,        -- trade volume
  Time   int)        -- timestamp
```

An input tuple consists of four attributes each of which is represented as a 32-bit value. Based on the definition of

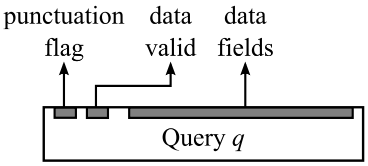

Fig. 1. Wiring Interface (figure cited from [8]).

window semantics [5], we can rewrite the sliding-window aggregate query (Query Q_3 of [7]) as follows.

Query Q_3: "Count the number of trades of UBS (Union Bank of Switzerland) shares for the past 10 minutes (600 seconds) and update the result every 1 minute (60 seconds)."

```
SELECT Time, count(*) AS Number
  FROM Trades [RANGE 600 seconds
               SLIDE 60 seconds
               WATTR Time]
 WHERE Symbol = "UBSN"
```

In Query Q_3, $WATTR$ indicates the windowing attribute (*i.e.,* Time) over which $RANGE$ and $SLIDE$ are specified [5]. The details of the implementation of Query Q_3 are provided in the following section.

V. Implementation Details

A. Wiring Interface

Following the notation of [7], Fig. 1 (cited from [8]) shows the black box view of a hardware implementation for a query q. In Fig. 1, *punctuation flag* or *data valid* is a one-bit signal which indicates the presence of a punctuation or a tuple, respectively. In addition, *data fields* represent n-bit-wide data which are regarded as a set of n parallel wires. For example, datum on the parallel wires is considered as a punctuation when the *punctuation flag* is asserted (*i.e.,* set to logic "1"). Similarly, the data lines are regarded as a valid tuple when the *data valid* is asserted.

B. Hardware Execution Plan

We implement a 4-stage pipeline hardware for Query Q_3 as illustrated in Fig. 2. The gray-shaded boxes in Fig. 2 represent flip-flop registers which store intermediate results at the end of each stage. These registers can be regarded as pipeline registers, and each stage of the pipeline can use the result of the previous stage. It should be also mentioned that each stage requires only one clock cycle to complete. The arrows in Fig. 2 indicate the connections between the pipeline stages. According to the notation of [7], the *data fields* do not represent the order of each field. Note that the label "$*$" means "all of the remaining fields" in the *data fields*.

1) Selection Operation: The beginning two stages of Fig. 2 correspond to a selection operation. In **Stage 1**, *Symbol* field of the data bus is compared to a constant value ("UBSN") which is specified in the WHERE expression of Query Q_3 (indicated as $\boxed{=}$ in Fig. 2). At the same time, the result of the comparison is labeled as a one-bit *is_equal* flag and added to the data bus. In **Stage 2**, a logical AND gate (indicated as $\boxed{\&}$ in Fig. 2) computes whether an input tuple is valid or not. If the tuple should be discarded, the *data valid* flag is negated (*i.e.,* set to logic "0") for the next pipeline stage. Actually, these two stages are implemented based on the approach proposed in [7], providing the same functionality as the beginning two stages of Query Q_1 in [7]. The main difference, however, is the presence of the *punctuation flag* field which is required for **Stage 3**. It is stated in [5] that some operators, such as selection, simply pass punctuations through to the next operator in a query plan. **Stage 1** and **Stage 2** meet the above requirement since *punctuation flag* field is directly connected to the next stage of the pipeline as shown in Fig. 2.

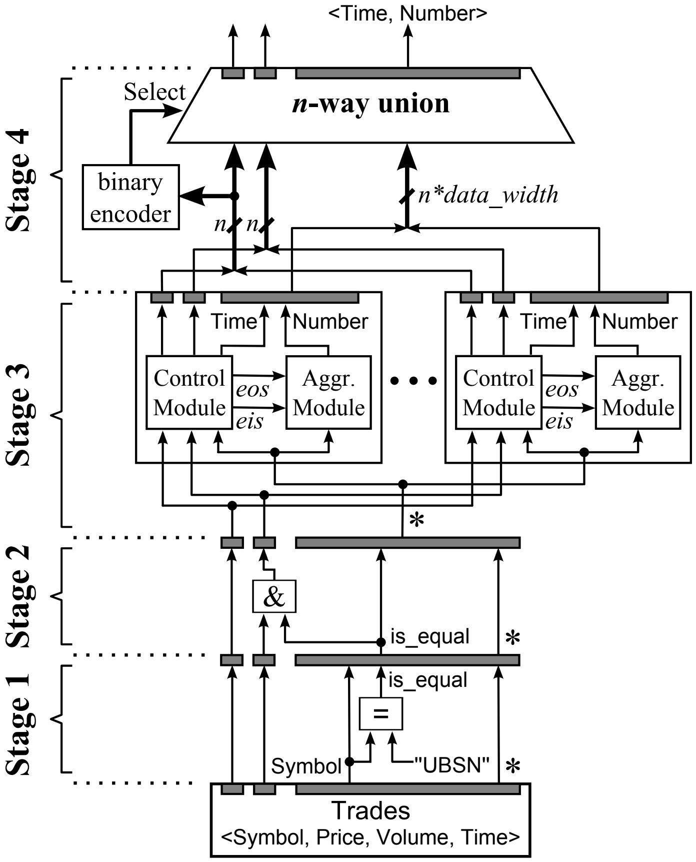

Fig. 2. Hardware execution plan for Query Q_3.

2) Windowing and Aggregation: The next step is **Stage 3** of the pipeline which corresponds to windowing and aggregation operators. In **Stage 3**, a number of window-aggregation modules are instantiated as shown in Fig. 2. They provide sliding-window functionality and can concurrently compute aggregate functions. The number of window-aggregation modules to be instantiated is calculated by using $RANGE$, $SLIDE$, and $SLACK$ parameters (see the following Equation 1 and 2). The detail about the calculation of N_{WIN} is discussed in [8].

$$N_{\text{WIN}} = \left\lceil \frac{RANGE}{SLIDE} \right\rceil + x, \text{ where } x \in \mathbb{Z}^{+} \tag{1}$$

$$x \geq \frac{SLACK + RANGE}{SLIDE} - \left\lceil \frac{RANGE}{SLIDE} \right\rceil \tag{2}$$

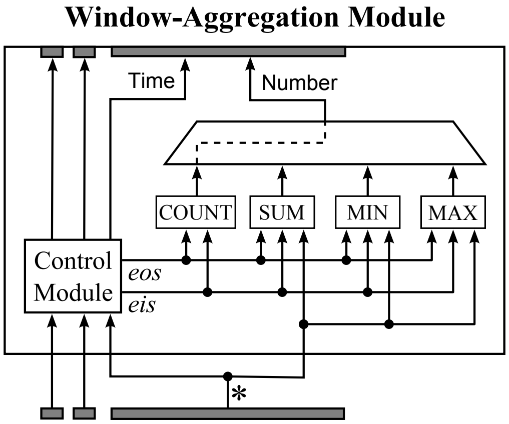

Fig. 3. Block diagram of a window-aggregation module.

Aggregation Module. Each window instance consists of *aggregation module* and *control module* as shown in **Stage 3** of Fig. 2. A more detailed block diagram of a single window-aggregation module is depicted in Fig. 3. It is stated in [7] that algebraic aggregate functions such as COUNT, SUM, AVERAGE, MIN, and MAX can be implemented in a straightforward fashion on an FPGA. Glacier [7] supports the above five aggregate functions, and in this work, we also focus on the same aggregate operators. In fact, AVERAGE can be obtained with the combination of two aggregate values: SUM and COUNT. Therefore, we implement other four aggregate operators as shown in Fig. 3. Since Query Q_3 requires $\mathrm{count}(*)$ function, the result of the COUNT operator is selected as the output value (indicated as the broken line in Fig. 3).

The aggregate operator incrementally computes aggregate value and only stores the current (partial) result of the aggregation. It requires two control signals, enable input stream (eis) and end of stream (eos), to maintain the aggregate value. These signals are provided by the control module as illustrated in Fig. 3. The eis signal indicates whether or not data on the *data fields* should be considered as a valid tuple for the current window. Whenever eis is asserted, the aggregation operator accepts input tuple and records its contribution to the partial result. If eis is negated, the aggregation operator simply ignores the input data and waits for the next tuple to arrive. The other signal, eos, indicates whether an input stream reaches the end of the current window. When eos is asserted, it means that the current window is no longer active, and the aggregate operator should reset its internal state.

Control Module. Each control module maintains its own window states and provides two control signals (*i.e.,* eis and eos) to the aggregation module. The control module maintains window states by updating its internal registers called win_{begin} and win_{end}. These registers represent the beginning and the end of the current window, respectively. The control module uses win_{begin} and win_{end} registers to generate the two control signals, eis and eos. Details about how to maintain these registers and to generate the control signals are provided in Algorithm 1 and Algorithm 2, respectively.

Algorithm 1 Maintain window states (win_{begin} and win_{end})

```
State Registers:
$win_{\mathrm{begin}}(i)$: the beginning of the $i$-th window instance
$win_{\mathrm{end}}(i)$: the end of the $i$-th window instance
Initialization:
for all $i$ such that $1 \leq i \leq N_{\mathrm{WIN}}$ do
  $win_{\mathrm{begin}}(i) \leftarrow WATTR_{\mathrm{start}} + (i-1) \times SLIDE$
  $win_{\mathrm{end}}(i) \leftarrow WATTR_{\mathrm{start}} + (i-1) \times SLIDE + RANGE$
end for
Synchronous Update:
for all $i$ such that $1 \leq i \leq N_{\mathrm{WIN}}$ do
  for each clock cycle do
    if punctuation flag is asserted and
    $WATTR \geq win_{\mathrm{end}}(i)$ then
      $win_{\mathrm{begin}}(i) \leftarrow win_{\mathrm{begin}}(i) + N_{\mathrm{WIN}} \times SLIDE$
      $win_{\mathrm{end}}(i) \leftarrow win_{\mathrm{end}}(i) + N_{\mathrm{WIN}} \times SLIDE$
    end if
  end for
end for
```

Algorithm 2 Generate asynchronous signals (eis and eos)

```
Asynchronous Signals:
$eis(i)$: input enable signal for the $i$-th window instance
$eos(i)$: output enable signal for the $i$-th window instance
Asynchronous Update:
for all $i$ such that $1 \leq i \leq N_{\mathrm{WIN}}$ asynchronously do
  if punctuation flag is negated then
    negate $eos(i)$ signal
    if data valid is asserted and
    $win_{\mathrm{begin}}(i) \leq WATTR < win_{\mathrm{end}}(i)$ then
      assert $eis(i)$ signal
    else
      negate $eis(i)$ signal
    end if
  else {punctuation flag is asserted}
    negate $eis(i)$ signal
    if $WATTR \geq win_{\mathrm{end}}(i)$ then
      assert $eos(i)$ signal
    else
      negate $eos(i)$ signal
    end if
  end if
end for
```

Algorithm 1 describes how to initialize and update the two registers, win_{begin} and win_{end}. Since the windowing attribute (*i.e.,* $WATTR$) of Query Q_3 is defined as $TIME$, $WATTR_{\mathrm{start}}$ is equivalent to $TIME_{\mathrm{start}}$ which indicates the start time of the execution of the query. Initialization or update operation described in Algorithm 1 can be completed in one clock cycle. All of the control modules concurrently perform the same operation on each cycle in a synchronous manner.

TABLE I
SPECIFICATIONS OF XC6VLX240T-1

# of Slice Registers	301,440
# of Slice LUTs	150,720
# of Slices	37,680
# of BRAM (36Kbit)	416
# of DSP48	768

TABLE II
RESOURCE USAGE AND CLOCK FREQUENCY AGAINST DIFFERENT SIZES OF SLIDING WINDOWS

	Size of the Time-based Sliding Window (*i.e.*, $RANGE$)					
	10 min	20 min	30 min	40 min	50 min	60 min
# of window-aggregation modules	11	21	31	41	51	61
# of Slice Registers	1,855	3,442	4,975	6,453	8,095	9,594
# of Slice LUTs	1,764	3,342	5,011	6,615	7,844	9,241
# of Occupied Slices	663	1,153	1,550	2,073	2,650	3,264
Maximum clock frequency (MHz)	160	157	157	158	157	157

Algorithm 2 describes how to generate the two control signals, eis and eos. It is important to emphasize that the implementation of eis and eos signals is fully asynchronous, which means that the aggregation module can use these signals within the same clock cycle as soon as they are generated. In other words, all of the operations performed in a window-aggregation module can be completed in a single clock cycle.

3) Union Operation: It is stated in [7] that, from a data flow point of view, the task of an algebraic union operator is to merge the outputs of several source streams into a single output stream. As shown in Fig. 2, the n-way union operator merges the outputs of n window-aggregation modules and generates a single result stream.

The implementation of the union operator is based on a multiplexer component. According to a select signal, the multiplexer component transfers the result of i-th window-aggregation module to the output registers of **Stage 4**. As illustrated in Fig. 2, the select signal is provided by a binary encoder component. It should be also mentioned that **Stage 4** requires only one clock cycle to complete its operation.

Glacier [7] evaluates the complexity and performance of the resulting circuits in terms of *latency* and *issue rates*. Issue rate is defined as the number of tuples that can be processed per cycle. The overall latency and the issue rate of the proposed implementation are 4 cycles and 1 tuple/cycle, respectively.

VI. EVALUATION

The proposed design is implemented on a Virtex-6 FPGA (XC6VLX240T-1) included in the Xilinx ML605 Evaluation Kit. The specification of the FPGA used to implement the design is given in Table I. Xilinx ISE 14.4 is used as an FPGA development environment during the implementation process (*i.e.,* synthesis, map, and place & route).

A. Resource Utilization and Performance

In order to evaluate the resource utilization and performance of the proposed design, Query Q_3 is implemented with different sizes of sliding windows. The $RANGE$ of a window is increased from 10 minutes to 60 minutes, by increments of 10 (*i.e.,* a total of six different configurations). It should be also noted that all of the implemented queries have the same $SLIDE$ parameter as Query Q_3 (*i.e.*, 60 seconds). In addition, $SLACK$ value is also assumed 60 seconds for all configurations. Finally, the query is synthesized with a timing constraint of 6.37 ns for each configuration, which yields the target clock frequency of 157 MHz.

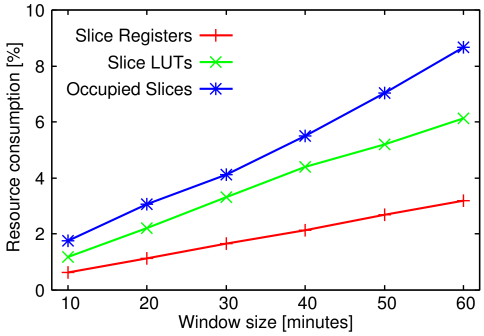

Fig. 4. Overall resource consumption as a percentage of the total available resources on a Xilinx XC6VLX240T-1 FPGA device.

1) Resource Utilization: Overall resource consumption is shown in Fig. 4. The x-axis of Fig. 4 represents the size of the time-based sliding window (*i.e.*, $RANGE$) from 10 to 60 minutes. The y-axis of the same figure indicates the resource consumption as a percentage of the total available resources on a Xilinx XC6VLX240T-1 FPGA device. As shown in Fig. 4, all three graphs (*i.e.*, Slice Registers, Slice LUTs, and Occupied Slices) are almost linearly increased with increasing window size, as expected. The increase in window size results in a higher $\frac{RANGE}{SLIDE}$ ratio. This implies an increase in the number of window-aggregation modules (*i.e.,* N_{WIN}, recall from Equation 1). This is the main reason for the increased resource utilization. It should be also emphasized that a relatively small percentage of the overall FPGA resources is required to implement the query. For example, when the size of window is 10 minutes, slice usage is particularly low (less than 2%). Even if the size of window is increased up to 60 minutes, overall slice utilization is still less than 9%.

Table II shows the hardware resource usage and the maximum clock frequency of the implemented query for each window size. The number of window-aggregation modules instantiated can be easily calculated by Equation 1 and Equation 2, using $RANGE$, $SLIDE$, and $SLACK$ parameters. The resource usage is measured in terms of the number of slice registers, the number of slice LUTs (Look-Up Tables), and the number of occupied slices. The clock frequency is obtained from post-place & route static timing report, which is provided by Xilinx's Timing Analyzer tool.

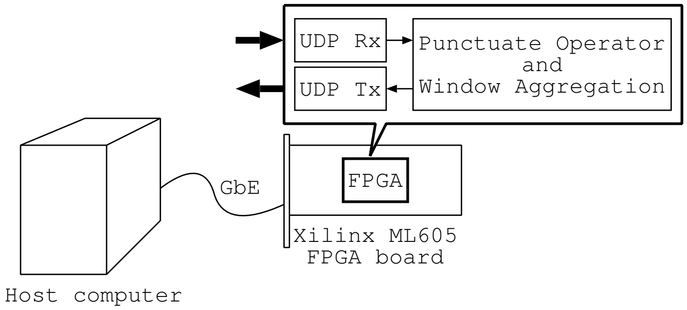

Fig. 5. Overview of the Experimental System.

2) Performance Evaluation: As shown in Table II, each implementation achieves the target clock frequency of 157 MHz. Equivalently, this means that all implementations meet the timing constraint of 6.37 ns. Since the issue rate of the implemented queries is equal to 1 tuple/cycle, the proposed implementation can process 157 million tuples per second for different sizes of windows. We can calculate the peak throughput by multiplying the data width of an input tuple by the clock frequency. Recall that the data width of a tuple is 128 bits; therefore, multiplying 157 million tuples/s by 128 bits/tuple yields 20,096 Mbps. Thus, the peak throughput can be estimated at 20 Gbps. As for latency, recall that the latency of the implemented queries is equal to 4 cycles, and the clock period is 6.37 ns if we assume a clock rate of 157 MHz. Hence, multiplying 4 by 6.37 ns yields 25.48 ns.

These data lead us to the conclusion that the proposed approach can accomplish both high throughput (over 150 million tuples per second) and low latency (the order of a few tens of nanoseconds) which are essential for stream processing systems to handle a huge volume of data for real-time applications.

B. Experimental Measurement

An overview of the experimental system is depicted in Fig. 5. The experimental system consists of the ML605 FPGA board and a host computer which are directly connected by a dedicated Gigabit Ethernet cable (indicated as “GbE” in Fig. 5). To simulate a disordered input stream, we implement a data generator to produce an input stream with bounded disorder. The data generator on the host computer first randomly generates input tuples in non-decreasing order with respect to their timestamp attribute. After that, the positions of the tuples are randomized in such a way that no tuples are to be late or out-of-order more than 60 seconds in the stream.

We measured the number of clock cycles elapsed from when the first tuple arrived at the UDP Rx module until the last result was transferred from the UDP Tx module. For each configuration given in Table II, we calculated the maximum throughput achieved by the experimental system, using the measured values. It should be noted that all results generated by the query circuit have been verified by the host computer. This has been confirmed by comparing expected results with those sent from the UDP Tx module. In our experiments, we obtained exactly the same results as those expected. This means that the proposed implementation can properly handle out-of-order tuples.

Results of the experiments show that the proposed implementation achieves an effective throughput up to around 760 Mbps, which is the upper bound of the available bandwidth that the network interface (*i.e.,* the UDP Rx module) could handle. This is equivalent to nearly 6 million tuples per second, which means that the proposed implementation can process significantly high tuple rates at wire speed. Furthermore, we have also performed experiments on other aggregation functions, such as SUM, MIN, and MAX, and obtained almost the same performance as Query Q_3 (*i.e.,* around 760 Mbps and nearly 6 million tuples/s).

VII. Conclusions

In this paper, we have proposed a design and implementation of an FPGA-based accelerator for sliding-window aggregates over disordered data streams. With the proposed approach, a sliding-window query can be implemented on an FPGA as an order-agnostic operator, which can process input tuples in their arrival order without sorting them into the “correct” order. The proposed accelerator utilizes punctuations, and this significantly reduces the input-to-output latency because there is no need to buffer and reorder incoming tuples. Our experiments demonstrate that nearly 6 million tuples can be processed per second directly from the network interface. To the best of our knowledge, this is the first paper that proposes design and implementation of a punctuation-aware sliding-window aggregate operator on an FPGA device.

One direction for future work is to conduct an experiment on the scalability of the proposed approach, especially at higher $\frac{RANGE}{SLIDE}$ ratios (more than 60). Another direction is to address multi-query execution and its optimization.

References

[1] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. B. Zdonik, “Aurora: a new model and architecture for data stream management,” *VLDB J.*, vol. 12, no. 2, pp. 120–139, 2003.

[2] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom, “Stream: The stanford data stream management system,” Stanford InfoLab, Technical Report 2004-20, 2004.

[3] C. D. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk, “Gigascope: A stream database for network applications,” in *SIGMOD Conference*, 2003, pp. 647–651.

[4] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh, “Data Cube: A relational aggregation operator generalizing Group-By, Cross-Tab, and Sub-Total,” in *ICDE*, 1996, pp. 152–159.

[5] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker, “Semantics and evaluation techniques for window aggregates in data streams,” in *SIGMOD Conference*, 2005, pp. 311–322.

[6] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, and D. Maier, “Out-of-order processing: a new architecture for high-performance stream systems,” *PVLDB*, vol. 1, no. 1, pp. 274–288, 2008.

[7] R. Mueller, J. Teubner, and G. Alonso, “Streams on wires - a query compiler for FPGAs,” *PVLDB*, vol. 2, no. 1, pp. 229–240, 2009.

[8] Y. Oge, M. Yoshimi, T. Miyoshi, H. Kawashima, H. Irie, and T. Yoshinaga, “FPGA-based implementation of sliding-window aggregates over disordered data streams,” *IEICE Technical Report*, vol. 112, no. 376, pp. 105–110, 2013, CPSY2012-74.

[9] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras, “Exploiting punctuation semantics in continuous data streams,” *IEEE Trans. Knowl. Data Eng.*, vol. 15, no. 3, pp. 555–568, 2003.