An Efficient and Scalable Implementation of Sliding-Window Aggregate Operator on FPGA

An Efficient and Scalable Implementation of Sliding-Window Aggregate Operator on FPGA

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 presents an efficient and scalable implementation of an FPGA-based accelerator for sliding-window aggregates over disordered data streams. With an increasing number of overlapping sliding-windows, the window aggregates have a serious scalability issue, especially when it comes to implementing them in parallel processing hardware (*e.g.,* FPGAs). To address the issue, we propose a resource-efficient, scalable, and order-agnostic hardware design and its implementation by examining and integrating two key concepts, called Window-ID and Pane, which are originally proposed for software implementation, respectively. Evaluation results show that the proposed implementation scales well compared to the previous FPGA implementation in terms of both resource consumption and performance. The proposed design is fully pipelined and our implementation can process out-of-order data items, or tuples, at wire speed up to 200 million tuples per second.**

I. Introduction

An important and growing class of applications deals with continuous data streams to identify emerging trends in real-time directly from the data streams. Many data processing tasks, such as financial analysis, traffic monitoring, and data processing in sensor networks, are required to handle a huge amount of data with certain time restrictions. To address the issue, database researchers have expanded the data processing paradigm from the traditional "store-and-process" model towards the "stream-oriented processing" model [1, 2, 4]. *Data Stream Management Systems* (DSMSs) deal with potentially infinite streams of data that must be processed for real-time applications, executing SQL-like *continuous queries* over data streams. Real-time processing is essential for DSMSs; in particular, low-latency and high-throughput processing are key requirements of systems that process unbounded and continuous input streams rather than fixed-size stored data sets.

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 Issues. Mueller *et al.* consider the use of FPGAs for data stream processing as co-processors [11]. They present an implementation technique for sliding-window queries on an FPGA. However, there remain two practical issues related to the implementation of sliding-window aggregation:

1) The first issue is that it is necessary to consider out-of-order arrival of tuples at a windowing operator.
2) The second issue is that a large number of overlapping sliding-windows cause severe scalability problems in terms of both performance and area.

Unfortunately, Mueller *et al.* neither address nor discuss both of the above issues. In our previous work [14, 15], we address the first issue regarding out-of-order arrival of tuples. To the best of our knowledge, however, the second issue is still an open question. In this paper, we focus on the second issue and present a scalable implementation for a sliding-window aggregate query on an FPGA.

Our Contribution. This paper proposes a scalable and order-agnostic hardware design of sliding-window aggregation and its implementation on an FPGA, by examining and integrating two key concepts: Pane [7] and *W*indow-*ID* (WID) [8]. Instead of replicating a large number of aggregation modules for overlapping sliding-windows, we divide each sliding window into non-overlapping sub-windows called *panes*. For each sub-window, or pane, we first calculate a sub-aggregate (*i.e.,* pane-aggregate), which is then shared by the aggregation of the multiple windows (*i.e.,* overlapping sliding-windows). The pane-based approach is originally proposed for software-based implementation to reduce the required buffer size and the computation cost [7]. In this work, however, we show that the same idea can provide significant benefits for hardware-based implementation, especially in terms of performance (*i.e.,* the maximum allowable clock frequency), area (*i.e.,* the hardware resource usage), and scalability.

Furthermore, we integrate the authors' work [14, 15], which is based on WID, into the proposed design in order to address the first issue, namely out-of-order arrival of tuples. As a result, the proposed implementation addresses both of the two problems discussed above (*i.e.,* the out-of-order arrival of tuples and the scalability issues). To the best of our knowledge, this is the first paper that integrates the two concepts, namely Pane and WID, for hardware-based approach, by designing and implementing an efficient and scalable sliding-window aggregate operator on an FPGA.

Finally, to demonstrate the effectiveness of the proposed design, we implement the same experimental system as that of [15] using Xilinx ML605 Evaluation board and measure data rates that the FPGA board can sustain. Our experiments show that the proposed implementation can process significantly high tuple rates at wire speed.

The rest of the paper is organized as follows. Section II presents necessary background and briefly reviews related work. Section III provides design concepts underlying the proposed approach. Section IV introduces a target query and presents its hardware implementation. Section V evaluates the proposed implementation with some experimental results. Finally, section VI concludes the paper by summarizing the results.

II. Background and Related Work

A. Continuous Query Processing on FPGAs

Mueller *et al.* show the potential of FPGAs as an accelerator for data intensive operations [10]. It is demonstrated in [10] that FPGAs can achieve competitive performance compared to modern general-purpose CPUs while providing remarkable advantages in terms of power consumption and parallel stream evaluation. Due to their low-latency and high-throughput processing advantages, FPGAs are used to process data streams.

For example, Sadoghi *et al.* propose an efficient event-processing platform called *fpga-ToPSS* [16], and demonstrate high-frequency and low-latency algorithmic trading solutions [18]. These works mainly focus on queries with selection operator. Alternatively, another work [17] concentrates on the execution of SPJ (Select-Project-Join) queries with multi-query optimization. Other works such as [3, 12, 13] focus on the acceleration of window join operators. The main focus of this paper, however, differs from all of the above works as we are primarily concerned with the scalability issue of the FPGA-based implementation of sliding-window aggregate queries.

B. Sliding-Window Aggregate Queries

Consider the following online auction example taken from Li *et al.* [7]. In this example, an online auction system monitors bids on auction items. We assume an input stream that contains information about each bid, the schema of which is defined as <*item-id, bid-price, timestamp*>. In addition, assume that the online auction system runs over the Internet, and each bid is streamed into a central auction server where a DSMS is running. Query Q_1 (cited from [7]) shows an example of a simple sliding-window aggregate query.

Query Q_1: "Find the maximum bid-price for the past 4 minutes and update the result every 1 minute."

```
SELECT max(bid-price), timestamp
  FROM bids [RANGE 4 minutes
             SLIDE 1 minute
             WATTR timestamp]
```

Following the definition of window semantics [8], Query Q_1 introduces a window specification which consists of a window type and a set of parameters that define a window. In Query Q_1, *sliding windows* have three parameters: $RANGE$, $SLIDE$, and $WATTR$. $RANGE$ indicates the size of the windows; $SLIDE$ indicates the step by which the windows move; $WATTR$ indicates the windowing attribute—the attribute over which $RANGE$ and $SLIDE$ are specified [8].

Given the specification above, the bid stream is divided into overlapping 4-minute windows starting every minute, based on the timestamp attribute of each tuple. For each window, the maximum value of bid-price is calculated, and Query Q_1 generates an output stream with schema <*max, timestamp*> where the timestamp attribute specifies the end of the window.

C. Glacier

Mueller *et al.* propose a query-to-hardware compiler, called Glacier [11], which is a compiler that provides a library of components and transforms continuous queries into logic circuits to be implemented on an FPGA. Glacier generates logic circuits by composing library components on an operator-level basis and supports basic continuous queries involving selection, aggregation, grouping, and windowing operators. The windowing operator provides sliding-window functionality and aggregate operator includes four distributive (*i.e.,* COUNT, SUM, MIN, and MAX) and an algebraic (*i.e.,* AVERAGE) aggregate functions, which are classified by Gray *et al.* [6]. As windowing and aggregation operators are provided by the library, Glacier can compile sliding-window aggregate queries into hardware circuits; however, there are two important issues related to the implementation of sliding-window aggregation.

The first issue is that 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. The assumption does not always fit into a realistic setting where some degree of disorder (*i.e.,* out-of-order arrival of tuples) might be happened. For example in Query Q_1, 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.

The second key issue is the scalability in terms of both resource consumption and performance. Glacier relies on simultaneous evaluations of overlapping sliding-windows by instantiating a number of aggregation modules on an FPGA. The number of the aggregation modules required to be instantiated by Glacier, N_{Glacier}, is calculated based on two parameters: RANGE and SLIDE (see the following Equation 1).

$$N_{\text{Glacier}} = \left\lceil \frac{\mathit{RANGE}}{\mathit{SLIDE}} \right\rceil + 1 \qquad (1)$$

Although this approach may be considered as a possible solution for a relatively small $\frac{\mathit{RANGE}}{\mathit{SLIDE}}$ ratio (*e.g.,* a few tens of the aggregation modules), the approach suffers from the scalability issues for a large $\frac{\mathit{RANGE}}{\mathit{SLIDE}}$ ratio (*e.g.,* several hundreds or thousands of the aggregation modules).

It is stated in [7] that sliding-window aggregate queries allow users to aggregate input streams at a user-specified granularity (*i.e., RANGE*) and interval (*i.e., SLIDE*), and thus provide the users a flexible way to monitor streaming data. However, due to the replication strategy of the aggregation modules for overlapping sliding-windows, the number of replicas linearly increases with increasing $\frac{\mathit{RANGE}}{\mathit{SLIDE}}$ ratio. This results in serious scalability problems especially for large RANGE and/or small SLIDE values. In fact, even if a small $\frac{\mathit{RANGE}}{\mathit{SLIDE}}$ ratio is considered, the replication strategy discussed above leads to extremely poor resource utilization.

D. WID-based Implementation

Abadi *et al.* classify query operators into two categories: *order-agnostic* and *order-sensitive* [1]. It is stated in [1] that 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.

*W*indow-*ID* (WID) [8] is proposed for a software-based implementation of order-agnostic window aggregation. It is stated in [9] that WID provides a method to implement window aggregate queries in an order-agnostic way by using punctuations [19]. Informally, a *punctuation* is a kind of tuple which contains control information and is embedded in a data stream. It is stated in [8] that punctuations can be used to indicate that no more tuples having certain attribute values will be seen in the stream. Therefore, the punctuations are useful to unblock some blocking operators, such as group-by and aggregation, signaling the end of each window. The formal definition and details of the punctuation can be found in [19].

Contrary to the software-based implementation [8], our previous work [14] proposes hardware-based implementation of WID. In the previous work [14], a sliding-window aggregate query, which is directly connected with a UDP/IP stack [5], is implemented on an FPGA. The proposed implementation of sliding-window aggregate circuit demonstrates wire-speed performance on a Xilinx ML605 FPGA board with a Gigabit Ethernet connection [15].

In order to implement WID-based approach on an FPGA, we need an upper bound for the required hardware resources. For the purpose of determining the upper bound, our previous work [14] introduces a new parameter, called *slack* [1]. Slack defines an upper bound on the degree of disorder and any tuple arriving after its corresponding period is discarded. It is stated in [1] that some aggregate operators, such as COUNT, SUM, AVERAGE, MIN, and MAX, can simply delay closing windows according to the slack specification. This approach enables us to handle disordered streams appropriately for sliding-window aggregation.

The authors' previous work basically 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. The number of the aggregation modules of WID-based approach, N_{WID}, is determined by using window parameters (*i.e., RANGE* and SLIDE) and a slack specification (see the following Equation 2).

$$N_{\text{WID}} = \left\lceil \frac{\mathit{RANGE} + \mathit{SLACK}}{\mathit{SLIDE}} \right\rceil \qquad (2)$$

As shown in Equation (1) and (2), the required numbers of the aggregation modules (*i.e.,* N_{Glacier} and N_{WID}) linearly increase with increasing $\frac{\mathit{RANGE}}{\mathit{SLIDE}}$ ratio. This is because both Glacier and our previous works rely on simultaneous evaluations of overlapping sliding-windows by simply replicating the window-aggregation modules. Since this approach causes the scalability issues in terms of the maximum clock frequency and the hardware resource usage, it is crucial to design and implement a scalable hardware accelerator for sliding-window aggregate operator that can handle large $\frac{\mathit{RANGE}}{\mathit{SLIDE}}$ ratios. In this paper, we adopt a two-step aggregation method using panes [7] and address the scalability problem of the previous implementations even if a large $\frac{\mathit{RANGE}}{\mathit{SLIDE}}$ ratio is considered.

III. Design Concept

A. Sliding Windows and Panes

In this paper, we first divide each sliding-window into disjoint sub-windows, called panes, instead of replicating the window-aggregation modules for overlapping windows. For example, Fig. 1 illustrates overlapping sliding-windows (only the first four windows) for Query Q_1 from Section II-B. Recall from Query Q_1 that the bid stream is divided into overlapping 4-minute windows, each of which starts every 1 minute. Accordingly, in Fig. 1, all windows have $\mathit{RANGE} = 4$ and $\mathit{SLIDE} = 1$, respectively.

How we divide these four sliding-windows into panes is illustrated in Fig. 2. It is stated in [7] that the RANGE and the SLIDE of panes equal to the same value (*i.e., $\mathit{RANGE}_{\text{Pane}} = \mathit{SLIDE}_{\text{Pane}}$*) and, given a sliding-window aggregate query, they are calculated as the greatest common divisor (GCD) of the RANGE and the SLIDE of the original query. Since the original query (*i.e.,* Query Q_1) has $\mathit{RANGE} = 4$ and $\mathit{SLIDE} = 1$, we obtain $\mathit{RANGE}_{\text{Pane}} = \mathit{SLIDE}_{\text{Pane}} = \text{GCD}(4, 1) = 1$ minute. By the definition, the number of panes per window is $\mathit{RANGE}_{Q_1} / \mathit{RANGE}_{\text{Pane}} = 4$. This can be easily noticed that each 4-minute window is composed of four consecutive panes as shown in Fig. 2.

B. Two-Step Aggregation: PLQ and WLQ

In addition to dividing each sliding-window into multiple panes, the original aggregate query is decomposed into two

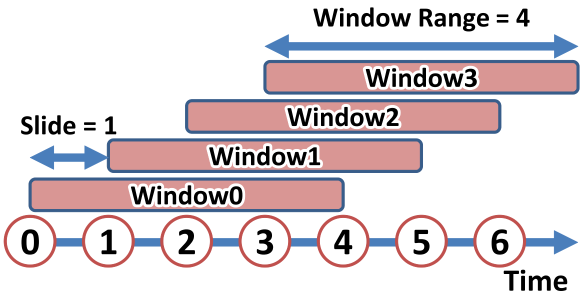

Fig. 1. Overlapping Sliding-Windows for Query Q_1 from Section II-B ($RANGE = 4$ minutes and $SLIDE = 1$ minute).

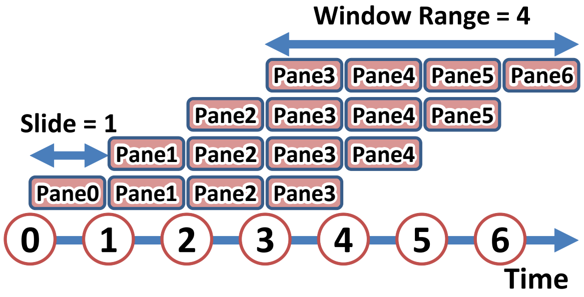

Fig. 2. Each sliding window of Query Q_1 is divided into four non-overlapping sub-windows (*i.e.,* panes), each of which has $RANGE = 1$ minute and $SLIDE = 1$ minute, respectively.

sub-queries: a *pane-level sub-query* (PLQ) and a *window-level sub-query* (WLQ) [7]. We adopt this two-step aggregation approach and our hardware implementation of sliding-window aggregation is based on these two sub-queries. It should be also mentioned that Li *et al.* [7] use the term *pane-aggregates* and *window-aggregates* for the results of the PLQ and the WLQ, respectively. In this work, we use the same terms for the results of the sub-queries.

In order to evaluate a sliding-window aggregate query by using panes, we need to determine window specifications and aggregate functions of the PLQ and WLQ sub-queries, respectively. We can use the same windowing attribute (*i.e.,* $WATTR$) as that of the original query for both sub-queries. In addition, we have already discussed how to determine the $RANGE$ and the $SLIDE$ of panes. As for the WLQ, we use the same $RANGE$ and $SLIDE$ values as those of the original query.

Li *et al.* point out that aggregate functions of the two sub-queries depend on the aggregate function of the original query [7]. In this paper, we focus on the same aggregate functions as Glacier mentioned in Section II-C to implement sliding-window aggregate queries on an FPGA. These are typical aggregate functions in traditional database systems as well as DSMSs. In fact, AVERAGE function can be obtained with the combination of two aggregate values: SUM and COUNT, by simply dividing SUM by COUNT for each window. Therefore, we should consider the remaining four distributive aggregate functions (*i.e.,* COUNT, SUM, MIN, and MAX). Table I summarizes the relation between the aggregate function of the original query and the corresponding aggregates of the PLQ and WLQ. It turns out that, except for COUNT, we use the same aggregate function as the original query for both of the PLQ and WLQ sub-queries. When the original query is COUNT, the PLQ is also COUNT and the WLQ should be SUM, respectively.

TABLE I
RELATION BETWEEN ORIGINAL QUERY, PLQ, AND WLQ

	Aggregate Functions			
Original sliding-window query	COUNT	SUM	MIN	MAX
Pane-level sub-query (PLQ)	COUNT	SUM	MIN	MAX
Window-level sub-query (WLQ)	SUM	SUM	MIN	MAX

It is stated in [7] that the PLQ is a simple *tumbling-window* aggregation, which can be regarded as a special case of a sliding-window aggregate query whose window size (*i.e.,* $RANGE$) is equal to the hop size (*i.e.,* $SLIDE$). For each non-overlapping sub-window (*i.e.,* pane), the PLQ calculates an aggregate value, which is an intermediate result for the original sliding-window aggregate query. For example, the following Query Q_2 shows the PLQ of the original query (*i.e.,* Query Q_1).

Query Q_2: "Find the maximum ***bid-price*** for the past ***1 minute*** and update the result every ***1 minute***."

```
SELECT max(bid-price) as p-max, timestamp
  FROM bids [RANGE 1 minute
             SLIDE 1 minute
             WATTR timestamp]
```

In Query Q_2, the bid stream is divided into non-overlapping 1-minute windows starting every 1 minute. For each window, the maximum value of bid-price is calculated and Query Q_2 generates pane-aggregates with schema <*p-max, timestamp*> where the timestamp attribute indicates the end of each pane.

The other sub-query, WLQ, is a sliding-window query over the intermediate results of the PLQ and produces the final result for each window. For example, the following Query Q_3 shows the WLQ of the original query (*i.e.,* Query Q_1).

Query Q_3: "Find the maximum ***p-max*** value for the past 4 minutes (*i.e.,* ***4 panes***) and update the result every 1 minute."

```
SELECT max(p-max) as w-max, timestamp
  FROM panes [RANGE 4 minutes
              SLIDE 1 minute
              WATTR timestamp]
```

Query Q_3 accepts the pane-aggregates as its input and runs over the output stream of Query Q_2 (*i.e.,* the PLQ). In the WLQ, each pane (except for the first three panes) contributes four windows. For example, as shown in Fig. 2, Pane3 contributes to Window0 through Window3. Similarly, Pane4 contributes to Window1 through Window4 and so forth. For each window, the WLQ computes the max of p-maxes of the last four panes and generates the window-aggregate with schema <*w-max, timestamp*> where the timestamp indicates the end of the window.

C. Hardware Cost Model

1) Pane-Buffer: In addition to the PLQ and WLQ, we also need to consider the design of *pane-buffer* when it comes to implementing the two-step aggregation on an FPGA. The pane-buffer is a cyclic first-in first-out (FIFO) buffer with support for random-access reads. As its name suggests, it stores the intermediate results of the PLQ (*i.e.,* pane-aggregates). It should be also noted that, by using panes, a sliding-window aggregate query can be evaluated with constant buffer space in dependent of the number of input tuples. Given a set of window specifications (*i.e.,* $RANGE$, $SLIDE$, and $WATTR$), we can determine the size of the pane-buffer, S_{buffer}, in terms of the number of pane-aggregates. In fact, the required buffer space is equal to the number of panes per window; therefore, the following Equation 3 gives us the size of the pane-buffer.

$$S_{\text{buffer}} = \frac{RANGE}{\text{GCD}(RANGE, SLIDE)} \tag{3}$$

Given a set of window specification, we can easily calculate S_{buffer} as a constant value. With the constant bound on the size of the pane-buffer, we can efficiently implement the buffer using on-chip Block RAMs (BRAMs) on an FPGA. This is a significant difference between the approach adopted in this paper and that of Glacier. While Glacier and our previous works are not able to utilize BRAMs and only use the limited logic resources on an FPGA, the proposed approach balances logic and BRAM utilization. This results in a considerable area reduction and a higher maximum frequency.

2) Number of the Aggregation Modules: The hardware design of PLQ is based on the WID-based approach discussed in Section II-D. The main difference, however, is that the $RANGE$ of the PLQ is always equal to its $SLIDE$ value. As a result, we can simplify Equation 2 to calculate the number of the PLQ aggregate modules, N_{PLQ}, required to be instantiated (see the following Equation 4).

$$N_{\text{PLQ}} = \left\lceil \frac{RANGE_{\text{Pane}} + SLACK}{SLIDE_{\text{Pane}}} \right\rceil \tag{4}$$

By substituting $RANGE_{\text{Pane}} = SLIDE_{\text{Pane}}$, we obtain Equation 5.

$$N_{\text{PLQ}} = \left\lceil \frac{SLACK}{SLIDE_{\text{Pane}}} \right\rceil + 1 \tag{5}$$

The hardware design of WLQ is based on a sequential evaluation of the pane-aggregates. That is to say, with the pane-buffer mentioned above, we merely require to instantiate a single aggregation module to implement the WLQ. Therefore, the number of the WLQ aggregate module, N_{WLQ}, is always equal to one (*i.e.,* $N_{\text{WLQ}} = 1$). Finally, by adding N_{PLQ} and N_{WLQ}, one obtains the total number of the aggregation modules, N_{Total}, required to implement the proposed design (see the following Equation 6).

$$N_{\text{Total}} = N_{\text{PLQ}} + N_{\text{WLQ}} = \left\lceil \frac{SLACK}{SLIDE_{\text{Pane}}} \right\rceil + 2 \tag{6}$$

Equation 6 suggests that the number of the total aggregation modules of the proposed pane-based hardware design is not affected by $\frac{RANGE}{SLIDE}$ ratio. In other words, contrary to Glacier (Equation 1) and WID-based implementation (Equation 2), the proposed design can handle large $\frac{RANGE}{SLIDE}$ ratios on the order of, say, hundreds or even thousands.

Moreover, recall from Section II-D that $SLACK$ defines an upper bound on the degree of disorder in order to wait for late tuples to arrive before finishing aggregate calculations. It is however stated in [1] that, given the real-time requirements of many stream applications, it is essential that it be possible to "time out" aggregate computations, even at the expense of accuracy. Since larger $SLACK$ values result in longer latencies, we should restrict ourselves to a relatively small $SLACK$ value. For example, assuming the same $SLACK$ as in [14] (*i.e.,* 60 seconds) for Query Q_1, one obtains $N_{\text{PLQ}} = 2$ and $N_{\text{WLQ}} = 1$, which yield $N_{\text{Total}} = 3$. This means that the required number of the aggregation modules (*i.e.,* N_{Total}) remains constant with increasing $\frac{RANGE}{SLIDE}$ ratio. Thus, the proposed approach does not suffer from the scalability problems observed in Glacier and the WID-based implementation.

IV. Implementation Details

A. Motivating Application and Target Query

Glacier presents an implementation of a sliding-window aggregate query on an FPGA. The implementation of the query (Query Q_3 of Glacier [11]) includes a windowing operator which implicitly relies on the arrival sequence of input tuples. Contrary to Glacier, WID-based approach [14] permits windowing on any attribute, allowing a bounded disorder of the tuples. This work also focuses on the same query as a case study and proposes a resource-efficient, scalable, and order-agnostic hardware implementation by using panes.

We assume the same financial trading application as that of previous work [11, 14, 15]. Similar to WID-based implementation, the proposed approach requires an explicit timestamp attribute to define windows over an input stream. The schema of the input stream, called Trades, is defined 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 field. Hence, each tuple has a total size of $4 \times 32 = 128$ bits, or 16 bytes. Based on the definition of window semantics [8], we can rewrite the sliding-window aggregate query as follows.

Query Q_4: "Count the number of trades of UBS (Union Bank of Switzerland) shares for the past 600 seconds and update the result every 60 seconds."

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

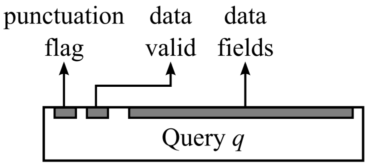

Fig. 3. Wiring Interface (figure cited from [14]).

Given the window specification of Query Q_4, the input stream is divided into overlapping 10-minute windows ($RANGE$ = 600 seconds) starting every minute ($SLIDE$ = 60 seconds), based on the *Time* attribute of each tuple. For each window, Query Q_4 counts the number of trades whose Symbol equal to UBSN (*i.e.,* WHERE Symbol = "UBSN"). The query produces an output stream with schema $<$ *Time, Number* $>$ where the *Time* attribute indicates the end of each window. The details of the implementation of Query Q_4 are provided in the following subsections.

B. Wiring Interface

We adopt the same notation as that of Glacier and WID-based implementation. Following the notation of [11], Fig. 3 (cited from [14]) shows the black box view of a hardware implementation for a query q. In Fig. 3, *punctuation flag* and *data valid* are one-bit signals which indicate the presence of a punctuation and 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.

C. Hardware Execution Plan

An overview of a hardware execution plan for Query Q_4 is illustrated in Fig. 4. As shown in Fig. 4, Query Q_4 is implemented as a synchronous 5-stage pipeline. The first three stages correspond to pane-level sub-query (PLQ) and the remaining two stages are related to window-level sub-query (WLQ). The implementation details of PLQ and WLQ are discussed in the following subsections IV-D and IV-E, respectively.

The gray-shaded boxes in Fig. 4 represent flip-flop registers which can be regarded as pipeline registers. The pipeline stages share a common clock signal and they are inserted between each stage as shown in Fig. 4. These registers buffer intermediate results at the end of each stage and the successive stages can use the result of the previous stage. The arrows in Fig. 4 indicate the connections between the pipeline stages. According to the notation of [11], 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*.

D. Implementation of Pane-Level Sub-Query (PLQ)

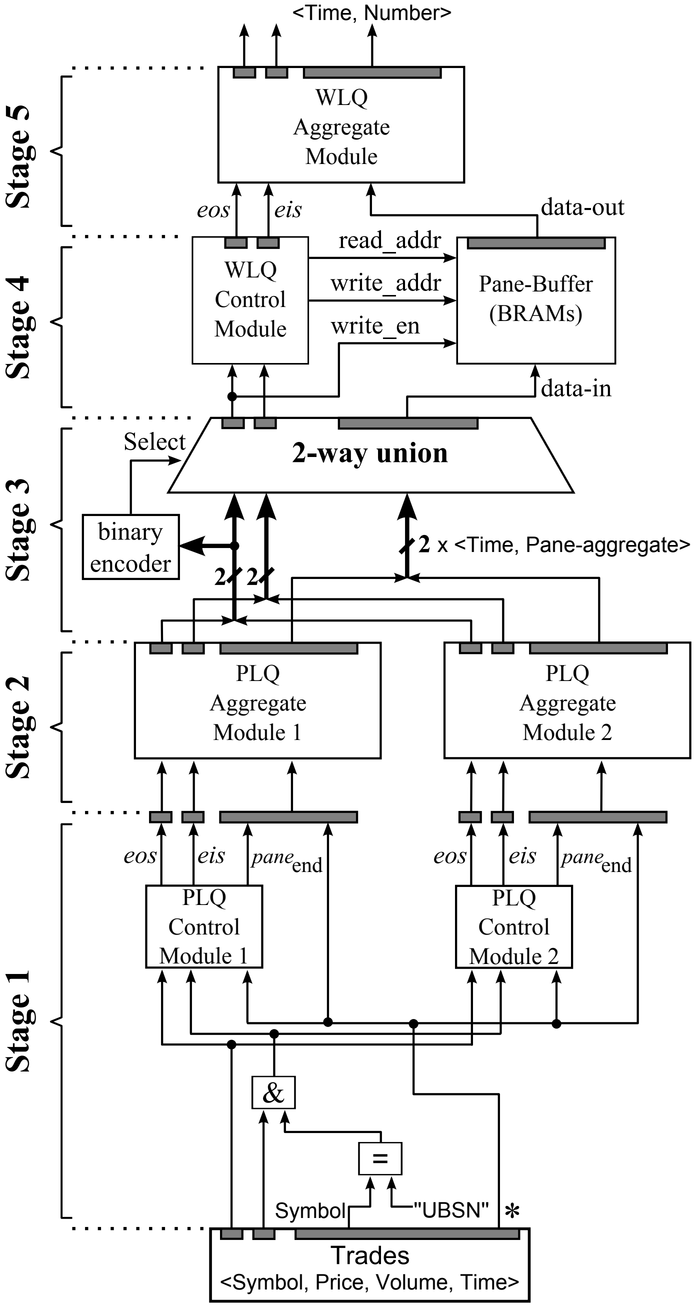

Fig. 4. Hardware execution plan for Query Q_4.

The PLQ is implemented as a 3-stage pipeline and the first three stages of Fig. 4 (*i.e.,* **Stage 1**, **Stage 2**, and **Stage 3**) show its data flow. In **Stage 1**, *Symbol* field of the data bus is compared to a constant string, "UBSN", which is specified in the WHERE expression of Query Q_4 (indicated as $\boxed{=}$ in Fig. 4). At the same time, a logical AND gate (indicated as $\boxed{\&}$ in Fig. 4) computes whether an input tuple is valid for the query, using the result of the comparison. If the tuple should be discarded (*i.e.,* not satisfy the WHERE condition), the *data valid* flag is negated (*i.e.,* set to logic "0") for the next PLQ control modules. The PLQ control modules correspond to windowing operators that provide sliding-window functionality. We require the same number of PLQ control modules as PLQ aggregate modules. Hence, the number of PLQ control

Algorithm 1 Maintain pane states for PLQ

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

Algorithm 2 Generate asynchronous control signals for PLQ

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

modules to be instantiated is calculated by Equation 5 (see Section III-C).

Each PLQ control module maintains pane states and provides two control signals, eis and eos, to the next stage of the pipeline (*i.e.,* **Stage 2**). The eis stands for *enable input stream* and it indicates whether or not data on the *data fields* should be considered as a valid tuple for the current pane. The other signal, eos, stands for *end of stream* and it indicates whether an input stream reaches the end of the current pane. The PLQ control module is responsible for its own states by updating its internal registers called $pane_{\text{begin}}$ and $pane_{\text{end}}$. These registers represent the beginning and the end of the current pane, respectively. The control module uses $pane_{\text{begin}}$ and $pane_{\text{end}}$ 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 [14].

Algorithm 1 describes how to initialize and update the two registers, $pane_{\text{begin}}$ and $pane_{\text{end}}$. Since the windowing attribute (*i.e.,* $WATTR$) of Query Q_4 is defined as $TIME$, $WATTR_{\text{start}}$ is equivalent to $TIME_{\text{start}}$ 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 PLQ control modules concurrently perform the same operation on each cycle in a synchronous manner.

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. As shown in Fig. 4, eis and eos signals are connected to the data valid and punctuation flag registers, respectively. This means that these pipeline registers can be updated within the same clock cycle as soon as eis and eos signals are changed. In other words, all of the operations performed in **Stage 1** can be completed in a single clock cycle.

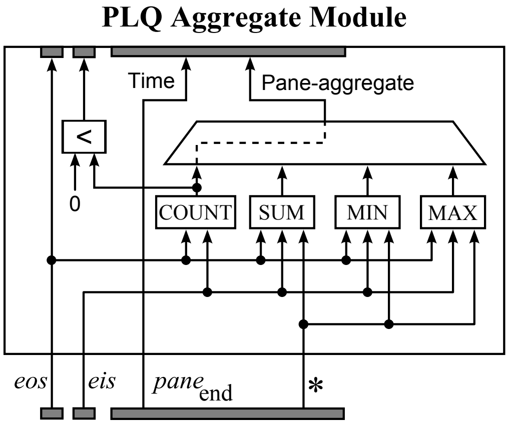

Fig. 5. Block diagram of a PLQ aggregate module.

The next step is **Stage 2** of the pipeline which corresponds to aggregate operators. In **Stage 2**, two PLQ aggregate modules are instantiated as shown in Fig. 4. A more detailed block diagram of a single PLQ aggregate module is depicted in Fig. 5. The PLQ aggregate module includes four aggregate operators as shown in Fig. 5. It is stated in [11] that aggregate functions such as COUNT, SUM, MIN, and MAX can be implemented in a straightforward fashion on an FPGA. Since the PLQ requires $\text{count}(*)$ function, the result of the COUNT operator is selected as the output value (indicated as the broken line in Fig. 5).

The aggregate operator incrementally computes aggregate value and only stores the current (partial) result of the aggregation. It requires two control signals (*i.e., eis* and *eos*) to maintain its aggregate value. Whenever eis is asserted, the aggregate operator accepts input tuple and records its contribution to the partial result. If eis is negated, the aggregate operator simply ignores the input data and waits for the next tuple to arrive. When eos is asserted, it means that the current pane is no longer active and the aggregate operator should reset its internal state. It should be noted that, similar to **Stage 1**, all operations performed in **Stage 2** can be completed in a single clock cycle.

We adopt a similar approach as in [15] to implement a union operator, which is based on a multiplexer component. The main difference however is the required size of the multiplexer. In the previous work [15], the size of the multiplexer increases with increasing $\frac{RANGE}{SLIDE}$ ratio and hence leads to scalability problems. On the other hand, the proposed approach is not affected by the $\frac{RANGE}{SLIDE}$ ratio; in other words, our method requires a constant-size multiplexer. This is a significant difference between the proposed implementation and that of [15].

According to a select signal, the multiplexer component transfers the result of i-th PLQ aggregate module to the output registers of **Stage 3**. As illustrated in Fig. 4, the select signal is provided by a binary encoder component. It is stated in [11] 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. 4, the 2-way union operator merges the outputs of two PLQ aggregate modules and generates a single result stream. It should be also mentioned that **Stage 3** requires only one clock cycle to complete its operation.

E. Implementation of Window-Level Sub-Query (WLQ)

The WLQ is implemented as a 2-stage pipeline and the last two stages of Fig. 4 (*i.e.,* **Stage 4** and **Stage 5**) show its data flow. **Stage 4** includes a single WLQ control module and the pane-buffer. Similar to the PLQ control module, the WLQ control module maintains its internal states and provides two control signals, eis and eos, to the final stage of the pipeline (*i.e.,* **Stage 5**). In addition, the WLQ control module also provides read and write addresses to the pane-buffer.

Details about how to maintain the window states and to generate the control signals are provided in Algorithm 3 and Algorithm 4, respectively. Algorithm 3 describes how to initialize and update four internal registers: wr_addr, rd_addr, rd_addr_prev, and $pane\_counter$. It should be mentioned that wr_addr and rd_addr registers are connected to the write_addr and read_addr ports of the pane-buffer (see **Stage 4** of Fig. 4), respectively. Algorithm 4 describes how to generate the two control signals, eis and eos, for the WLQ aggregate module.

As discussed in Section III-C, the pane-buffer is a cyclic first-in first-out (FIFO) buffer and its implementation is based on on-chip Block RAMs (BRAMs). BRAMs support dual-ports and each port has its own data-in, data-out, and address bus. We use simple dual-port mode, that is to say, one port can only write and the other port can only read data. As illustrated in Fig. 4, the data field of **Stage 3** is connected to the data-in port (*i.e.,* write-only port) of the pane-buffer. Similarly, the data-out port (*i.e.,* read-only port) of the pane-buffer is directly connected to the next stage (*i.e.,* **Stage 5**).

Stage 4 requires a total of three clock cycles to complete its operation when the last pane-aggregate of each window arrives from the previous stage (*i.e.,* **Stage 3**). Specifically, when the punctuation flag is asserted, one clock cycle is required to update the wr_addr register. After that, another clock cycle is consumed to update the rd_addr and $pane\_counter$ registers. Finally, the third clock cycle is used to retrieve data from the pane-buffer, which is implemented using BRAM primitives.

When it comes to the implementation of **Stage 5** of Fig. 4, the WLQ aggregate module is implemented almost in the same way as PLQ aggregate module (see Fig. 5). Hence, all operations performed in **Stage 5** can be completed in a single clock cycle. Mueller *et al.* [11] evaluate 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 7 cycles and 1 tuple/cycle, respectively.

Algorithm 3 Maintain window states for WLQ

```
State Registers:
$wr\_addr$: write-address register of the pane-buffer
$rd\_addr$: read-address register of the pane-buffer
$rd\_addr\_prev$: previous value of the read-address register
$pane\_counter$: pane counts in the current window
Initialization:
$wr\_addr \leftarrow 1$
$rd\_addr \leftarrow 0$
$rd\_addr\_prev \leftarrow 0$
$pane\_counter \leftarrow 0$
Synchronous Update:
for each clock cycle do
  $rd\_addr\_prev \leftarrow rd\_addr$
  if $pane\_counter <$ PANES_PER_WINDOW then
    if $rd\_addr \neq wr\_addr$ and $rd\_addr + 1 \neq wr\_addr$ then
      $rd\_addr \leftarrow rd\_addr + 1$
      $pane\_counter \leftarrow pane\_counter + 1$
    end if
  else
    $rd\_addr \leftarrow rd\_addr -$ PANES_PER_WINDOW $+ 2$
    $pane\_counter \leftarrow 1$
  end if
  if punctuation flag is asserted then
    $wr\_addr \leftarrow wr\_addr + 1$
  end if
end for
```

Algorithm 4 Generate asynchronous control signals for WLQ

```
Asynchronous Signals:
$eis$: input enable signal for the WLQ aggregate module
$eos$: output enable signal for the WLQ aggregate module
Asynchronous Update:
if $rd\_addr \neq rd\_addr\_prev$ then
  assert $eis$ signal
else
  negate $eis$ signal
end if
if $pane\_counter <$ PANES_PER_WINDOW then
  negate $eos$ signal
else
  assert $eos$ signal
end if
```

TABLE II
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 III
BLOCK RAM UTILIZATION

$\frac{RANGE}{SLIDE}$	# of BRAMs	
64	1	(0.2%)
128	1	(0.2%)
256	1	(0.2%)
512	2	(0.5%)
1024	4	(1.0%)
2048	8	(1.9%)
4096	17	(4.1%)

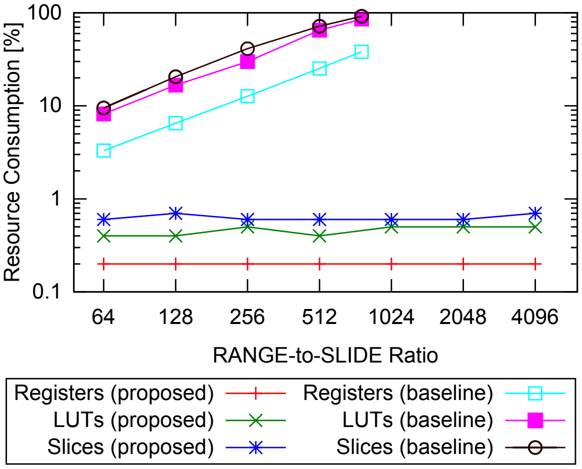

Fig. 6. Comparison of the overall resource consumption between the proposed implementation and the WID-based implementation (baseline) [14].

V. EVALUATION

We have developed the proposed design in VHDL from scratch and implemented it on a Virtex-6 FPGA (XC6VLX240T-1) included in a Xilinx ML605 evaluation board.The specification of the FPGA is given in Table II. Xilinx ISE 14.4 is used during the implementation.

A. Resource Utilization and Performance

In order to evaluate the scalability of the proposed approach, the proposed design (*i.e.*, Fig. 4) and the WID-based approach [14] are implemented for the same target query (*i.e.*, Query Q_4). We have modified the $RANGE$ parameter of the query and the $\frac{RANGE}{SLIDE}$ ratio is increased by multiples of 2, beginning with 64 up to 4096 (*i.e.,* a total of seven different configurations). The proposed implementation is synthesized with a timing constraint of 5 ns for each configuration, which yields the target clock frequency of 200 MHz.

1) Resource Utilization: The comparison of the overall resource consumption is shown in Fig. 6. The x-axis represents $\frac{RANGE}{SLIDE}$ ratio of the time-based sliding window. The y-axis indicates the resource consumption (in log scale) in terms of percentages of the total available resources on the target FPGA device. In Fig. 6, the proposed implementation and the WID-based implementation [14] are labeled “proposed” and “baseline”, respectively.

Results of Fig. 6 suggest that the proposed implementation achieves significant area reduction compared to the baseline. For instance, when $\frac{RANGE}{SLIDE} = 512$, the baseline consumes over 70% of the available slices on the target FPGA whereas the proposed implementation only requires 0.6% of the available slices, by using only two BRAMs on the same FPGA (*i.e.,* about 0.5% of the total available BRAMs). The required number of BRAMs for each configuration is given in Table III.

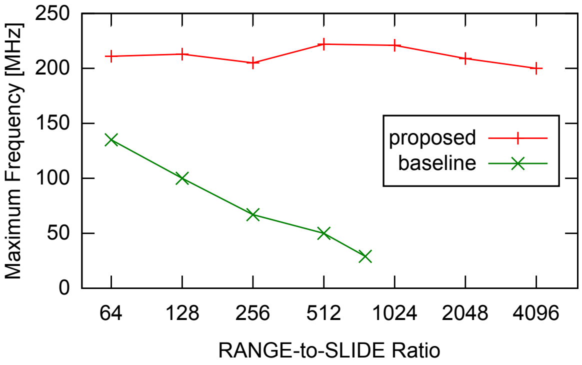

Fig. 7. Comparison of the maximum clock frequency between the proposed implementation and the WID-based implementation (baseline) [14].

Moreover, as shown in Fig. 6, all three graphs of the baseline almost linearly increases with increasing $\frac{RANGE}{SLIDE}$ ratio. As a result, when $\frac{RANGE}{SLIDE} = 768$, the baseline utilizes over 90% of the available slices and if we increase the ratio (*i.e.,* $\frac{RANGE}{SLIDE} \geq 1024$), the query cannot be implemented on the target device due to finite area of the FPGA. On the other hand, however, the proposed implementation does not suffer from this limitation as shown in Fig. 6. All three graphs of the proposed implementation are almost constant and do not increase with increasing $\frac{RANGE}{SLIDE}$ ratio; therefore, the proposed design provides better scalability compared to the baseline.

2) Performance Evaluation: The comparison of the maximum clock frequency is shown in Fig. 7. The x-axis and the y-axis represent $\frac{RANGE}{SLIDE}$ ratio and the clock frequency, respectively. The clock frequency is obtained from post-place & route static timing report, which is provided by Xilinx's Timing Analyzer tool. As shown in Fig. 7, the clock frequency of the baseline drops sharply with increasing $\frac{RANGE}{SLIDE}$ ratio. In contrast, the clock frequency of the proposed design remains largely unaffected by the $\frac{RANGE}{SLIDE}$ ratio. Specifically, the proposed implementation meets the timing constraint of 5 ns and maintains the target clock frequency of 200 MHz for each configuration. The fact that the proposed design can still sustain high frequencies is a good indication for the scalability. For instance, when $\frac{RANGE}{SLIDE} = 512$, the baseline can operate at only up to 50 MHz on the target FPGA whereas the proposed implementation can operate at over 200 MHz. This means that the proposed approach achieves over 4× performance improvement with significantly reduced hardware cost.

Since the issue rate of the proposed design is equal to 1 tuple/cycle, the proposed implementation can process 200 million tuples per second. As for latency, recall that the latency of the implemented queries is equal to 7 cycles, and the clock period is 5 ns if we assume a clock rate of 200 MHz. Hence,

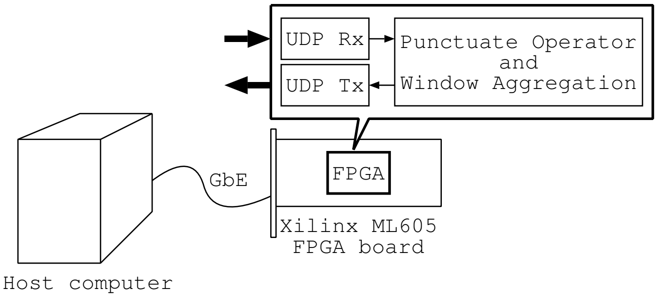

Fig. 8. Overview of the Experimental System (figure cited from [15]).

multiplying 7 by 5 ns yields 35 ns. These data lead us to the conclusion that the proposed design is scalable against $\frac{RANGE}{SLIDE}$ ratio and can accomplish both high throughput (over 200 million tuples per second) and low latency (the order of a few tens of nanoseconds).

B. Experimental Measurement

It is stated in [11] that a key aspect of using an FPGA for data stream processing is its flexibility that enables us to insert custom hardware logic into an existing data path. For example, the proposed sliding-window aggregate circuit can be directly connected to the physical network interface. In order to verify the effectiveness of the proposed method, we implement the same experimental system as that of [15].

Our experiments are based on a Xilinx ML605 FPGA board, which includes the Virtex-6 FPGA and a Gigabit Ethernet interface. An overview of the experimental system is depicted in Fig. 8. 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. 8). Further details of the experimental system can be found in [15].

We have measured the effective throughput of the proposed implementation on the ML605 FPGA board. Results of the experiments show that the proposed implementation achieves an effective throughput up to around 760 Mbps for different $\frac{RANGE}{SLIDE}$ ratios. This is the upper bound of the available bandwidth that the network interface (*i.e.,* the UDP Rx module [5]) 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, even if large $\frac{RANGE}{SLIDE}$ ratios are considered (*e.g.,* $\frac{RANGE}{SLIDE} \geq 1024$).

VI. Conclusions

In this paper, we have proposed an efficient and scalable hardware design of sliding-window aggregate operator and its implementation on an FPGA. The proposed design adopts a two-step aggregation method using panes and supports disordered data arrival with punctuations. The proposed implementation is scalable with the increasing $\frac{RANGE}{SLIDE}$ ratio and significantly reduces the required logic elements by efficiently utilizing Block RAMs. Results show that the proposed implementation can achieve considerable performance improvement over the baseline implementation for large $\frac{RANGE}{SLIDE}$ ratios. To the best of our knowledge, this is the first paper that proposes design and implementation of an FPGA-based sliding-window aggregate operator by using panes. One direction for future work is to provide flexibility allowing run-time configuration. Another direction is to address multi-query execution. We also plan to compare the proposed approach with other methods.

Acknowledgment

This work was supported in part by the Semiconductor Technology Academic Research Center (STARC).

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] J. bo Qian, H. bing Xu, Y. Dong, X. jun Liu, and Y. li Wang, "FPGA acceleration window joins over multiple data streams," *Journal of Circuits, Systems, and Computers*, vol. 14, no. 4, pp. 813–830, 2005.

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

[5] e-trees.Japan, Inc., "e7UDP/IP IP-core," http://e-trees.jp/index.php/en/products/e7udpip-ipcore [Online; accessed 14-July-2013].

[6] 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.

[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," *SIGMOD Record*, vol. 34, no. 1, pp. 39–44, 2005.

[8] 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.

[9] 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.

[10] R. Mueller, J. Teubner, and G. Alonso, "Data processing on FPGAs," *PVLDB*, vol. 2, no. 1, pp. 910–921, 2009.

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

[12] Y. Oge, T. Miyoshi, H. Kawashima, and T. Yoshinaga, "Design and implementation of a handshake join architecture on FPGA," *IEICE Transactions*, vol. 95-D, no. 12, pp. 2919–2927, 2012.

[13] Y. Oge, T. Miyoshi, H. Kawashima, and T. Yoshinaga, "A fast handshake join implementation on FPGA with adaptive merging network," in *SSDBM*, 2013, pp. 44:1–44:4.

[14] 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.

[15] 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 *MCSoC*, 2013 (to appear).

[16] M. Sadoghi, H.-A. Jacobsen, M. Labrecque, W. Shum, and H. Singh, "Efficient event processing through reconfigurable hardware for algorithmic trading," *PVLDB*, vol. 3, no. 2, pp. 1525–1528, 2010.

[17] M. Sadoghi, R. Javed, N. Tarafdar, H. Singh, R. Palaniappan, and H.-A. Jacobsen, "Multi-query stream processing on fpgas," in *ICDE*, 2012, pp. 1229–1232.

[18] M. Sadoghi, H. Singh, and H.-A. Jacobsen, "Towards highly parallel event processing through reconfigurable hardware," in *DaMoN*, 2011, pp. 27–32.

[19] 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.