PAPER

Design and Evaluation of a Configurable Query Processing Hardware for Data Streams

Yasin OGE[†a], *Student Member*, Masato YOSHIMI[†b], Takefumi MIYOSHI[††c], Hideyuki KAWASHIMA[†††d], Hidetsugu IRIE[††††e], *Members*, *and* Tsutomu YOSHINAGA[†f], *Senior Member*

SUMMARY In this paper, we propose Configurable Query Processing Hardware (CQPH), an FPGA-based accelerator for continuous query processing over data streams. CQPH is a highly optimized and minimal-overhead execution engine designed to deliver real-time response for high-volume data streams. Unlike most of the other FPGA-based approaches, CQPH provides on-the-fly configurability for multiple queries with its own dynamic configuration mechanism. With a dedicated query compiler, SQL-like queries can be easily configured into CQPH at run time. CQPH supports continuous queries including selection, group-by operation and sliding-window aggregation with a large number of overlapping sliding windows. As a proof of concept, a prototype of CQPH is implemented on an FPGA platform for a case study. Evaluation results indicate that a given query can be configured within just a few microseconds, and the prototype implementation of CQPH can process over 150 million tuples per second with a latency of less than a microsecond. Results also indicate that CQPH provides linear scalability to increase its flexibility (*i.e.,* on-the-fly configurability) without sacrificing performance (*i.e.,* maximum allowable clock speed).

key words: *FPGA, query processing, data stream, sliding-window aggregation, configurable hardware architecture*

1. Introduction

An important and growing class of applications requires the ability to process online data streams on the fly in order to identify emerging trends in a timely manner. *Data Stream Management Systems* (DSMSs) [1] deal with potentially infinite streams of data that should be processed for real-time applications, executing SQL-like *continuous queries* [2] over data streams. It is essential for DSMSs that incoming data be processed in real time, or at least near real-time, depending on the applications' requirements. In particular, low-latency and high-throughput processing are key requirements for systems that process unbounded and high-rate data streams. In order to meet the requirements, there is currently a great deal of interest in the potential of using field-programmable gate arrays (FPGAs) as custom accelerators for continuous query processing over data streams [3]–[9].

Motivating Issue. One of the key challenges for DSMSs is an efficient support for *sliding-window queries* [10] over unbounded streams. A common limitation from which most FPGA-based approaches suffer is that the existing approaches impose significant overhead on run-time query registration/modification. It is mentioned in an earlier work [11] that while supporting query modification at run time is almost trivial for software-based techniques, they are highly uncommon for custom hardware-based approaches such as FPGAs. Moreover, as stated in another work [12], given the dynamic environment of data streams, queries can join and leave a streaming system at any time. It is therefore imperative for a query processing accelerator to support on-the-fly configurability for easy adaptation to the dynamic environment. To address the problem, Najafi *et al.* propose Flexible Query Processor (FQP) [11] for sliding-window *join* queries. To the best of our knowledge, however, sliding-window *aggregate* queries—an important class of sliding-window queries—are not discussed in their work [11].

Our Contribution. This paper presents Configurable Query Processing Hardware (CQPH), a highly optimized and minimal-overhead query processing engine, especially designed for sliding-window aggregate queries. CQPH is an FPGA-based query processor that contains a collection of configurable hardware modules, each of which supports *(i)* filtering, *(ii)* grouping, and *(iii)* aggregation. CQPH is highly optimized for performance with a fully pipelined implementation to exploit the increasing degree of parallelism that modern FPGAs support. In addition, the proposed design imposes minimal overhead on query configuration. More specifically, CQPH can support registration of new queries as well as modification of existing queries, without a time-consuming compilation process which is a common drawback of the previous approaches.

This paper is an extended version of the authors' previous work [13]. The main contributions of the present paper are summarized as follows.

- Three configurable hardware modules are designed to execute sliding-window aggregate queries:

Manuscript received May 27, 2015.
Manuscript revised August 3, 2015.
Manuscript publicized September 14, 2015.

[†]The authors are with The University of Electro-Communications, Chofu-shi, 182–8585 Japan.
[††]The author is with e-trees. Japan, Inc., Hachioji-shi, 192–0045 Japan.
[†††]The author is with University of Tsukuba, Tsukuba-shi, 305–8577 Japan.
[††††]The author is with The University of Tokyo, Tokyo, 113–8656 Japan.
a) E-mail: oge@comp.is.uec.ac.jp
b) E-mail: yoshimi@is.uec.ac.jp
c) E-mail: miyoshi@e-trees.jp
d) E-mail: kawasima@cs.tsukuba.ac.jp
e) E-mail: irie@mtl.t.u-tokyo.ac.jp
f) E-mail: yosinaga@is.uec.ac.jp

DOI: 10.1587/transinf.2015EDP7203

1. selection module with efficient support for multiple selection conditions,
2. group-by module based on a scalable systolic architecture, and
3. window-aggregation module supporting a large number of overlapping sliding windows.

- Two-phase configuration approach is adopted to provide on-the-fly configurability for given queries:

1. a fully automated integration of the hardware modules to implement CQPH on FPGA, and
2. run-time query configuration with a dedicated query compiler implemented for CQPH.

- The proposed approach is evaluated in terms of

1. latency, throughput, and configuration time;
2. resource utilization and maximum clock frequency with a case study; and
3. performance measurement of a prototype system implemented on a Xilinx FPGA platform.

The proposed approach overcomes the limitations of the previous work [4], [5], [13]–[15], by offering a great degree of flexibility for on-the-fly query configuration. To the best of our knowledge, this is the first paper that presents an FPGA-based query processor that can support run-time configuration of sliding-window aggregate queries with a large number of overlapping sliding windows.

The rest of the paper is organized as follows. Section 2 briefly reviews related work. Section 3 provides design concepts of the proposed approach. Section 4 describes the architecture of CQPH. Section 5 evaluates the proposed design with some experimental results. Finally, Sect. 6 concludes the paper with a summary.

2. Background and Related Work

2.1 Continuous Query Processing on FPGAs

Sadoghi *et al.* propose an event-processing platform called *fpga-ToPSS* [7], and demonstrate high-frequency and low-latency algorithmic trading solutions [8]. These projects mainly focus on queries with selection operator. Alternatively, another work [9] concentrates on the execution of SPJ (Select-Project-Join) queries with multi-query optimization. Other works [6], [16]–[18] focus on the acceleration of window join operators. Mueller *et al.* propose Glacier [4], [5], a query-to-hardware compiler for continuous queries. Glacier can compile a query that includes selection, group-by operation, and sliding-window aggregation. Finally, our previous work [14], [15] presents custom-designed hardware for sliding-window aggregate queries. The above works require full circuit compilation to implement dedicated hardware for different kinds of query workloads. As a result, static compilation process imposes significant overhead on dynamic workload changes.

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

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

2.2 Dynamic Re-Configuration of FPGAs

Most of the previous FPGA-based approaches suffer from a common limitation, namely, lack of flexibility to adapt to dynamic workload changes. One possible solution is to exploit partial reconfiguration technology of FPGAs along with prebuilt libraries of custom-designed components. For example, Dennl *et al.* [19], [20] propose partial reconfiguration-based approach to accelerate a subset of SQL queries for traditional database systems.

Another promising solution is the approach adopted in *skeleton automata* [21], [22] or *Flexible Query Processor* (FQP) [11]. The main idea is to implement a generic template circuit along with a number of configuration registers/memories. The major advantage of both works is that they can offer run-time configurability without long running static compilation or the partial reconfiguration. CQPH shares similar motivation with skeleton automata and FQP; however, target workloads are quite different. The main focus of this paper differs from these works as we are primarily concerned with sliding-window aggregate queries which are not in the scope of the earlier work [11], [21], [22].

2.3 Sliding-Window Aggregate Queries

Figure 1 shows an example of a sliding-window aggregate query [23]. Query Q_1 introduces 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 [24]. Given the specification above, a bid stream is divided into overlapping 4-minute windows starting every minute.

One of the common approaches for DSMSs to unblock aggregate operators is to use special annotations, called *punctuations* [25]. For example, an "end-of-window" punctuation can be embedded into the bid stream for Query Q_1 to unblock MAX aggregate operator at the end of each window. The formal definition and details of the punctuation can be found in Tucker's work [25].

3. Hardware Design Concept

3.1 Two-Step Aggregation: PLQ and WLQ

In this paper, we adopt a two-step aggregation method using *panes* [23]. Each sliding window is divided into non-overlapping sub-windows called panes (we refer Fig. 1 and

Fig. 2 of the authors' previous work [13] for illustration). In addition, a given aggregate query is decomposed into two sub-queries: *pane-level sub-query* (PLQ) and *window-level sub-query* (WLQ). For example, PLQ and WLQ of Query Q_1 are shown in Fig. 2 and Fig. 3, respectively. For each pane, Query Q_2 calculates an intermediate result (*i.e., p-max*) of the original query. After that Query Q_3 accepts *p-max* values as its input and produces a final result (*i.e., w-max*) for each window.

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

Fig. 2 Q_2: "Find the maximum ***bid-price*** as ***p-max*** for the past ***1 minute*** 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]
```

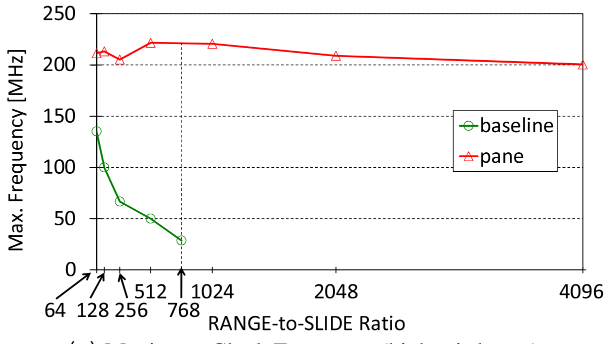
Fig. 3 Q_3: "Find the maximum ***p-max*** value for the past ***4 minutes*** and update the result every ***1 minute***."

In CQPH, the pane-based approach is utilized to obtain significant benefits in terms of performance (*i.e.,* maximum allowable clock frequency), area (*i.e.,* hardware resource utilization), and scalability. In particular, the proposed design does not suffer from the scalability problems observed in Glacier [5] and our previous work [14], [15] (see Fig. 4 (a) and Fig. 4 (b)). We can refer to the previous work [13] for further details.

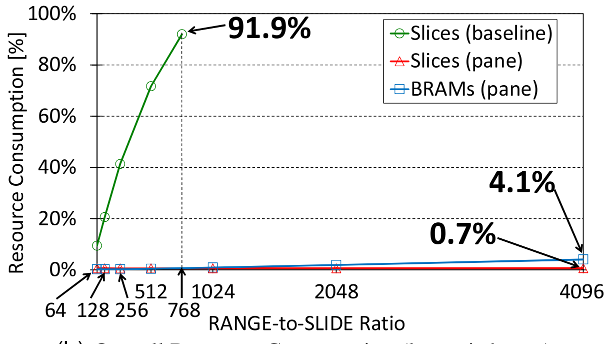
3.2 Two-Phase Configuration Approach

In this paper, we adopt two-phase configuration approach to support on-the-fly configuration of continuous queries, instead of implementing a static query processing hardware that is fully tailored for a specific query. The basic idea of the two-phase configuration approach is illustrated in Fig. 5. The proposed approach is based on *static* and *dynamic* configuration mechanisms.

Static Configuration of FPGA. CQPH is designed as a parameterized HDL model; therefore, static configuration parameters should be provided to generate an application-specific CQPH instance (see Table 1). The implementation of CQPH follows a normal FPGA design flow as shown in Fig. 5 (a). Given a set of static configuration parameters,

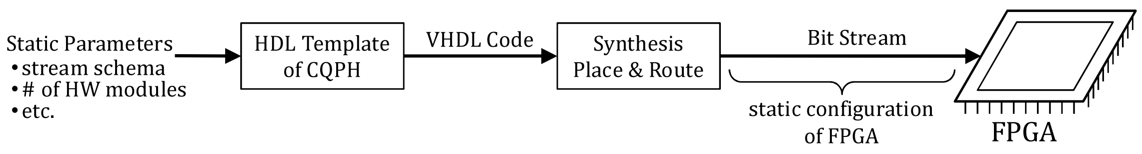


(a) Maximum Clock Frequency (higher is better).

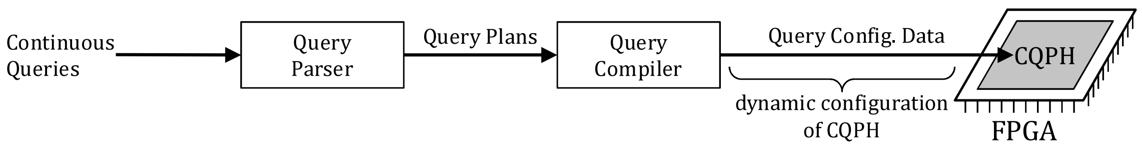


(b) Overall Resource Consumption (lower is better).

Fig. 4 Comparison of the simple replication-based approach [15] (labeled "baseline") and the pane-based approach [13] (labeled "pane"). Each design is implemented on a XC6VLX240T FPGA for the same window parameters (reprinted from the authors' previous work [13]).



(a) One-time-only configuration of an FPGA to implement CQPH by using a standard FPGA tool chain.



(b) On-the-fly configuration of continuous queries by updating internal registers of CQPH at run time.

Fig. 5 Two-phase configuration: static configuration of FPGA (a) and dynamic configuration of CQPH (b).

Table 1 List of statically configured parameters

Stream schema (Tuple Width, # of Attributes)
Pane Buffer Size (Data Width, # of Entries)
of Selection Predicate Modules (Sect. 4.1)
of Boolean Expression Trees (Sect. 4.1)
of Group-by Manager Modules (Sect. 4.2)
of Aggregation Pipelines (Sect. 4.3)
of Pipeline Stages of Union (Sect. 4.4)

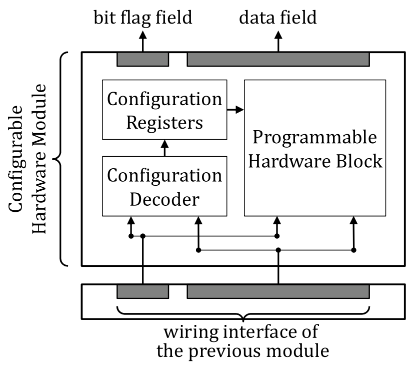

bit flag field
data field
Configurable Hardware Module
Configuration Registers
Configuration Decoder
Programmable Hardware Block
wiring interface of the previous module

Fig. 6 Black-box view of a configurable hardware module.

VHDL description of CQPH is fed to a standard FPGA tool chain (*e.g.,* synthesis, place-and-route) to generate the actual low-level representation of the FPGA-specific circuit. It should be noted that users are required to go through the static configuration process only once to implement CQPH prior to run-time execution of continuous queries.

Dynamic Configuration of CQPH. In order to support run-time configuration of continuous queries, we have implemented a dedicated parser/compiler, *CQPH-compiler*, which can compile continuous queries into *query configuration data* for CQPH. In the proposed design, query configuration data are divided into a set of *configuration tuples* which are then streamed into CQPH. Figure 5 (b) illustrates the compilation process of continuous queries to create query configuration data for CQPH. Note that the compilation process of CQPH does not require a time-consuming synthesis process of FPGA. As a result, CQPH can provide a significant degree of flexibility for run-time query configuration.

CQPH template design consists of a number of configurable hardware modules. Figure 6 illustrates a black-box view of a configurable hardware module and its wiring interface. Each hardware module has its own *bit flag field* and *data field* as connection interfaces. It should be noted that each hardware module adheres the same wiring interface to connect another module. For example, datum on the data field is considered as a part of the query configuration data (*i.e.,* a configuration tuple) when a configuration flag of the bit flag field is asserted (*i.e.,* set to logic "1").

Each configuration tuple consists of two main parts: (i) configuration data field, and (ii) target ID field. As its name suggests, configuration data field contains query configuration data for a specific hardware module. Target ID

```
SELECT * FROM Stream WHERE <boolean expr.>
```

Fig. 7 Q_4: Template of selection-based filtering.

```
SELECT <windowing attribute>,
       <grouping attribute>,
       <aggregate function>
  FROM Stream[RANGE <window size>
              SLIDE <hop size>
              WATTR <windowing attribute>]
 WHERE <boolean expression> (optional)
 GROUP BY <grouping attribute> (optional)
```

Fig. 8 Q_5: Template of window-based aggregation.

field contains a unique identifier assigned to each configurable hardware module. With the proposed approach, users can easily update or modify configuration registers of each hardware module to change the behavior of a programmable hardware block at run time.

3.3 Supported Capabilities of CQPH

The current prototype of CQPH can be configured to execute continuous queries that follow certain patterns. In particular, CQPH supports queries from simple filtering to window-based aggregation (see Fig. 7 and Fig. 8). In Query Q_4 and Q_5, any expression between '<' and '>' can be configured at run time (*i.e.,* dynamically configurable via configuration registers). This is a significant difference between CQPH and a static FPGA-based query processor. In fact, CQPH enables users to add, modify or remove continuous queries at negligible cost compared to Glacier [5] and our previous work [14], [15].

4. CQPH Architecture

In this section, we present the details of CQPH architecture. An overview of the CQPH architecture is illustrated in Fig. 9. In the proposed design, we adopt push-based processing model with a fully pipelined implementation of the configurable hardware modules. Arrows in Fig. 9 represent the direction of data flow between each module, and there is no loopback connections between any two modules. CQPH architecture designed this way remains fully pipelined and operates in a strict streaming fashion at wire-speed rate. This guarantees wire-speed performance and CQPH can accept one input tuple per clock cycle independent of the query workload.

4.1 Selection Operator

Shared selection module (see the bottom of Fig. 9) determines whether or not an incoming tuple satisfies a given set of selection predicates. A selection predicate, or simply a predicate, specifies a condition that is either true or false

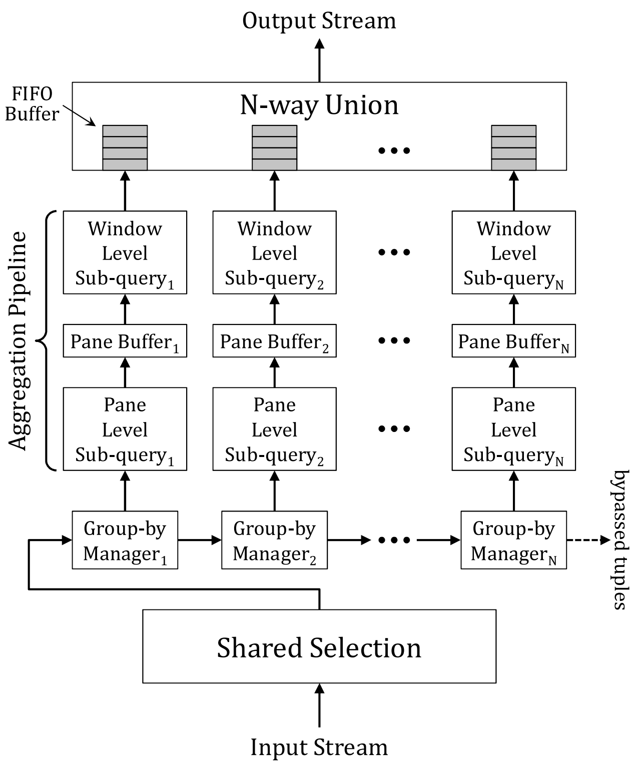


Fig. 9 Overview of the data flow of CQPH Architecture.

```
<boolean expr.> ::= True | False | <tree>
<tree> ::= <predicate>
         | (<tree> AND <tree>)
         | (<tree> OR <tree>)
<predicate> ::= <attribute> <op.> <literal>
<op.> ::= = | ≠ | > | ≥ | < | ≤
```

Fig. 10 Boolean expressions supported by CQPH.

about an input tuple. In SQL-like queries, a selection predicate is typically given as a Boolean expression in WHERE clauses. The current prototype of CQPH only accepts predicates on fixed-length attributes; however, it can still support different kinds of selection conditions, ranging from a single predicate to complex Boolean expressions (see Fig. 10).

In the proposed design, the shared selection module consists of two types of configurable hardware modules: *(i) selection predicate module* and *(ii) binary reducer module*. These hardware modules do not have to be allocated to queries statically. Rather, one can assign an arbitrary number of selection predicates for a specific query at run time. In other words, the same circuit can be utilized for either many queries each of which is assigned to a single predicate or fewer queries with complex Boolean expressions. In either case, the total number of the selection predicate modules limits the number of Boolean expressions that can be processed simultaneously. For example, Fig. 11 illustrates a simplified block diagram of the shared selection module, which is assigned to Query $Q_6 \sim Q_8$ (see Fig. 12).

With the proposed design, each *Boolean expression*

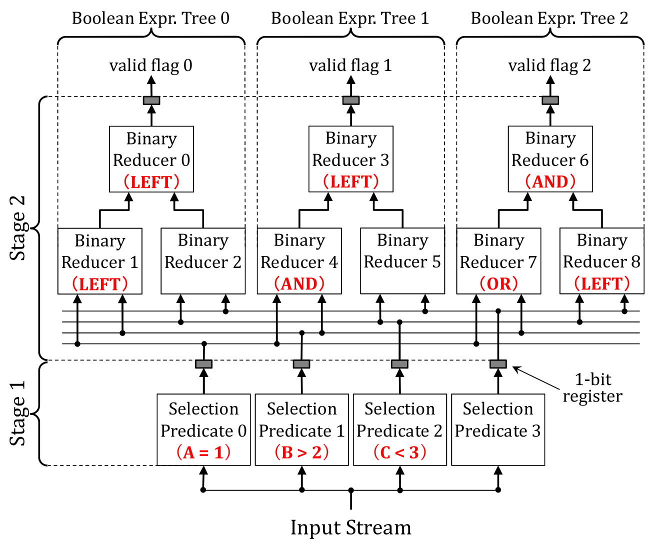


Fig. 11 A simplified block diagram of the shared selection module instantiated with the following parameters:
(i) # of selection predicate modules = 4 (**Stage 1**)
(ii) # of Boolean expression trees = 3 (**Stage 2**)

```
Q6: SELECT * FROM S WHERE A=1
Q7: SELECT * FROM S WHERE A=1 AND B>2
Q8: SELECT * FROM S WHERE (A=1 OR B>2) AND C<3
```

Fig. 12 $Q_6 \sim Q_8$: "Given an input stream S: <*A, B, C*>, select all tuples that satisfy predicates of each query."

tree (*i.e.,* Stage 2 of Fig. 11) can be configured to evaluate a selection condition, by sharing the results of the *selection predicate modules* (*i.e.,* Stage 1 of Fig. 11). For this purpose, CQPH-compiler keeps track of information about the conditions that are already assigned to each module. When a new query is registered, CQPH-compiler compares the new Boolean expression with currently registered ones and decides whether it is possible to share any of selection predicates. In contrast, when a registered query is removed, corresponding modules are cleared unless these modules are shared by the other queries.

Recall from Fig. 6 that the wiring interface of each module consists of a bit flag field and data field. The valid flags of Boolean expression trees are integrated into the bit flag field of the shared selection module. It should also be mentioned that the shared selection module simply forwards data field to the next module. Therefore, for example, when *valid flag* 0 and 2 are asserted (*i.e.,* set to logic "1") and *valid flag* 1 is negated (*i.e.,* set to logic "0"), datum on the data field is considered as a valid tuple for Query Q_6 and Q_8, but not for Q_7.

4.2 Group-by Operator

In the proposed design, we regard the group-by operation as a tuple-routing problem. As shown in Fig. 9, each *Group-by Manager (GM)* module accepts a new tuple from West

```
Input:
  1: West_in port
Output:
  2: East_out port
  3: North_out port
Initialization (dynamically configurable registers):
  4: $qid_{reg}$ ← a specific Query ID
  5: $group_{reg}$ ← NULL
Synchronous Update:
  6: for each clock cycle do
  7:     extract $valid\_flag[qid_{reg}]$ from West_in port
  8:     if $valid\_flag[qid_{reg}] = 1$ then
  9:         extract grouping_attribute from West_in port
 10:         if $group_{reg}$ = NULL then
 11:             $group_{reg}$ ← grouping_attribute
 12:             North_out ← West_in
 13:             $valid\_flag[qid_{reg}]$ ← 0
 14:         else if $group_{reg}$ = grouping_attribute then
 15:             North_out ← West_in
 16:             $valid\_flag[qid_{reg}]$ ← 0
 17:         end if
 18:     end if
 19:     East_out ← West_in
 20: end for
```

Fig. 13 Pseudo code of the routing logic for queries *with* a GROUP-BY clause.

```
Synchronous Update:
  1: for each clock cycle do
  2:     extract $valid\_flag[qid_{reg}]$ from West_in port
  3:     if $valid\_flag[qid_{reg}] = 1$ then
  4:         North_out ← West_in
  5:         $valid\_flag[qid_{reg}]$ ← 0
  6:     end if
  7:     East_out ← West_in
  8: end for
```

Fig. 14 Simplified version of the routing logic for queries *without* a GROUP-BY clause.

port and forwards it to either (i) East port or (ii) both of the two output ports (*i.e.,* North and East ports). In fact, there are two kinds of routing logic inside a GM module: one for queries *with* a GROUP-BY clause (see Fig. 13) and the other for queries *without* a GROUP-BY clause (see Fig. 14). Since the latter is a simplified version of the former, we focus on the former case in the following example.

Figure 15 illustrates the basic idea of how input tuples are processed by GM modules. Each GM module includes a *Query ID (qid)* register which can be configured at the dynamic-configuration phase. The example in Fig. 15 assumes that GM_{n-1} and GM_n have already been configured for qid = 0. Similarly, GM_{n+1} has been configured for qid = 1. These IDs are related to the valid flags of each tuple, which means that *qid 0* and *qid 1* are related to *valid flag 0* and *valid flag 1*, respectively.

According to the given routing logic, input tuples are processed based on the *valid flag fields* and *grouping-attribute value*. For instance, at the time $t = t_0$, GM_{n-1} receives a new tuple which belongs to *Group X*. Notice that

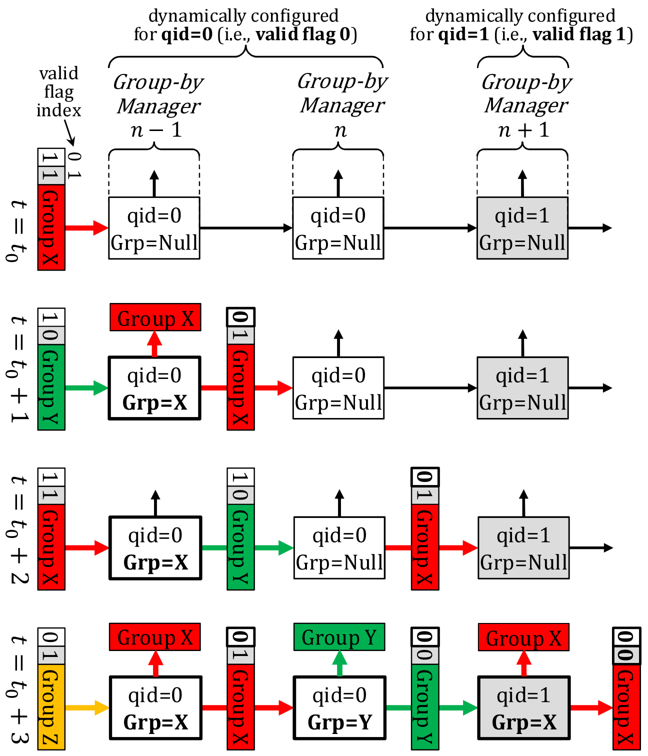


Fig. 15 A simple example for the group-by operation.

valid flag 0 is asserted and this means that the input tuple is valid for *qid 0*. Since GM_{n-1} is not yet assigned to any group (*i.e.,* Grp=Null), *Group X* is registered to GM_{n-1} at the next clock cycle (*i.e.,* $t = t_0 + 1$). At the same time, the input tuple is forwarded to both North (aggregation pipeline) and East (next GM module) ports. It is also important to note that *valid flag 0* is negated to indicate that the corresponding tuple has already been processed for *qid 0*.

Note that after the time $t = t_0$, GM_{n-1} takes responsibility for tuples with *Group X* and the other tuples are simply bypassed to GM_n. For example, when $t = t_0 + 1$, GM_{n-1} receives a new tuple with *Group Y*, which is simply bypassed to GM_n at the next clock cycle (*i.e.,* $t = t_0 + 2$). In contrast, when $t = t_0 + 2$, GM_{n-1} receives a new tuple with *Group X*. In this case, the input tuple (with *Group X*) is forwarded to both North and East ports with *valid flag 0* negated as shown in Fig. 15.

For those queries *with* a GROUP-BY clause, each aggregation pipeline receives tuples from one group only. In addition, incoming tuples are always processed on a first-come-first-served basis, by aggregation pipelines independently of each other. It should be also mentioned that, for those queries *without* a GROUP-BY clause, GM modules use the simplified version of the routing logic (*i.e.,* Fig. 14). In this case, input tuples are routed based only on *valid flag fields*; therefore, an aggregation pipeline can receive different kinds of tuples.

In the current implementation of CQPH-compiler, the compiler requires the maximum number of groups that should be handled by CQPH. In practice, the total number

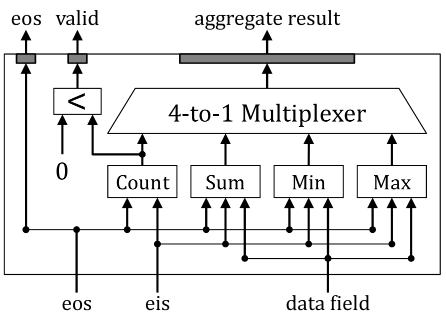


Fig. 16 A simplified block diagram of the aggregation module that includes four sub-modules (*i.e.,* aggregate circuits): COUNT, SUM, MIN, and MAX.

of GM modules provisioned in the CQPH limits the number of groups (or queries) that can be processed simultaneously. Note that CQPH can support the same order of groups as Glacier [5] (*e.g.,* less than a hundred). If the number of groups exceeds the limit of CQPH at run time, these groups can be obtained from East port of the last GM module (see the dashed arrow of GM_N in Fig. 9). By using a similar approach as in Ibex [26], these groups could be bypassed to a host system for further processing (though the processing of bypassed tuples is out of scope of this paper).

4.3 Window-Aggregation Operator

Window-aggregation operators are implemented based on our previous work [13]. In particular, the current version of CQPH can support the following aggregate functions: COUNT, SUM, MIN, and MAX. For the AVERAGE function, the division (*i.e.,* SUM/COUNT) can be performed by software after CQPH computes COUNT and SUM values for each window. Note that these aggregate functions can be decomposed into pane-level and window-level sub-queries (*i.e.,* PLQ and WLQ) [13]. As shown in Fig. 9, a pair of PLQ and WLQ is implemented in a pipelined fashion with a pane buffer. As described in Sect. 4.2, each aggregation pipeline can be mapped to a group or query at run time. With the proposed design, the multi-pipeline architecture allows users to execute multiple continuous queries concurrently.

It is important to note that each aggregation module processes incoming tuples immediately after arrival, rather than batching them up until a pane or window closes. In particular, PLQ and WLQ modules do not store all of the incoming tuples. Instead, CQPH adopts a similar approach as in Glacier [5] and the input tuples are directly forwarded to aggregation circuits inside PLQ or WLQ module (see Fig. 16). In fact, this approach has the advantage that each aggregation circuit needs to provide storage just for the amount of state it requires (*i.e.,* a fixed amount of state), rather than maintaining the entire pane or window [5]. In other words, each aggregate circuit incrementally computes its own aggregate value and only stores the current (*i.e.,* partial) result of the aggregation.

Each aggregation module requires two control signals: *enable input stream (eis)* and *end of stream (eos)* as shown in Fig. 16. Note that these signals can be generated with a simple counter for count-based windows whereas time-based windows require additional logic to generate the control signals. For example, Fig. 17 describes how *eis* and *eos* signals can be generated for a time-based window (*e.g.,* PLQ). Whenever *eis* is asserted, *data field* should be considered as a valid tuple and the aggregation module accepts the input tuple. Once the input stream reaches the end of the current pane, *eos* is asserted and the aggregate value is emitted to the upstream data path. Interested readers can refer to the authors' previous work [13] for more details.

```
Initialization (dynamically configurable registers):
 1: $pane_{\mathrm{begin}} \leftarrow time_{\mathrm{start}}$
 2: $pane_{\mathrm{end}} \leftarrow time_{\mathrm{start}} + RANGE_{\mathrm{PLQ}}$
 3: $pane_{\mathrm{move}} \leftarrow SLIDE_{\mathrm{PLQ}}$
Synchronous Update:
 4: for each clock cycle do
 5:     $eos \leftarrow 0$ (default value)
 6:     $eis \leftarrow 0$ (default value)
 7:     if $punctuation\_flag = 1$ then
 8:         if $pane_{\mathrm{end}} \leq timestamp_{\mathrm{punc.}}$ then
 9:             $eos \leftarrow 1$
10:             $pane_{\mathrm{begin}} \leftarrow pane_{\mathrm{begin}} + pane_{\mathrm{move}}$
11:             $pane_{\mathrm{end}} \leftarrow pane_{\mathrm{end}} + pane_{\mathrm{move}}$
12:         end if
13:     else if $valid\_flag = 1$ then
14:         if $pane_{\mathrm{begin}} < timestamp_{\mathrm{tuple}} \leq pane_{\mathrm{end}}$ then
15:             $eis \leftarrow 1$
16:         end if
17:     end if
18: end for
```

Fig. 17 Generation of *eis* and *eos* signals for PLQ.

4.4 Union Operator

From a data flow point of view, a union operator accumulates several source streams into a single output stream [5]. CQPH adopts a similar approach as that of Glacier [5] to implement a union operator. In CQPH, the union module is implemented in a pipelined fashion, and the number of pipeline stages (N_{S}) can be given as a static configuration parameter. A typical value of N_{S} is one or two, depending on the size of the union module. Note that there is a trade-off between latency and the maximum clock frequency. A large value of N_{S} imposes a latency cost (*i.e.,* two clock cycles per stage); however, this can increase the operating frequency by reducing the critical path delay of the union module.

5. Evaluation

5.1 Latency and Throughput Characteristics

Latency. Latency directly corresponds to the observable response time. Table 2 summarizes latencies of each operation in CQPH. N_{G} and N_{S} represent the number of Group-by Manager modules and the number of pipeline stages of Union module, respectively. Latency is measured in terms of the number of the clock cycles elapsed in the circuit. In

Table 2 Latency and issue rate of each operation

	Latency		Issue Rate
	Lower	**Upper**	
Selection	2	2	1
Group-by	1	N_G	1
Window Aggregation	6	6	1
Union	$2N_S + 2$	$2N_S + 2$	1
Overall Operations	$2N_S + 11$	$N_G + 2N_S + 10$	1

Table 3 Dynamic configuration time for a given query

	Lower	**Upper**
Selection Predicate	0	$N_{SP} = 2^n, n \in \mathbb{Z}^+$
Binary Reducer	1	$N_{SP} - 1$
Group-by Manager	1	N_G
Pane-level Sub-query	2	$2N_G$
Window-level Sub-query	2	$2N_G$
Total Configuration Time	6	$2N_{SP} + 5N_G - 1$

Table 2, the lower bound indicates the minimum latency whereas the upper bound corresponds to the maximum latency. For instance, assuming $N_S = 1$, CQPH can produce the first output tuple of each window in just 13 clock cycles. In other words, the proposed design can respond with a latency of only 130 nanoseconds when clocked at 100 MHz.
Throughput. Issue rate is defined as the number of tuples that can be processed per clock cycle. As shown in Table 2, issue rate of each operation is 1 tuple/cycle, which means that each operator can accept a new tuple every clock cycle. Note that CQPH operates in a strict streaming fashion independent of the query workload, and the maximum throughput of the circuit is directly dependent on its clock rate. For example, the proposed design can process up to 100 million tuples per second when clocked at 100 MHz.

5.2 Dynamic Configuration Time

Table 3 shows dynamic configuration time of CQPH in terms of the number of clock cycles. N_G and N_{SP} represent the number of Group-by Manager modules and the number of Selection Predicate modules, respectively. The lower bound of Table 3 corresponds to a simple sliding-window aggregate query that has neither WHERE nor GROUP-BY clause. The upper bound represents a more generic form with WHERE and GROUP-BY clause (see Query Q_5). In either case, the proposed design can offer run-time configurability at negligible cost (*e.g.,* the order of microseconds) compared to query-tailored circuits [5], [14], [15] (*e.g.,* several minutes or even up to several hours).

5.3 Case Study

This case study considers the same financial trading application as the previous work [5], [13]. We assume an input stream with the following schema: <*Symbol, Price, Volume, Time*>. In this application, over a million tuples can arrive per second, and at the same time, latency is critical

Table 4 Static configuration parameters

Stream schema	128 bits, 4 attributes
Pane Buffer Size	64 bits, 2048 entries
# of Selection Predicate Modules	$N_{SP} \in \{2, 4, 8, 16, 32, 64\}$
# of Boolean Expr. Trees	$N_G \in \{2, 4, 8, 16, 32, 64\}$
# of Group-by Manager Modules	N_G
# of Aggregation Pipelines	N_G
# of Pipeline Stages of Union	$N_S = \begin{cases} 1 & \text{if } N_G \leq 8 \\ 2 & \text{otherwise} \end{cases}$

and measured in units of microseconds [5]. It is therefore crucial for CQPH to meet these performance requirements. For the above application, we have implemented CQPH on a Kintex-7 XC7K325T FPGA (50,950 slices, 407,600 registers, 203,800 LUTs, 445 BRAMs) [27]. The configuration parameters are given in Table 4. Xilinx ISE 14.4 is used during the implementation process (*e.g.,* synthesis and place-and-route). CQPH is synthesized with a timing constraint of 6.37 ns for each configuration, which yields the target clock frequency of 156 MHz.

Note that when sufficient I/O bandwidth is available, the theoretical peak performance of CQPH can be calculated as follows (see Eq. (1)).

$$T_{peak} = d \times f \tag{1}$$

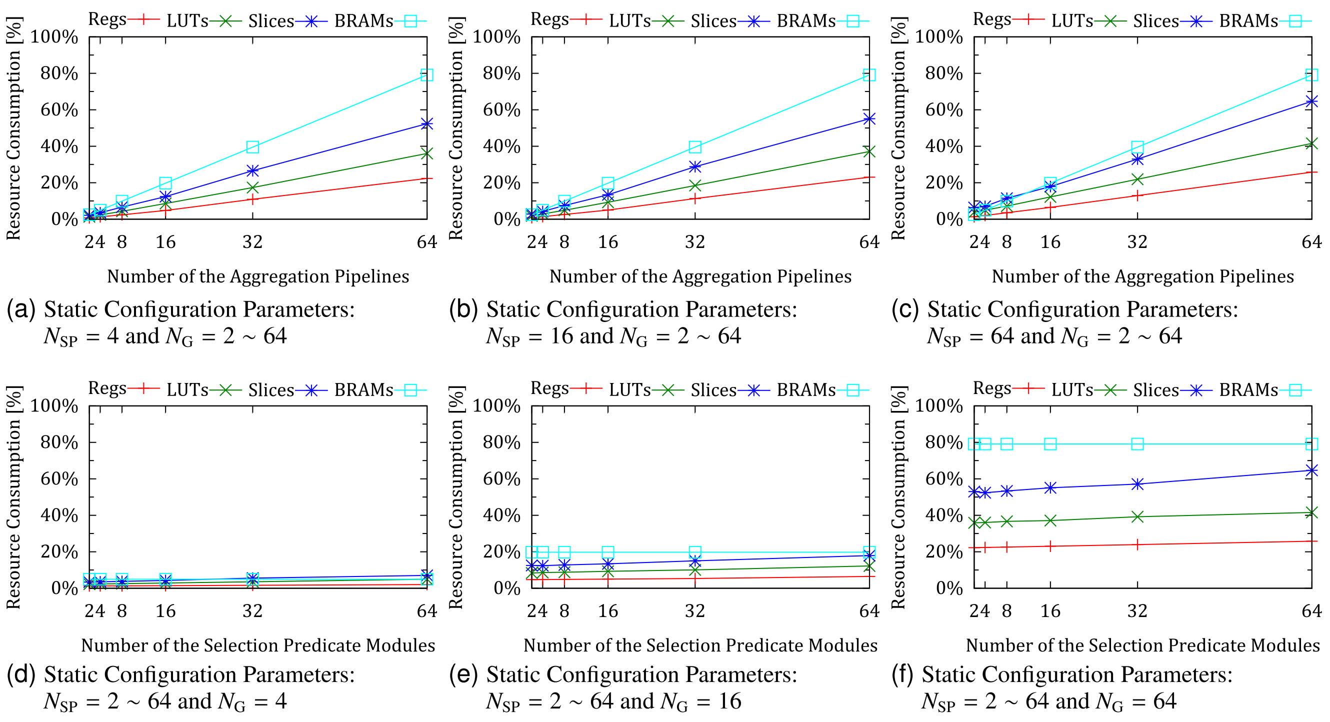
where:

T_{peak}	= theoretical peak throughput
d	= data width of input tuple
f	= operating clock frequency

For example, data width is 128 bits in the case study (see Table 4) and if we assume a clock rate of 156 MHz, the peak performance is equivalent to nearly 20 Gbps (*i.e.,* 128 bits × 156 MHz = 19,968 Mbps).

5.3.1 Hardware Resource Utilization and Performance

Hardware Resource Utilization. The resource consumption is shown in Fig. 18. Each graph represents the resource consumption in terms of percentages of the total available resources on the target FPGA. These graphs indicate trade-offs between area (*i.e.,* resource utilization) and flexibility. For example, all four graphs (*i.e.,* Registers, LUTs, Slices, BRAMs) of Fig. 18 (a), 18 (b), and 18 (c) linearly increases with increasing N_G. These graphs suggest that CQPH provides linear scalability in terms of the hardware resource utilization. Furthermore, each graph of Fig. 18 (d), 18 (e), and 18 (f) indicates a relatively small increase with respect to N_{SP}. This means that we can pre-allocate a large number of selection predicate modules to support complex Boolean expressions.
Performance. Each implementation meets the timing constraint of 6.37 ns and achieves the target clock frequency of 156 MHz. For each configuration, we have obtained almost similar results (*i.e.,* above 156 MHz) from post-place & route static timing report, which is provided by Xilinx's Timing Analyzer tool. Since the issue rate is



(a) Static Configuration Parameters: $N_{SP} = 4$ and $N_G = 2 \sim 64$

(b) Static Configuration Parameters: $N_{SP} = 16$ and $N_G = 2 \sim 64$

(c) Static Configuration Parameters: $N_{SP} = 64$ and $N_G = 2 \sim 64$

(d) Static Configuration Parameters: $N_{SP} = 2 \sim 64$ and $N_G = 4$

(e) Static Configuration Parameters: $N_{SP} = 2 \sim 64$ and $N_G = 16$

(f) Static Configuration Parameters: $N_{SP} = 2 \sim 64$ and $N_G = 64$

Fig. 18 Overall resource consumption of FPGA to implement CQPH for different configuration parameters.

equal to 1 tuple/cycle, CQPH can process up to 156 million tuples/second. As for latency, CQPH can respond in the order of microseconds with a cycle time of 6.37 ns. These data lead us to the conclusion that the proposed design can accomplish both high-throughput (over 150 million tuples/second) and low-latency (in the order of microseconds) requirements of the application.

It is also important to emphasize that the maximum clock frequency remains unaffected by the increasing number of N_{SP} or N_G. The fact that CQPH can still sustain a given clock rate is a good indication for the scalability. In other words, by increasing N_{SP} or N_G, we can easily increase the flexibility and the workload capacity of CQPH. For example, if we use a larger FPGA and increase N_G value, the maximum number of groups can be increased for GROUP-BY aggregate queries. Moreover, the number of aggregation pipelines (*i.e.*, N_G) determines the upper limit on the number of parallel queries that can be executed by CQPH at one time. For instance, if we assume simple aggregate queries (without a GROUP-BY clause) and increase N_G from 64 to 128, CQPH can simultaneously execute up to 128 parallel queries without sacrificing the performance (*i.e.*, throughput).

5.3.2 Experimental Measurement

A key aspect of using an FPGA for data stream processing is its flexibility that enables us to insert custom logic into an existing data path [5]. For example, CQPH can be tightly integrated with the physical network interface [28] inside an FPGA. Our experiments are based on a KC705

```
SELECT Symbol, <AGGREGATE>, Time
  FROM StockTick[RANGE <WIN_RANGE> seconds
                 SLIDE <WIN_SLIDE> seconds
                 WATTR Time]
 WHERE Symbol in (<SYMBOL_LIST>)
 GROUP BY Symbol
```

Fig. 19 Q_9: Template of a benchmark query.

Table 5 Query parameters

<AGGREGATE>	:	count(*), max(Price), min(Price), sum(Price), max(Volume), min(Volume), sum(Volume)
<SYMBOL_LIST>	:	subset of symbols *e.g.*, 'GOOG', 'AAPL', 'YHOO', etc.
<WIN_RANGE>	:	60, 300, 600, 1800
<WIN_SLIDE>	:	1, 5, 10, 30

board [27], which includes a Gigabit Ethernet interface and a 1 GB DDR3 SDRAM. The experimental system consists of a KC705 board and a host pc, which are directly connected by a dedicated Ethernet cable. In this experiment, we use random query sets each of which consists of 1, 2, 4, 8, 16, 32, or 64 queries. Each query is based on a template query given in Fig. 19. From Table 5, a single value is selected for each of <AGGREGATE>, <WIN_RANGE>, and <WIN_SLIDE>. As for <SYMBOL_LIST>, we choose 1, 4, or 16 different symbols at random, respectively.

Experiment 1. We have measured the effective throughput of CQPH ($N_{SP} = 64$ and $N_G = 64$) on the KC705 FPGA board. Results of the experiments show that sliding-window aggregate queries on the CQPH achieves an effective throughput up to around 760 Mbps for each query set. This is the upper bound of the available bandwidth that the network interface [28] can handle without packet loss. It should be emphasized that this is equivalent to nearly 6 million tuples per second, which means that the proposed setup can process significantly high tuple rates at wire-speed with zero packet loss.

Experiment 2. Since the Gigabit Ethernet is not sufficient to saturate CQPH, we use the DDR3 memory as a data source to emulate a 10 Gigabit Ethernet speed. In this setup, we have implemented a dedicated AXI master interface for CQPH, and a Xilinx AXI Interconnect core IP is used to connect the CQPH and the DDR3 memory. It should be mentioned that bus width of the AXI interconnect and data width of the AXI master interface are both set to 128 bits. In addition, we set 100 MHz of clock frequency for all hardware components including *(i)* the AXI interconnect, *(ii)* the AXI master interface of the CQPH, and *(iii)* the CQPH itself to achieve over 10 Gbps throughput; namely, the theoretical peak throughput of this setup is 128 bits × 100 MHz = 12,800 Mbps (recall from Eq. (1)).

By design, CQPH can accept one input tuple per clock cycle; therefore, in order to achieve the theoretical peak performance, it is important to provide the CQPH with a new tuple every clock cycle. In practice, however, the AXI interconnect and the AXI master interface become a critical bottleneck due to the AXI protocol overhead. In fact, when input tuples are transferred from the DDR3 memory via the AXI interconnect, the CQPH can achieve over 10,400 Mbps effective throughput for each query set in our experiments. This corresponds to the memory access speed of the evaluation setup; thus, in effect, the CQPH is limited by the memory read speed of the AXI master interface. Nevertheless, these results suggest that the CQPH can still support even faster 10 Gigabit Ethernet at line rate when clocked at 100 MHz.

6. Conclusions

In this paper, we have presented Configurable Query Processing Hardware (CQPH), a highly optimized and minimal-overhead query processing engine for data streams. Evaluation results indicate that CQPH can deliver real-time response (in the order of microseconds) for high-volume data streams (over 150 million tuples per second). It is also indicated that CQPH provides linear scalability in terms of area with a constant clock rate. Finally, our experiments demonstrate wire-speed performance by directly manipulating the network packets. One direction for future work is to utilize the off-chip memory resources and integrate CQPH into a DSMS. Another direction is to implement CQPH on multiple FPGAs to achieve further scalability.

Acknowledgements

This work was partially supported by JSPS KAKENHI (26330061) and Grant-in-Aid for JSPS Fellows (26·10862).

References

[1] Y. Ahmad and U. Çetintemel, “Data stream management architectures and prototypes,” in Encyclopedia of Database Systems, pp.639–643, 2009.
[2] S. Babu, “Continuous query,” in Encyclopedia of Database Systems, pp.492–493, 2009.
[3] R. Mueller, J. Teubner, and G. Alonso, “Data processing on FPGAs,” Proc. VLDB Endow. vol.2, no.1, pp.910–921, 2009.
[4] R. Mueller, J. Teubner, and G. Alonso, “Glacier: A query-to-hardware compiler,” SIGMOD Conference, pp.1159–1162, 2010.
[5] R. Mueller, J. Teubner, and G. Alonso, “Streams on wires – A query compiler for FPGAs,” Proc. VLDB Endow., vol.2, no.1, pp.229–240, 2009.
[6] J. Teubner and R. Mueller, “How soccer players would do stream joins,” SIGMOD Conference, pp.625–636, 2011.
[7] M. Sadoghi, M. Labrecque, H. Singh, W. Shum, and H.-A. Jacobsen, “Efficient event processing through reconfigurable hardware for algorithmic trading,” Proc. VLDB Endow., vol.3, no.1-2, pp.1525–1528, 2010.
[8] M. Sadoghi, H. Singh, and H.-A. Jacobsen, “Towards highly parallel event processing through reconfigurable hardware,” DaMoN, pp.27–32, 2011.
[9] M. Sadoghi, R. Javed, N. Tarafdar, H. Singh, R. Palaniappan, and H.-A. Jacobsen, “Multi-query stream processing on FPGAs,” ICDE, pp.1229–1232, 2012.
[10] W.G. Aref, “Window-based query processing,” in Encyclopedia of Database Systems, pp.3533–3538, 2009.
[11] M. Najafi, M. Sadoghi, and H.A. Jacobsen, “Flexible query processor on FPGAs,” Proc. VLDB Endow., vol.6, no.12, pp.1310–1313, 2013.
[12] S. Krishnamurthy, C. Wu, and M.J. Franklin, “On-the-fly sharing for streamed aggregation,” SIGMOD Conference, pp.623–634, 2006.
[13] 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,” CANDAR, pp.112–121, 2013.
[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, CPSY2012-74, 2013.
[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,” MCSoC, pp.55–60, 2013.
[16] Y. Oge, T. Miyoshi, H. Kawashima, and T. Yoshinaga, “Design and implementation of a handshake join architecture on FPGA,” IEICE Trans. Inf. Syst., vol.E95-D, no.12, pp.2919–2927, Dec. 2012.
[17] Y. Oge, T. Miyoshi, H. Kawashima, and T. Yoshinaga, “A fast handshake join implementation on FPGA with adaptive merging network,” SSDBM, pp.44:1–44:4, 2013.
[18] J.-B. Qian, H.-B. Xu, Y.-S. Dong, X.-J. Liu, and Y.-L. Wang, “FPGA acceleration window joins over multiple data streams,” Journal of Circuits, Systems, and Computers, vol.14, no.4, pp.813–830, 2005.
[19] C. Dennl, D. Ziener, and J. Teich, “On-the-fly composition of FPGA-based SQL query accelerators using a partially reconfigurable module library,” FCCM, pp.45–52, 2012.
[20] C. Dennl, D. Ziener, and J. Teich, “Acceleration of SQL restrictions and aggregations through FPGA-based dynamic partial reconfiguration,” FCCM, pp.25–28, 2013.
[21] J. Teubner, L. Woods, and C. Nie, “Skeleton automata for FPGAs: Reconfiguring without reconstructing,” SIGMOD Conference, pp.229–240, 2012.
[22] J. Teubner, L. Woods, and C. Nie, “*XLynx* – An FPGA-based XML filter for hybrid XQuery processing,” ACM Trans. Database Syst.,

vol.38, no.4, pp.23:1–23:39, 2013.
[23] 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.
[24] J. Li, D. Maier, K. Tufte, V. Papadimos, and P.A. Tucker, "Semantics and evaluation techniques for window aggregates in data streams," SIGMOD Conference, pp.311–322, 2005.
[25] 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.
[26] L. Woods, Z. István, and G. Alonso, "Ibex – An intelligent storage engine with support for advanced SQL off-loading," Proc. VLDB Endowment, vol.7, no.11, pp.963–974, 2014.
[27] Xilinx, Inc., "Kintex-7 FPGA KC705 Evaluation Kit," http://www.xilinx.com/products/boards-and-kits/ek-k7-kc705-g.html [Online; accessed 18-May-2015].
[28] e-trees.Japan, Inc., "e7UDP/IP IP-core." http://e-trees.jp/index.php/en/products/e7udpip-ipcore/ [Online; accessed 18-May-2015].
[29] L. Liu and M.T. Özsu, eds., Encyclopedia of Database Systems, Springer US, 2009.

Yasin Oge received his B.E. degrees in Computer Engineering and Telecommunication Engineering (Double Major) from Istanbul Technical University, Turkey in 2010. Since April 2011, he has been studying information network systems at the Graduate School of Information Systems, University of Electro-Communications (UEC), Japan. He received his M.E. degree in 2012, and is currently working toward the Ph.D. degree at the same university.

Masato Yoshimi recieved the B.E., M.E., and Ph.D. degrees from Keio University, Japan, in 2004, 2006, and 2009, respectively. He was Assistant Professor in Doshisha University by 2012. He is currently Assistant Professor in Graduate School of Information Systems, University of Electro-Communications. His research interests include the areas of reconfigurable computing and parallel processing.

Takefumi Miyoshi received his B.E., M.E., and D.E. degrees from Tokyo Institute of Technology in 2003, 2005, and 2007, respectively. He is currently an engineer in e-trees.Japan, Inc., and a founder of Wasalabo, LLC. His research interests include compiler techniques and co-design of hardware and software.

Hideyuki Kawashima received Ph.D. from Science for Open and Environmental Systems Graduate School of Keio University, Japan. He was a research associate at Department of Science and Engineering, Keio University from 2005 to 2007. From 2007 to 2011, he was an assistant professor at both Graduate School of Systems and Information Engineering and Center for Computational Sciences, University of Tsukuba, Japan. From 2011, he is an assistant professor at Faculty of Information, Systems and Engineering, University of Tsukuba.

Hidetsugu Irie is an associate professor of computer architecture at Graduate School of Information Science and Technology, the University of Tokyo. His research interests include microarchitectures, and human-computer interactions. He has a Ph.D. in Information Science and Technology from The University of Tokyo. He was a researcher of Japan Science and Technology Agency from 2004 to 2008, was an assistant professor of The University of Tokyo from 2008 to 2010, and was an associate professor of The University of Electro-communications from 2010 to 2015. He is a member of ACM, IEEE, and IPSJ.

Tsutomu Yoshinaga received his B.E., M.E., and D.E. degrees from Utsunomiya University in 1986, 1988, and 1997, respectively. From 1988 to July 2000, he was a research associate of Faculty of Engineering, Utsunomiya University. Since August 2000, he has been with the Graduate School of Information Systems, UEC, where he is now a professor. His research interests include computer architecture, interconnection networks, and network computing. He is a member of ACM, IEEE, and IPSJ.