Flexible Query Processor on FPGAs

Mohammadreza Najafi
MSRG
Technical University Munich

Mohammad Sadoghi
IBM T. J. Watson
Research Center

Hans-Arno Jacobsen
MSRG
Technical University Munich

ABSTRACT

In this work, we demonstrate Flexible Query Processor (FQP), an online reconfigurable event stream query processor. FQP is an FPGA-based query processor that supports select, project and join queries over event streams at line rate. While processing incoming events, FQP can accept new query expressions, a key distinguishing characteristic from related approaches employing FPGAs for acceleration. Our solution aims to address performance limitations experienced with general purpose processors needing to operate at line rate and lack of on the fly reconfigurability with custom designed hardware solutions on FPGAs.

1. INTRODUCTION

There is a growing interest in accelerating data management and event stream processing with FPGAs [6, 7, 8]. These approaches are characterized by a processing pipeline that synthesizes static (sets of) queries into circuits operating on an FPGA (i.e., a field-programmable gate array). The synthesis step often takes on the order of hours. which is the norm for FPGA designs, but is too inflexible for modern-day data processing needs, which requires the application to be able to change the query on the fly, without having to wait hours for the synthesis computations to complete.

Moreover, many existing approaches to accelerating data stream processing through FPGAs [2, 4, 5] assume that for processing query modifications, the incoming event stream is halted, the new query or query set are synthesized into configuration information for the FPGA, and the new configuration is uploaded onto the FPGA, before processing the event stream resumes. While synthesis of queries into circuits and processing of queries may overlap, a significant amount of time is required to halt, re-configure, and resume operation, which may take up to several minutes. More importantly, this style of processing requires extra logic for buffering, handling of dropped events, requests for re-transmissions, and additional data flow controlling tasks, which renders this style of processing difficult in practice. Also, these concerns were often ignored in the above listed approaches.

A common limitation most FPGA-based approaches suffer from, relates to the inherent complexity of the design synthesis process itself. This process and its inner workings like, for example, logic-optimization and technology mapping, are NP-hard problems, and all exiting synthesis algorithms are heuristic in nature [1], which leads to the drastic increase in synthesis time as applications grow in size. Therefore, approaches that require fewer commitment in re-synthesis are preferred because they offer far less amortized query modification overhead. Furthermore, a design that requires synthesis only once, may benefit from additional optimization of the input query set, which may be applied to achieve a closer to an optimal solution.

In this demonstration, we aim to fill the gap between software solutions which provide the greatest degree of flexibility in query modification needs, as opposed to hardware solutions which offer massive performance gains, by designing FQP that accepts new queries in an online fashion without disrupting the processing of incoming event streams. While supporting query modifications at run-time is almost trivial for software-based techniques, they are highly uncommon for custom hardware-based approaches, such as FPGAs, and have so far not received much attention in the growing body of work on accelerating data processing with FPGAs.

FQP is based on an *online reconfigurable query engine* (ORQE) that is a modifiable hardware circuit, which is comprised of a number of "online programmable" sub-blocks (referred to as OP-Blocks) that are composed to implement the query semantics. The OP-Block enables online changes to queries based on a number of parameters, including variable tuple size, projection, selection, join conditions, and query window size.

In the design of FQP, we have dealt with a number of challenges as summarized in the following. Often a static FPGA-based query processor must over-provision resources to handle the largest expected (intermediate) tuple size, which under-utilizes system resources. Secondly, the potential change in tuple size between the join operation's inputs and output adds new challenges, especially, when there is the need to use the join result as input for other operations. The situation worsens when needing to support nested joins. Thirdly, FQP is designed to support a number of OP-Block sets. Internally, each OP-Block set consists of several chained OP-Blocks. OP-Block sets are connected using a configurable bridge. Modifying (or reprogramming) an OP-Block within a set requires visiting all OP-Blocks in the chain until the target OP-Block is reached, which imposes challenges for mapping operators onto an OP-Block set. Finally, a robust method is needed to route and coordinate events across a set of chained OP-Blocks and to selectively bypass certain blocks.

The contributions of this work are four-fold: (1) We develop FQP that unlike the state-of-the-art enables online changes of queries without interrupting query processing over incoming event streams and without the need to resynthesise the design. (2) We unify and share the underlying storage buffer for both data and operator parameters of a query. (3) We support variable tuple sizes by proposing a segment-at-a-time processing model, namely, an abstraction that divides a tuple into smaller chunks that are streamed and processed as a consecutive set of segments. This strategy avoids the

Proceedings of the VLDB Endowment, Vol. 6, No. 12

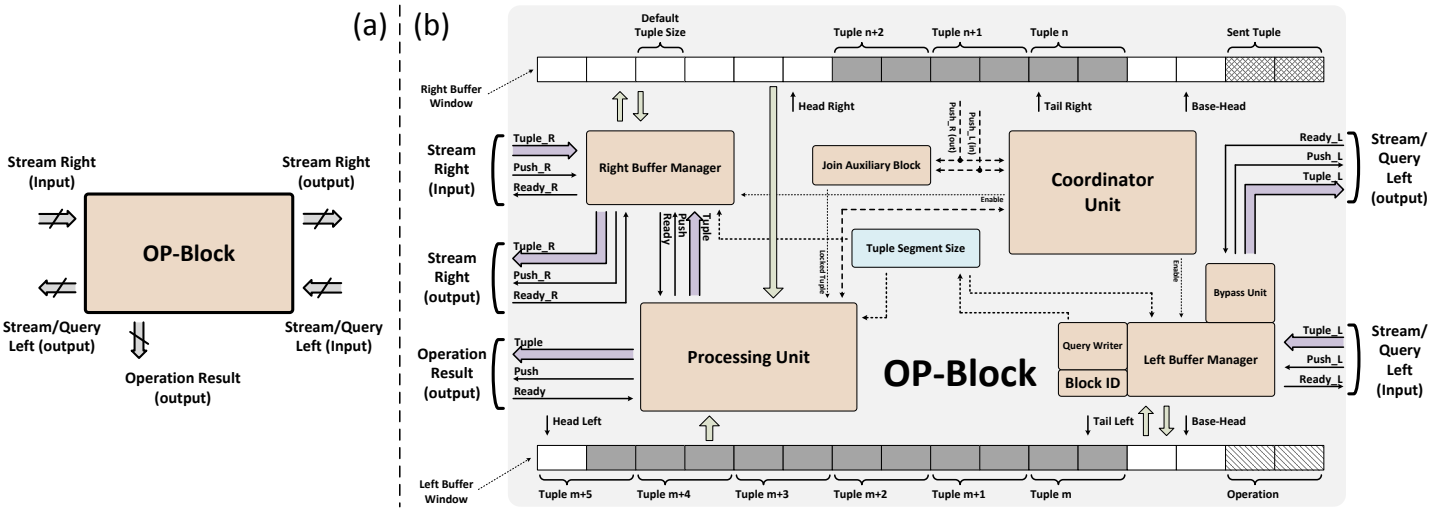

Figure 1: Online programmable block (OP-Block)

need for over-provisioning of hardware resources. (4) We propose a generic topology skeleton for chaining OP-Blocks in order to efficiently react to query workload changes through reprogramming.

2. BACKGROUND & RELATED WORK

An FPGA is a semiconductor device with programmable lookup-tables that are used to implement truth tables for logic circuits with a small number of inputs (on the order of 4 to 6, typically). FPGAs may also contain memory in the form of flip-flops and block RAMs (BRAMs), which are small memories (on the order of a few kilo-bits), that together provide a small storage capacity but a large bandwidth for circuits in the FPGA. Thousands of these building blocks are connected with a programmable interconnect to implement large circuits.

Related approaches are focused on accelerating event stream processing and SQL query processing on FPGAs.

Sadoghi *et al.* [6] developed a design to accelerate the evaluation of Boolean expression queries on FPGAs. Although their design allows changes to queries that are stored in off- and on-chip memories, methods for supporting online query changes were not discussed in-depth. Furthermore, Sadoghi *et al.* [5] proposed a Rete-based multi-query processing engine on FPGAs by developing custom circuit blocks for join, select, and project operations that exploit both inter- and intra-parallelism of queries. However, their work aimed to support the processing of event streams over a static set of queries and did not allow the application to change queries without re-synthesizing the circuit.

Teubner *et al.* [8] proposed an FPGA-based stream processing engine for realizing XML projections that maintains XML path expressions in FPGA on-chip memory (i.e., in BRAM). This allows to dynamically add new XPath expressions or modify existing XPath expressions without the need to re-synthesis. While their work shares the similar objective of achieving flexible query processing, their focuses was limited to XML projections, while we focus on SQL-like query processing. Furthermore, Teubner *et al.* [7] proposed a design for implementing a stream join on FPGAs. Their design aimed to efficiently evaluate a single join operation over two event streams; however, their design is static and lacks the flexibility for online changes to queries.

3. STREAM PROCESSING MODEL

Our event stream data processing model follows an attribute-value pair form, which closely resembles a database tuple, but, unlike traditional databases, we do not assume a fixed schema for the event data stream. Similarly, our event stream language follows the traditional database SPJ queries including selection (σ), projection (π), and join ($\bowtie$, $\bowtie_\theta$) operations.

In fact, we adapt PADRES SQL (PSQL) [3], an expressive SQL-based declarative language for registering continuous queries against event streams over a count-based sliding window model. Essentially, the sliding window is a snapshot of an observed finite portion of the event stream. Formally, the stream processing model is defined as follows: *Given a stream of events and a set of SPJ queries, execute the queries continuously over the event stream and output the resulting tuples.*

In what follows, we use the term event and tuple interchangeably.

4. FLEXIBLE QUERY PROCESSOR

FQP is comprised of three key modules. (1) A reprogrammable circuit, namely, the OP-Block that is able to realize selection, projection, and join functionalities. (2) The online reconfigurable query engine (ORQE) that manages the interactions among sets of OP-Blocks and an architectural skeleton that defines how OP-Blocks are connected. Different instances of FQP may comprise different connection topologies among OP-Blocks or may be configured with a different number of OP-Blocks per set of OP-Blocks. (3) The query assigner module is responsible for scheduling and mapping queries onto the available OP-Blocks.

4.1 OP-Block

The OP-Block is the basic building block of ORQE that is programmable to implement either a selection and projection operating on a single event stream or a join operating on two incoming event streams. OP-Blocks enable the efficient utilization of the FPGA's resources while offering the ability to support online query changes.

The internal structure of an OP-Block is detailed in Figure 1. Event streams flow between input and output ports of an OP-Block. For example, for realizing a join operation, the right join input stream flows through an OP-Block via the *Stream Right (input)* port to the *Stream Right (output)* port, while the left input stream flows through via the *Stream/Query Left (input)* port to the *Stream/Query Left (output)* port. The signal that triggers changes to a query always arrives at the *Stream/Query Left (input)* port. In contrast, for implementing selection and projection operations, only a single event stream is needed, and the stream always flows from the *Stream Right (input)* to the *Stream Right (output)* port.

There are two notable properties reflected in our OP-Block design. First, the advantage of an OP-Block compared to creating a specialized non-reusable join, selection, and projection circuitry, i.e., the challenge to achieve the right balance between a general-purpose processing unit and many (over-)provisioned specialized processing units. Second, identifying a minimum set of OP-Block

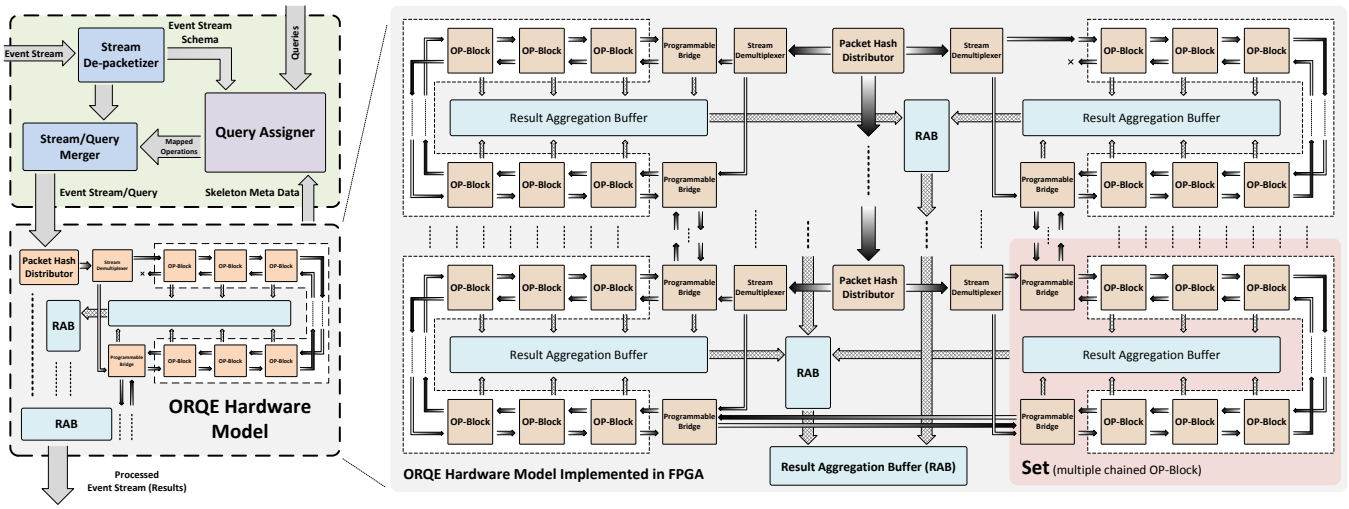

Figure 2: Flexible query processor

features that are necessary to allow OP-Blocks to serve as a basic building block of our hardware query engine.

We recognize that the join operation is inherently more costly than selection and projection operations, both from a design complexity and from an amount of required resources point of view, e.g., preallocating two window buffers when joining two event streams. In absence of OP-Blocks, if queries were modified to include an additional join operation, while there are only unused, pre-allocated projection/selection blocks on the board, then this modification would have to be rejected and a full re-synthesis of all queries would be required despite the available, unused and idle resources on the board – the *block over-provisioning problem*. Nevertheless, one could argue that an over-provisioning is the answer to this problem at the cost of ignoring resource underutilization. However, this over-provisioning raises yet another challenge in hardware because adding nested joins or new projection potentially changes the width of the resulting tuple, thereby forcing an over-provision of precious wiring (input and output ports) of each block in order to accommodate the increasing and variable tuple size – the *wire over-provisioning problem*.

Our proposed OP-Block has two novel features for solving these above-mentioned challenges. To address the block over-provisioning problem, the internal structure of an OP-Block is designed to be reusable, either as a join operation, or multiple selection/projection operations. An OP-Block instantiates two separate buffer windows (right and left) which are necessary for a join operation. In a join operation, the right window is allocated to the right event stream, and the left buffer window is used to store the left event stream. A join operation fills a portion (i.e., segment) of the left window, and the rest is devoted to the left event stream. Similarly, for selection and projection, instead of allocating additional buffers, our design uses buffers in the left window to store a variable number of selection and projection conditions (limited by the size of the left window) to prevent wasting FPGA resources. The wire over-provisioning limitation is addressed by proposing a segment-at-a-time processing model in order to limit the width of input and output ports of OP-Blocks. Each tuple is divided into a set of consecutive segments, where the segment is the smallest processing unit and the segment size is denoted by ts. Using the segment-at-a-time model, we are able to develop tuple-size independent circuitry that supports any arbitrary tuple size at runtime without the need to resynthesize the design.

4.2 OP-Block Internal Architecture

The internal structure of an OP-Block is shown in Figure 1. Each OP-Block is (re)programmable to support operators such as selection, project, or join. When a new tuple enters an OP-Block, the tuple undergoes three steps. First, existing tuples in OP-Block buffers (if any) and operator conditions are fetched from the OP-Block's on-chip BRAM memory. Storing the operator conditions in BRAM is one of the enabling features for dynamically changing queries online. Second, depending on the encoded operator, the relevant comparison or evaluation is applied to the tuple. Third, any resulting tuples (e.g., if tuple passes the selection filtering condition) are written to the output buffer, which are also BRAM-resident.

A unique feature of the OP-Block is to support variable size tuples. We refer to this feature as *segment-at-a-time processing model*, in which tuples are pushed to OP-Blocks in chunks, driven by the tunable tuple segment size ts. If the tuple is longer than the OP-Block's configured segment size, then the tuple is read in several iterations, one segment is read in each iteration. For example, operations on a tuple that has size equal to $2 \times ts$, forces the OP-Block to process the tuple one segment at a time. In the first iteration, the operation is applied on the first segment of the tuple and in the second iteration the operation is executed over the second segment of the tuple. In each iteration, the result of the operation is fed to the output buffer. In the last iteration, a valid flag is added to the last segment of the result. This flag is necessary because the validity for each tuple segment result can not be determined until the very end of the execution (as soon as the result of the last segment is produced). In other words, this valid flag specifies the validity of the tuple as a whole. In addition, the segment-at-a-time processing model prevents long stalls and buffering that is required to deal with large tuple sizes; thus, our model reduces the interruption flow of tuples in the hardware circuitry. Our segment-at-a-time processing model of OP-Blocks enables fixed wiring in and out of an the OP-Block, which transforms an OP-Block to a re-usable and a generic SQL primitive construct that can be reprogrammed online.

The segment-at-a-time processing model favors predictable OP-Block output width size (i.e., the size of the tuples produced). In any hardware solution, there is always an upper-bound on the sustainable bandwidth. When the maximum sustainable bandwidth is reached (a highly plausible scenario for low selective queries), many new challenges are introduced. In particularly, to cope with the excess load, there is a need to provision intermediate buffers with variable width size due to variable tuple sizes, to trace variable-width data flow for re-transmission of data, and to place stalling mechanisms in order to prevent data loss. Our proposed segment-at-a-time processing model substantially simplifies these concerns and allows generic, tuple-size independent circuitry to be shared among saturated reprogrammable OP-Blocks.

By generalizing our segment-at-a-time processing model, we are

also able to support join operations with a wide and variable window size that go beyond the OP-Block's predetermined buffer sizes. To support arbitrary window sizes, we instantiate a chain of OP-Blocks, configured with identical join operations. In each OP-Block, a portion of the join's right window is devoted to the previously sent tuple from one OP-Block to its right. This is to keep two adjacent OP-Blocks synchronized, while an OP-Block transmits and receives a tuple to and from its neighbouring blocks.

In the following a brief description of the internal sub-blocks of an OP-Block is presented.

- **Buffer Manager** is responsible for the receiving and transmitting of tuples in and out of the left and right buffer windows. To support variable tuple sizes, this module takes more cycles to operate on each segment of a tuple.
- **Coordinator Unit** is the global synchronizer among sub-blocks in an OP-Block. It manages sub-blocks when receiving and processing new tuples.
- **Processing Unit** is responsible for comparing and evaluating tuples.
- **Join Auxiliary Block** monitors ports to and from the OP-Blocks' neighbour to coordinate simultaneous transmissions and receptions of tuples for join operations that straddle OP-Blocks. This sub-block also notifies the *processing unit* to include the *tuple* segment in the comparison process.
- **Bypass Unit** deals with the left event stream and query operations. The *Stream/Query Left (input)* port is used for programming of OP-Blocks as well as receiving the left event stream. The responsibility of this unit is to bypass query operations that do not belong to this OP-Block.
- **Block ID** is a statically assigned identifier to uniquely identify each OP-Block. This ID is also used to program operations for chained OP-Blocks.
- **Query Writer** is responsible for reprogramming an OP-Block and setting tuple segment sizes.

4.3 ORQE Skeleton Architecture

OP-Blocks can be attached in any arbitrary configuration to construct an OP-Block skeleton with a complex topology. However, in this work, we propose a basic expandable structure as the initial skeleton. Any particular topology could be optimized for a set of known queries and workload characteristics, but our objective is to devise a general topology that can easily be maintained and reprogrammed. Our ORQE block diagram is illustrated in Figure 2 consisting of the following internal blocks.

- **Packet Hash Distributor** is the gateway for receiving incoming event streams/queries and, based on its internal mapping table, decides to either process the request in its connected OP-Blocks, or to delegate the request to the next *Packet Hash Distributor*. The proposed architecture for distributing events/queries is suitable to keep latencies bounded while transmitting operations to their relevant OP-Blocks. This unit can also be configured to selectively route specific events to given OP-Blocks; however, in its default mode, it relies on broadcasting all incoming events.
- **Stream Demultiplexer** guides left the event stream and query operations to the left window buffer of the chained OP-Blocks and the right event stream to the right window buffer.
- **Programmable Bridge** is programmed to connect OP-Blocks to form a chain (e.g., to implement join operation with arbitrary window size).
- **Result Aggregation Buffers** are responsible for gathering event stream processing results.

4.4 Query Assigner

The *query assigner* is responsible for assigning and instantiating new queries to our ORQE as shown in Figure 2. This block processes *queries*, an *event stream schema*, and *skeleton meta data* as input and produces *mapped operations* as output. *Queries* are submitted for processing. The *event stream schema* is the set of event schemas. The *skeleton meta data* holds the topology placement map of OP-Blocks.

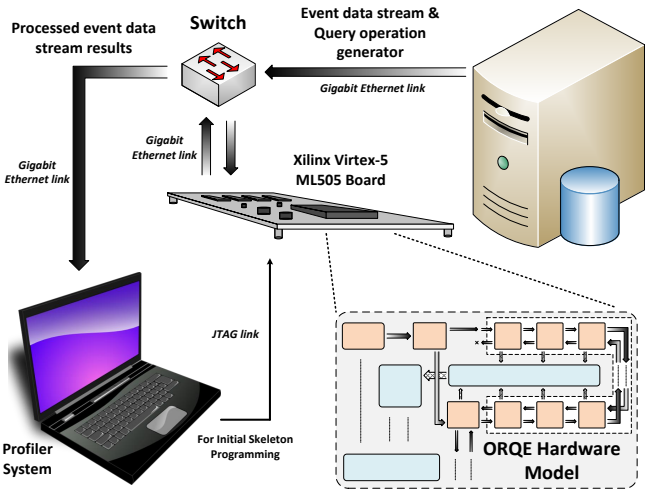

Figure 3: Demonstration setup

5. SOFTWARE DEMONSTRATION

We demonstrate the operation of FQP that processes event streams over user issued queries. Our platform (cf. Figure 3) provides a complete hardware solution residing on an FPGA, with related components (e.g., the query assigner) residing on the event stream generator machine.

The input is an event stream, and for evaluation, the system throughput is measured, i.e., the maximum sustainable input event rate. Events are encoded in UDP packets, which are transmitted over a directly connected 1 Gb/s Ethernet link. The direct, unshared Ethernet link reduces transmission reliability issues, except for packet drops that result from filling up of input/output buffers on the connected FPGA board, which then indicates the maximally sustainable processing rate.

Our setup includes a laptop that transmits the event stream in addition to the requested queries through a user interface over a 1 Gb/s Ethernet interface to FQP hosted on an Xilinx ML505 board featuring a Virtex-5 XC5VLX50T FPGA. The second laptop in Figure 3, namely the *profiler system*, is responsible for monitoring the processed event stream to gather processing statistics, which are shown in a graphical user interface, the *profiler utility*. A USB-JTAG link is used to upload the initial design onto the FPGA from the Xilinx ISE14.4 EDK development tool.

6. REFERENCES

[1] J. Cong and K. Minkovich. Optimality study of logic synthesis for LUT-Based FPGAs. *IEEE TCAD'07*.

[2] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: a stream database for network applications. In *SIGMOD'03*.

[3] H.-A. Jacobsen, V. Muthusamy, and G. Li. The PADRES event processing network: Uniform querying of past and future events. *it - Information Technology'09*.

[4] R. Mueller, J. Teubner, and G. Alonso. Streams on wires: a query compiler for FPGAs. *PVLDB'09*.

[5] M. Sadoghi, R. Javed, N. Tarafdar, H. Singh, R. Palaniappan, and H.-A. Jacobsen. Multi-query stream processing on FPGAs. In *ICDE'12*.

[6] M. Sadoghi, M. Labrecque, H. Singh, W. Shum, and H.-A. Jacobsen. Efficient event processing through reconfigurable hardware for algorithmic trading. *PVLDB'10*.

[7] J. Teubner and R. Mueller. How soccer players would do stream joins. In *SIGMOD'11*.

[8] J. Teubner, L. Woods, and C. Nie. Skeleton automata for FPGAs: reconfiguring without reconstructing. In *SIGMOD'12*.