Multi-Query Stream Processing on FPGAs

Mohammad Sadoghi, Rija Javed, Naif Tarafdar, Harsh Singh, Rohan Palaniappan, Hans-Arno Jacobsen
Middleware Systems Research Group (msrg.org)
University of Toronto, Canada

I. Introduction

The need for efficient real-time data analytics is an integral part of a growing number of data management technologies such as intrusion detection [1], algorithmic trading [6], and (complex) event processing [5]. What is common among all these scenarios is a predefined set of continuous queries and an unbounded event stream of incoming data that must be processed against the queries in real-time.

The challenges for today's real-time data analytics platforms are to meet the ever growing demands in processing large volumes of data at predictably low latencies across many application scenarios. The volume of traffic on the Internet has undergone an immense increase over the last decade which is apparent in deployments of high communication bandwidth links across the globe (e.g., OC192 at 9.92Gbit/s). While according to Gilbert's law, communication bandwidth is projected to double every 9 to 10 months, conventional computation system architectures are showing signs of saturation in terms of offering the necessary processing power to sustain demands imposed by future Internet bandwidth growths.

The need for more processing bandwidth is the key ingredient in enabling innovation in high-throughput real-time data analytics to process, analyze, and extract relevant information from streams of events. Therefore, as proliferation of data and bandwidth continues, it is becoming essential to expand the research horizon to go beyond the conventional software-based approaches and adopt other key enabling technologies such as reconfigurable hardware in form of Field Programmable Gate Arrays (FPGAs). An FPGA is a cost-effective hardware acceleration solution that has the potential to excel at analytics-based computations due to its inherent parallelism. FPGAs can exploit low-level data and functional parallelism in applications with custom, application-specific circuits that can be re-configured, even after the FPGA has been deployed. In addition, FPGAs can meet the required elasticity in scaling out to meet increasing throughput demands.

We propose an FPGA-based real-time data analytics platform that supports line-rate processing of data streams over a collection of continuous queries. Our contributions are three-folds. (1) We propose high-throughput, custom circuits to implement the relational algebra (i.e., selection, projection, and join) over a window of input events in order to effectively process a single SPJ (Select-Project-Join) query in reconfigurable hardware. The hardware implementation enables a high degree of parallelism and pipelining beyond the reach of software-based implementations. The custom circuits serve as a library of operators. (2) We introduce a novel multi-query optimization technique inspired from highly parallelizable rule-based system designs by mapping an SPJ-query into a Rete-like operator network [2]. We exploit the overlap among SPJ query plans by constructing a single global query plan to be executed in hardware. (3) We develop software-to-hardware multi-query processing techniques that map a set of SPJ queries into a Rete-like global query plan. Subsequently, the global plan is converted into Hardware Description Language (HDL) code using our "hardware library" of custom building blocks for the various relational algebra operators. These mapping techniques are akin to a compiler that can process a query expressed in our language into a custom circuit that processes event streams.

Moreover, in our design, we exploit parallelism while sustaining the onboard multi-giga bit throughput rates. First, owing to the inherent parallel nature of Rete-like processing, we synthesize custom logic to execute multiple query plans in parallel, while exploiting query plan overlaps. Second, owing to the potential for parallel processing within each relational operator, most notably the expensive join operation, we synthesize custom logic for the operator implementations.

II. Background & Related Work

An FPGA is a semiconductor device with programmable lookup-tables that is used to implement truth tables for logic circuits with a small number of inputs. FPGAs may also contain memory in the form of flip-flops and block RAMs (BRAMs), high-bandwidth on-chip memory.

Recent work has shown that FPGAs are a viable solution for building custom accelerated components. For instance, work in [6], [7] focused on atomic and stateless matching (i.e., select queries). However, our work concentrates on supporting a more general multi-query stream processing (stateful matching) specifically designed to accelerate the execution of SPJ queries. Alternatively, [4] presented an efficient FPGA implementation of a single query (without join) while [8] focused on the data flow for streaming join over a large window size that spans many processing cores. Our approach also differs from [4], [8] as we are primarily concerned with multi-query optimization (with joins computed over a moderate size window) using Rete-like processing networks, supporting a rich relational algebra over event data streams, and offering an unprecedented degree of inter- and intra-operator parallelism that is only available through low-level logic design.

III. FPGA Stream Processing Model

Our event stream data model is captured as attribute–value pairs, which closely resembles a database tuple, but, unlike traditional databases, we do not assume a fixed schemata for the event data stream. Similarly, our event stream language also follows traditional database SPJ queries including selection (σ_c), projection (π), and join ($\times$, $\bowtie_c$). In fact, we

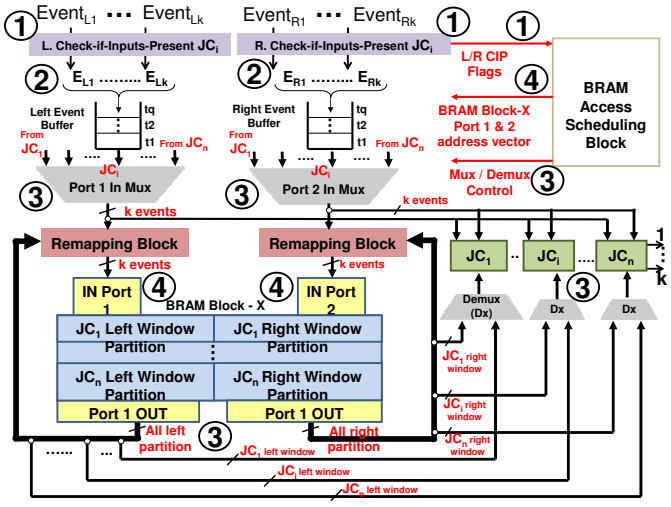

Fig. 1. Overview of parallel join processing

adapt PADRES SQL (PSQL) [3], an expressive SQL-based declarative language for registering continuous queries against event streams over an either time-based or 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 collection of continuous SPJ queries, the queries are continuously executed over the event stream.*

IV. Operator-to-Circuit Mappings

The first step to realize query processing on hardware is an efficient mapping of relational operators to custom circuits (custom processors). This mapping forms the basis of our query processing model on the FPGA. The operator mappings that we discuss are selection, projection, and join. In addition, we explore two circuit designs: the sequential and the parallelized. The sequential solution is tuned for scaling the number of supported queries while the parallel solution (focus of this work) is designed for achieving line-rate processing of event streams over a set of queries.

Selection Selection refers to the conditional test over attributes of an event. This test is a unary operation written as σ_c where c is a propositional formula over the logical operators $\vee$, $\wedge$ and $\neg$. A combinational circuit is used to implement this propositional formula, referred to as the *Selection Circuit* (*SC*). One of the key features of *SC* is that it evaluates the entire propositional formula in one clock cycle. To further accelerate the execution, in our parallelized scheme, we scale out the computation by replicating the *SC* block k times in order to evaluate up to k events in parallel and in one clock cycle. However, in our resource-aware sequential design, we create a single *SC* block and the necessary logic to serialize events to sequentially process the selection condition.

Projection Projection refers to the removal of certain attribute–value pairs from an event. Projecting out attributes is implemented as a combinational circuit, *Projection Circuit*, that uses a mask. A mask has as many bits as the number of attributes in an event such that all of its bits are set to one except those that correspond to the projected attributes. The

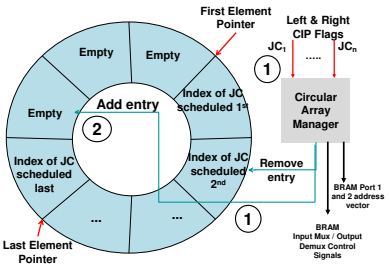

Fig. 2. BRAM Access Scheduling Block

circuit consists of two-input AND gates that are required to do a bitwise AND operation between an event and the mask.

Join At a high-level, the join operation ($\bowtie_c$) consists of two count-based (or time-based) sliding windows (left and right window) over two event streams and a join condition c (an arbitrary Boolean expression). The join operation follows the classical window-based-join semantics which is defined as a sequential procedure in two phases: (1) Upon arrival of a new event e at window w (either left or right), e is compared against every other event residing in the opposite window. And for every pair of events that satisfy the condition c, the two events are joined and added to the join-result stream. (2) The new event e overwrites the oldest event in w. In what follows, we first provide a brief background, then we focus on a novel transformation of this sequential software-oriented procedure into a highly parallelized hardware-oriented implementation by utilizing custom circuits coupled with local on-chip memory banks.

The join computation consists of the join condition c and the Phase 1 of the join semantics which together form the *Join Circuit* (*JC*); *JC* is also associated to a left and a right window. The *JC* block leverages on-chip BRAM memory as the medium for implementing the sliding window. Each block is independent of the others with its own address space and read and write ports. Furthermore, the BRAM port-width can be adjusted to sustain the necessary memory bandwidth, namely, reading and writing of the entire content of either a window (k events) or a BRAM block ($2k \times n$ events), in one cycle when relevant data is stored contiguously. This is referred to as k-way read and write, where k is the port-width. To support concurrent read and write, the BRAM is made dual-ported so that reading or writing is carried out on different ports. Finally, to fully utilize the available on-chip BRAM, we must coalesce sliding window buffers from up to n join operations into one of the many available BRAM blocks; thus, packing up to $2k \times n$ events into a BRAM block.

We propose key opportunities for intra- and inter-parallelized execution of window-based-join semantics. For intra-parallelism, first, it is observed that Phase 1 and 2 (of the join semantics) can be done in parallel because the read and write memory accesses are performed on opposite windows; consequently, no race condition occurs and no locking is needed as long as both phases are completed before accepting additional newly arriving events. Second, Phase 1 can be executed in parallel by comparing the new event e against every other event in the opposite window in one cycle. This is achieved by replicating the join condition circuitry k times and enabling k-way read memory access which in turn yields

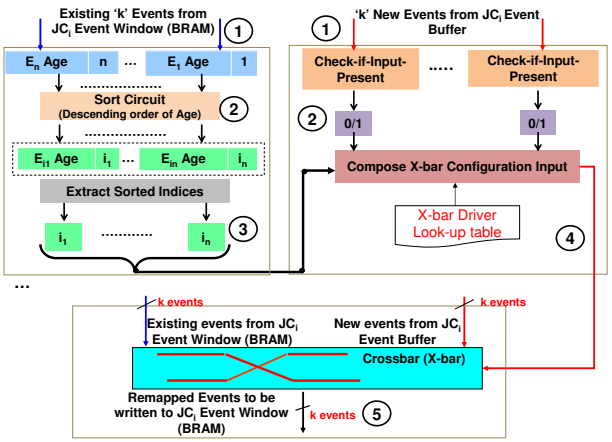

Fig. 3. Remapping Block

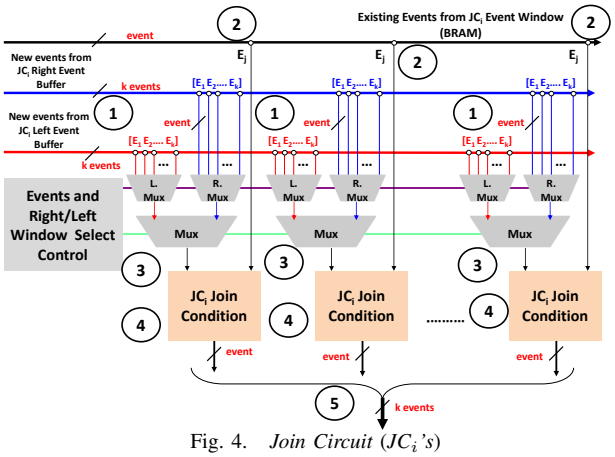

Fig. 4. *Join Circuit* (JC_i's)

up to k joined events. Also, Phase 1 and 2 can be extended to support up to k simultaneous new events for each window, yielding up to k^2 joined events in k cycles.

For inter-parallelism, we can scale up to n different join operations whose sliding windows are located in a shared BRAM block. Executing n joins in parallel is possible by replicating the entire join-semantics circuitry n times and enabling $(2kn)$-way read memory access. Notably, the machinery for implementing n joins is tightly coupled to a single BRAM block (i.e., promoting local memory access); thus, no central coordination is necessary among the various BRAM blocks and all can be active simultaneously.

We briefly present data and execution flow of each circuit. The high-level machinery for accepting up to k simultaneous events for each of the n join operations (intended for a single BRAM block) is captured in Fig. 1, which consists of the following inner-blocks. *BRAM Access Scheduling Block* (*BASB*) schedules access to sliding window buffer (cf. Fig. 2). *Remapping Block* (*RB*) advances sliding window by evicting the oldest event first (cf. Fig. 3). *Join Circuit* (*JC*) processes parallelized join (cf. Fig. 4).

The circuit in Fig. 1 captures the overall data and execution flow of n joins (in what follows we refer to the numbered steps for describing the figure). This circuit accepts as inputs k new events for either left or right window of every JC_i. The inputs are detected after passing through a combinational circuit called *Check-if-Inputs-Present* (*CIP*) (1). If any event detected for JC_i, then JC_i is marked as active and is sent to *BASB* for execution scheduling (1). All active *JC*s that simultaneously detect inputs are also scheduled to be executed in parallel. If active, JC_i cannot be executed immediately, then it is temporarily queued (2); otherwise, active *JC*s start receiving the contents of their windows and *JC*'s k new incoming events (3). Finally, in two parallel pipelines both active *JC*s are ran in parallel to carry out the join computation (Phase 1 of the join semantics) and for each active JC_i, *RB* is invoked to overwrite the oldest events with their corresponding k new incoming events (Phase 2 of join semantics) (4).

The *BASB* circuit (cf. Fig. 2) simply manages queuing (through a circular array) and scheduling active JC_i and orchestrating control signals for muxes and demuxes in order to access the BRAM's content (Steps 1-2).

RB (cf. Fig. 3) is responsible for evicting the oldest events when window w is full. After new events are detected by *CIP*, (1), events' ages are stored[1] in on-board registers (2). Next, the indices of the oldest events in w are extracted (3); these indices become the bases for configuring the crossbar for overwriting the oldest events with the newest ones (Steps 4-5).

The JC_i block (cf. Fig. 4) receives the content of its windows and the same k new events on every cycle (Steps 1-3). In each cycle, JC_i takes one of the new events and compares it against all the events in its corresponding window in parallel through replication of JC_i *Join Condition* k times (4). In each cycle up to k joined events are produced (5).

Time Complexity The selection and projection operators are implemented as combinational circuits and have complexity $O(1)$. The join operator complexity is rather involved. In particular, *RB* determines eviction of the oldest event and carries out the actual eviction in $O(1)$, which has to be repeated n times for each join, resulting in time complexity $O(n)$. Each JC_i evaluates k input events in $O(k)$ time (or $O(1)$ for a single event). The remaining components are executed in constant time such as *BASB* for BRAM access coordination; *CIP* for detecting input events; and muxes and demuxes for selecting and routing events. Hence, n join operations (within a BRAM block) can be done in parallel in $O(\texttt{max}(k, n))$.

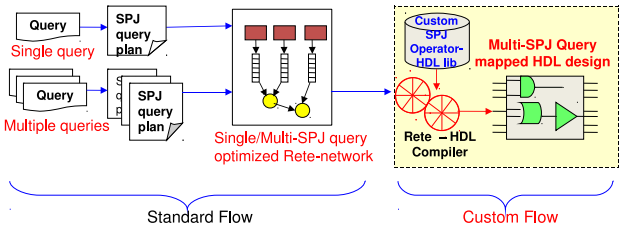

Fig. 5. Mutli-query optimization flow for HDL mapping

V. Multi-query-to-Circuit Mappings

In the previous section, we discussed how to map the building blocks of each query (i.e., the relation algebra) into circuits, thereby paving the way for efficiently processing a single SPJ query on hardware (cf. standard flow in Fig. 5). We now shift gears towards processing multiple queries efficiently in hardware, in which the novelty of our proposed approach is to go beyond executing a single optimized query plan on

[1]To sort efficiently in hardware, we use Bitonic sort, implemented as combinational circuits that requires a constant number of cycles for small input size, i.e., given d inputs, Bitonic sort has a comparator stage complexity of $O(\log^2 d)$ and requires $O(d \log^2 d)$ comparators.

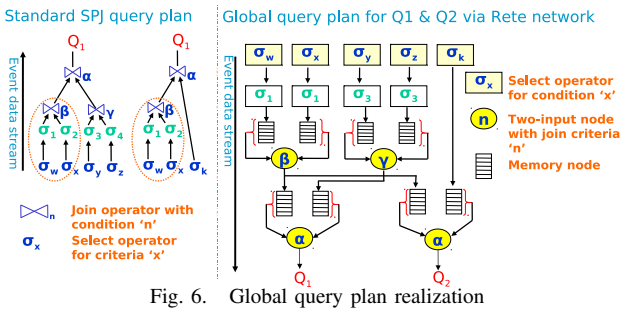

Fig. 6. Global query plan realization

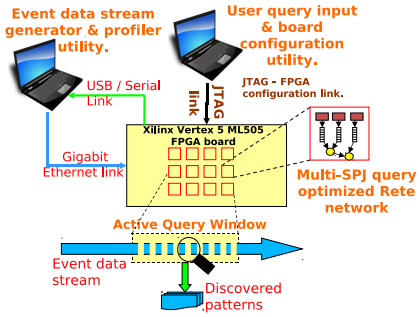

Fig. 7. Demo & evaluation setup

hardware and to support parallel processing of multiple queries on the FPGA (cf. custom flow in Fig. 5). An FPGA design is especially powerful in exploiting parallelism because any form of parallel execution can be directly mapped to logic circuits in hardware. In our implementation, this is accomplished by using a Rete-like event processing network to realize a single global query plan (cf. Fig. 6) that exploits the overlapping components among given SPJ query plans to further improve the resource utilization and execution of the global query plan on the FPGA.

A multi-query optimized event processing network is comprised of Rete-specific elements, e.g., *pattern detect nodes* and *join nodes*, that share functional resemblance with the key relational algebra operators, e.g., σ and $\bowtie$, respectively, which in turn constitute the elements of a standard SPJ query plan. Hence, a multi-query optimized event processing network, represented as an inverted global SPJ query plan, can ultimately be translated into a hardware design (as per custom flow in Fig. 5). The resulting hardware design is modular and in our implementation utilizes the three main rudimentary hardware building blocks (described in the previous section) that realize their relational algebra operator counterparts. The process of mapping a Rete-like graph representing a global SPJ query plan onto a Hardware Description Language (HDL) design (circuit) is captured in our custom flow (cf. Fig. 5) which includes a custom Rete-to-HDL compiler. The input to this compiler is a set of SPJ query plan(s) that are firstly used to build a multi-query optimized Rete network graph using the standard Rete algorithm. Secondly, the resulting Rete graph is decomposed to utilize appropriate HDL models from our custom HDL library for SPJ operators (cf. Section IV). Finally, predefined HDL design templates are referenced to build the final circuit that is targeted to execute the given input SPJ query(s) on the FPGA.

To further optimize the global query plan, we utilize a pipelined design that increases chip resource utilization by keeping all processing blocks active at all times. This design delivers a greater throughput than a non-pipelined counterpart at the cost of additional resources. To enable a pipelined design, additional buffering is required in order to avoid dropping events between pipeline stages; furthermore, additional logic circuits are required to orchestrate the flow of events downstream from one operator to the next. A detailed description of the inner workings of the pipelined design is omitted in the interest of the space.

VI. Demo Setup & Evaluation

We demonstrate our FPGA-based event processing platform targeted at processing event streams over a set of continuous queries at line-rate. Our platform provides an "active query window" (cf. Fig. 7) to analyze event streams over multiple queries in parallel while streams seamlessly pass through the platform. The filtered output stream is delivered to higher-level applications for further data analytics and monitoring purposes.

The input to our platform is a stream of events, and for our evaluations, we measure the system throughput, i.e., the maximum sustainable input event rate. Events are encoded in the packets of the transport protocol that transmits event data to the FPGA board. In our implementation, we use UDP as transport over a directly connected 1 Gb/s Ethernet link (cf. Fig. 7). Due to the direct, unshared Ethernet link, there are no severe transmission reliability issues to consider, except for the filling up of input/output buffers on the board. If no further events can be accepted by the board, packets are dropped and the maximal sustainable processing rate is achieved.

Our setup includes a first laptop that transmits an event data stream, over a 1 Gb/s Ethernet interface, to our platform hosted on an Xilinx Virtex 5 LXT ML505 FPGA board (cf. Fig. 7). In addition, a USB-JTAG link is employed to program the FPGA board through a second laptop loaded with the Xilinx ISE10.1 EDK development tool suite for design synthesis and bit stream generation. Thus, we upload the HDL design based on user input queries, generated through a SQL-like interface, using our custom Rete-HDL compiler (cf. Fig. 5). Finally, along with system level runtime statistics (e.g., query throughput and event buffer utilization), the query output is retrieved from the board via a dedicated secondary link for displaying the results in the "profiler utility" (cf. Fig. 7).

References

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

[2] A. Gupta, C. Forgy, and A. Newell. High-speed implementations of rule-based systems. *ACM Trans. Comput. Syst.'89*.

[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. *VLDB'09*.

[5] M. Sadoghi and H.-A. Jacobsen. BE-Tree: An index structure to efficiently match Boolean expressions over high-dimensional discrete space. In *SIGMOD'11*.

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

[7] M. Sadoghi, H. Singh, and H.-A. Jacobsen. Towards highly parallel event processing through reconfigurable hardware. In *DaMoN'11*.

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