Glacier: A Query-to-Hardware Compiler

Rene Mueller
rene.mueller@inf.ethz.ch

Jens Teubner
jens.teubner@inf.ethz.ch

Gustavo Alonso
alonso@inf.ethz.ch

Systems Group, Department of Computer Science, ETH Zurich, Switzerland

ABSTRACT

Field-programmable gate arrays (FPGAs) are a promising technology that can be used in database systems. In this demonstration we show *Glacier*, a library and a compiler that can be employed to implement streaming queries as hardware circuits on FPGAs. *Glacier* consists of a library of compositional hardware modules that represent stream processing operators. Given a query execution plan, the compiler instantiates the corresponding components and wires them up to a digital circuit. The goal of this demo is to show the flexibility of the compositional approach.

Categories and Subject Descriptors

H.2 [**Database Management**]: Systems; C.5 [**Computer System Implementation**]: VLSI Systems

General Terms

Design

Keywords

FPGA, stream processing, query compilation

1. INTRODUCTION

The emergence of multi-core technology has largely eliminated the CPU bottleneck in computing systems—often only to hit the next architectural limit of commodity hardware. *I/O bottlenecks* to external components, *high power consumption*, and *memory bottlenecks* have become critical issues in existing systems.

Field-programmable gate arrays (FPGAs) are seen as a possible escape out of this dilemma. Different research prototypes [3, 4, 6, 7] as well as actual products [1, 2] indicate the high potential of FPGA technology for the use in database systems.

In simple terms, FPGAs are chip devices that provide a configurable pool of logic resources. These logic resources can be used to implement arbitrary digital circuits. A key advantage of FPGAs over custom silicon chips is that the configuration is not fixed. An FPGA chip can be reprogrammed "in the field" even after the chip has been integrated into an appliance.

From systems design perspective FPGAs have several interesting properties. First of all, the large number of logic cells (758,000 cells for a recent Xilinx Virtex-6 XC6VLX760 chip [8]) offers a level of hardware parallelism that cannot be matched on traditional multi-core architectures or a graphics processors (GPUs). FPGAs operate at a significantly lower clock frequency and, hence, excel with extremely low power consumption. Finally, unlike CPUs or GPUs, they can be directly integrated into the data path. For instance, FPGAs can be used to filter or aggregate data arriving from the network or from disks [1, 4].

In [5], we presented *Glacier*, a component library and a compiler that translates streaming queries into hardware circuits. The component library consists of compositional modules that represent stream processing operators. Given a query, the compiler instantiates the necessary modules and connects them to a digital logic circuit which is then translated into an FPGA configuration in a traditional FPGA design flow. A virtue of *Glacier* is its full compositionality: *Glacier* accepts arbitrary compositions of the supported algebra operators (lined up in Table 1). In this demonstration we present our implementation of *Glacier* and show the flexibility of this approach using different queries.

$\pi_{a_1,\ldots,a_n}(q)$	projection
$\sigma_a(q)$	select tuples where field a contains `true`
$\circledast_{a:(b_1,b_2)}(q)$	arithmetic/Boolean operation $a = b_1 \star b_2$
$q_1 \cup q_2$	union
$agg_{b:a}(q)$	aggregate agg using input field a, $agg \in \{$`avg`, `count`, `max`, `min`, `sum`$\}$
$q_1\ \mathsf{grp}_{x\|c}\ q_2(x)$	group output of q_1 by field c, then invoke q_2 with x substituted by the group
$q_1 \boxplus^{t}_{x\|k,l} q_2(x)$	sliding window with size k, advance by l; apply q_2 with x substituted on each wind.; $t \in \{\text{time}, \text{tuple}\}$: time-, or tuple-based
$q_1 \circledcirc q_2$	concatenation; position-based field join

Table 1: Supported streaming algebra (a, b, c: field names; q, q_i: sub-plans; x: parameterized sub-plan input).

SIGMOD'10, June 6–11, 2010, Indianapolis, Indiana, USA.

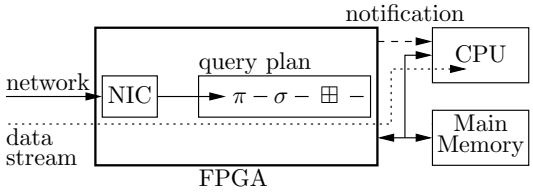


Figure 1: System Architecture: Stream engine between network interface and CPU.

1.1 Processing of Financial Streams

We adopt the use case from a collaboration with a Swiss bank. In their financial trading application, high-volume data streams from the Eurex stock exchange need to be processed in real time. The streams arrive over the network as a sequence of several ten thousand UDP network datagrams every second. Yet, low latency—particularly under high load—are critical and hard to reach with traditional software-based stream processing engines. In our approach we use *Glacier* to generate hardware execution plans and execute the queries on an FPGA. The FPGA is placed between the network interface and the higher layers of the trading application running on a traditional server.

Our demonstration uses a stripped-down version of the original data stream. A data generator lets us generate streams of configurable data rates that follow this schema.

1.2 System Integration

Glacier provides interface components that can accept data from the CPU or the network and write result stream back onto the network or back to the CPU. The network origin as well as the low-latency demands suggest a configuration like the one shown in Figure 1, where the FPGA is directly connected to the physical network interface. Parts of the UDP/IP network controller are implemented inside the FPGA fabric. After reception, data from the network is directly fed into the hardware implementation of a database query plan. Only the final query result is sent to a commodity system, e.g., to serve a user application.

2. QUERY-TO-HARDWARE COMPILER

We described the inner workings of the *Glacier* compiler and its component library in [5]. Here we only summarize those characteristics of *Glacier* that are the focus of this system demonstration.

Currently, *Glacier* supports all stream queries that can be composed from the streaming algebra listed in Table 1. In addition to these surface-level stream processing operators, the component library of *Glacier* provides interfaces for the network and the CPU, latency balancing operators, concatenation of streams and means to evaluate Boolean and arithmetic expressions.

2.1 Example Query

We illustrate the characteristics of *Glacier* based on the simple example query shown in Figure 2(b). It operates over a data stream whose schema is shown in Figure 2(a). The compiler parses the query and produces an internal representation of the corresponding query plan, as depicted in Figure 2(c). *Glacier* makes arithmetic and Boolean operations very explicit in its internal plans in order to prepare for its compositional compilation scheme.

Arithmetic and Boolean operators each add a new attribute to the stream schema. For example, the result of the string comparison Symbol = `"UBSN"` is written into a new field a. Similarly, the outcome of Volume = `100000` is written into attribute b. Both attributes are combined using a Boolean 'and' operation that produces another new column c. This way, selection σ_c does not need to be a higher-order operator, but only inspects a single Boolean column c to filter out non-matching tuples. At the root of the plan in Figure 2(c), the projection operator π discards the superfluous columns and establishes the requested output schema.

2.2 Circuit Generation

Glacier compiles the algebraic query plan into the description of a hardware circuit expressed in the hardware description language VHDL, which is then fed into a standard FPGA tool chain and loaded into the chip.

Figure 2(d) shows a graphical representation of the hardware plan generated for the algebraic plan in Figure 2(c). In this hardware plan, the tuples from the Trades stream enter the system at the bottom on an 128-bit wide bus. The tuples being processed move upwards the pipeline and leave the circuit at the top. Each operator is implemented in a separate pipeline stage. Tuples are paired with an additional `data_valid` signal that encodes whether the state of an operator describes a valid tuple or not.

Pipeline stages correspond to query operators in the algebraic plan. The two predicates '=' and '<' are evaluated first and the outcome of the comparisons represented as signals a and b, respectively. A logical 'and' $\boxed{\&}$ implements the Boolean expression $a \wedge b$, which is used in pipeline stage four to implement the selection operator σ_c: another logical 'and' $\boxed{\&}$ invalidates the `data_valid` signal whenever the filter condition is not met.

The fifth and last state implements the projection operation $\pi_{\mathsf{Price,Volume}}$. *Glacier* simply leaves signals unconnected if their corresponding attributes are to be discarded. This way, *Glacier* takes advantage of the circuit optimizer in the FPGA design flow. Sub-circuits whose output is never consumed will be pruned automatically, effectively implementing projection pushdown for free.

2.3 Performance Characteristics

The pipeline is driven by a common clock and data advances one stage every clock cycle. Therefore, the processed tuples leave the circuit with a latency of 5 FPGA cycles. The circuit is fully pipelined such that a new tuple can be inserted into the first stage during every clock cycle, which corresponds to an *issue rate* of 1 tuple per cycle.

The issue rate directly translates into throughput. The observable throughput in tuples per second depends on the clock rate the circuit is operated with. On current FPGA hardware, we can operate the circuit shown in Figure 2(d) at 100 MHz, resulting in a throughput of 100 million tuples per second at a latency of 50 ns (five pipeline stages with 10 ns latency each).

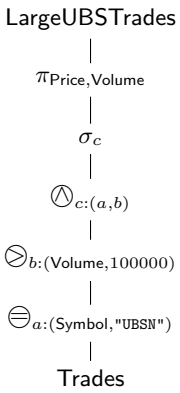
The circuits for aggregation queries and queries containing group-by require additional FPGA mechanisms (such as *content-addressable memories* or *multiplexers*). Currently, our implementation of *Glacier* supports algebraic aggregates

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

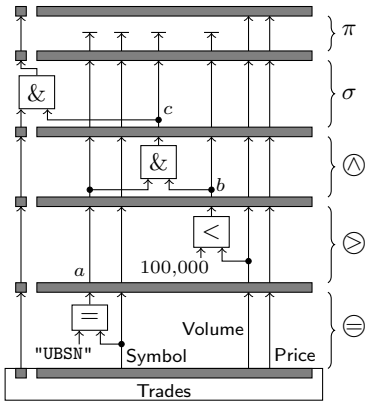
(a) Stream Declaration

```
SELECT Price, Volume
  FROM Trades
 WHERE Symbol = "UBSN" AND
       Volume > 100000
  INTO LargeUBSTrades
```

(b) Textual Query



(c) Algebraic Plan



(d) Hardware Circuit

Figure 2: Translation of a query into a plan and a hardware circuit.

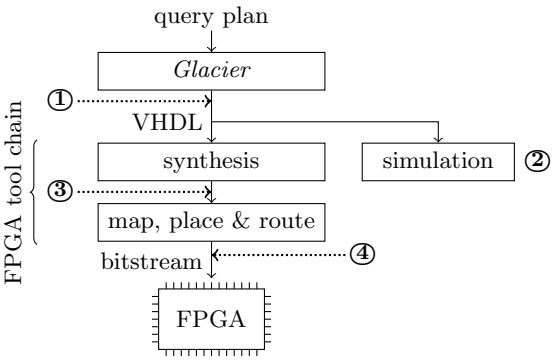


Figure 3: *Glacier* design flow from query to FPGA-based implementation. VHDL compilation done using available FPGA tool chain.

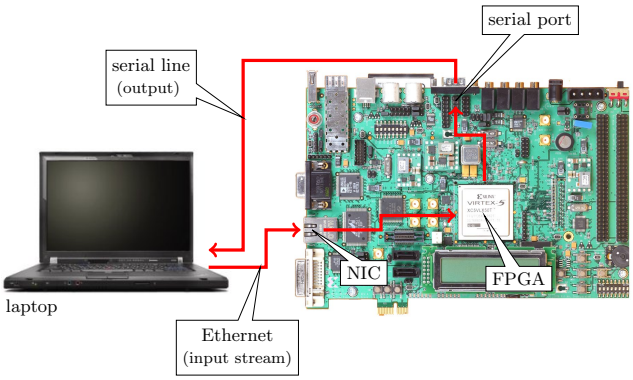


Figure 4: Demonstration setup. A synthetic Trades stream is fed into an FPGA development board, evaluated, and the result visualized on the laptop.

requiring a constant-sized state over time- and tuple-based windows. We refer to [5] for details.

3. DEMO SETUP

Our demonstration will showcase a full *Glacier* design flow as well as a fully functional stream engine that processes (synthetic) stock ticker data off a network.

Design Flow. The *Glacier* compiler is embedded into a *design flow* that uses a standard FPGA tool chain as its back-end, as shown in Figure 3. For the demonstration, we enriched the *Glacier* system with various hooks to illustrate the inner workings of hardware plan generation. The design flow can be intercepted at different processing stages, as indicated with ① through ④ in Figure 3.

Right after VHDL generation with *Glacier* ①, the generated VHDL code can be inspected on the console, but also using a high-level plan diagram (much like the one shown as Figure 2(d)). The VHDL code can also be fed into a *circuit simulator* ② to visualize all intermediate plan results and to verify the correctness of the generated circuit.

Different stages of the vendor-provided FPGA tool chain can be intercepted, too, in order to inspect the *schematic* ③ and *chip* ④ views of the generated hardware circuit. The former illustrates the composition of basic building blocks from the *Glacier* component library; the latter breaks the design down into low-level FPGA primitives and visualizes their placement and routing on the chip.

Stream Engine. We demonstrate a realistic stock ticker scenario with a hardware setup as shown in Figure 4. Synthetic stock ticker data is generated on a laptop machine. The data is sent over an Ethernet connection to the network port of an FPGA development board, where a user-specified query is evaluated fully in hardware. The outcome of the query is sent back to the presentation laptop using a second connection (a serial line), then visualized on the laptop screen.

Conference attendees will be able to experiment with both parts of the demonstration and, e.g., state their own queries and inspect the corresponding hardware circuits.

Acknowledgements

This work was supported by an *Ambizione* grant of the Swiss National Science Foundation under the grant number 126405 and by the Enterprise Computing Center (ECC) of ETH Zurich (`http://www.ecc.ethz.ch/`).

4. REFERENCES

[1] Netezza Corp. `http://www.netezza.com/`.

[2] Kickfire. `http://www.kickfire.com/`.

[3] A. Mitra, M. R. Vieira, P. Bakalov, V. J. Tsotras, and W. A. Najjar. Boosting XML Filtering Through a Scalable FPGA-Based Architecture. In *CIDR*, Asilomar, CA, 2009.

[4] R. Mueller, J. Teubner, and G. Alonso. Data Processing on FPGAs. *PVLDB*, 2(1), 2009.

[5] R. Mueller, J. Teubner, and G. Alonso. Streams on Wires—A Query compiler for FPGAs. *PVLDB*, 2(1), 2009.

[6] T. Oliver, B. Schmidt, and D. Maskell. Hyper Customized Processors for Bio-Sequence Database Scanning on FPGAs. In *FPGA*, Monterey, CA, 2005.

[7] J. Teubner, R. Mueller, and G. Alonso. FPGA Acceleration for the Frequent Item Problem. In *ICDE*, Long Beach, CA, 2010.

[8] Xilinx Inc. *Virtex-6 Family Overview. Data Sheet 150*, September 2009.