Streams on Wires — A Query Compiler for FPGAs

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

Taking advantage of many-core, heterogeneous hardware for data processing tasks is a difficult problem. In this paper, we consider the use of FPGAs for data stream processing as co-processors in many-core architectures. We present *Glacier*, a component library and compositional compiler that transforms continuous queries into logic circuits by composing library components on an operator-level basis. In the paper we consider selection, aggregation, grouping, as well as windowing operators, and discuss their design as modular elements.

We also show how significant performance improvements can be achieved by inserting the FPGA into the system's data path (*e.g.*, between the network interface and the host CPU). Our experiments show that queries on the FPGA can process streams at more than one million tuples per second and that they can do this directly from the network, removing much of the overhead of transferring the data to a conventional CPU.

1. INTRODUCTION

The current trends in hardware architecture towards many-core, heterogeneous machines open up interesting opportunities and difficult challenges for conventional data processing engines. While some of the known problems do not go away and even become more acute (the *memory wall* or the *I/O bottleneck*), new complex problems arise such as how to exploit the parallelism of many-core architectures; how to deal with heterogeneous cores; or the increasing intra-host *communication overhead*. An example of how future machines may look like is the Cell Broadband Engine which provides eight specialized "synergistic processing units" (SPUs) next to a general-purpose CPU. Similarly, hardware vendors have already announced that future cores in a many-core machine will not contain identical CPUs. Some will have floating points units while others will not; some will provide larger instruction sets; and some will just not be CPUs at all.

To address some of the challenging questions raised by these scenarios, at ETH Zurich we have started the *Avalanche* project. In *Avalanche*, we are studying the impact of modern computer architectures on data processing. As part of the efforts around *Avalanche*, we are assessing the potential for using FPGAs as additional cores in many-core machines and how they could be exploited for data processing purposes. In this paper, we report on the work we have done developing *Glacier*, a library of components and a basic compiler for continuous queries implemented on top of an FPGA. The ultimate goal of this line of work is to develop a hybrid data stream processing engine where an optimizer distributes query workloads across a set of CPUs (general-purpose or specialized) and FPGA chips. In here, we focus on how conventional streaming operators can be mapped to circuits on an FPGA; how they can be combined into queries over data streams; the proper system setup for the FPGA to operate in combination with an external CPU; and the actual performance that can be reached with the resulting system.

The paper makes the following contributions:

1. We describe *Glacier*, a component library and compiler for FPGA-based data stream processing. Besides classical streaming operators, *Glacier* includes specialized building blocks needed in the FPGA context. With the operators and specialized building blocks, we show in the paper how *Glacier* can produce FPGA circuits that run streaming queries.

2. Since FPGAs behave very differently than software, we provide an in-depth analysis of the *complexity and performance* of the resulting circuits. We discuss *latency* and *issue rates* as the relevant metrics that need to be considered by an optimizing plan generator.

3. Finally, we evaluate the *end-to-end performance* of an FPGA inserted between the network and the CPU and running a query compiled with *Glacier*. Our results show that the FPGA can process streams at a rate beyond one million tuples per second, far more than the CPU could. These results demonstrate the potential for using FPGAs as co-processors in engines running on many-core architectures.

Our work is organized as follows. The upcoming Section 2 motivates our work with an actual use case, before we give technical background on FPGAs (Section 3). Section 4 describes the *Glacier* compiler, which is complemented by auxiliary circuitry described in Section 5. Sections 6 and 7 describe opportunities for optimization and evaluate our work (respectively). We discuss related work in Section 8 and wrap up in Section 9.

2. STREAMS IN SOFTWARE

Our running example is based on an ongoing collaboration with a Swiss bank. Their financial trading application is subscribed to a subset of 35 event streams published by the Eurex stock exchange[1]. Each stream arrives as a sequence of several ten thousand UDP network packages per second, with package rates varying depending on the current market situation. Latency is a highly critical parameter and measured in units of microseconds (μs).

To abstract from the real application, we assume an input stream that contains a reduced set of information about each trade handled by Eurex (the actual streams are implemented as a compressed representation of the feature-rich FIX protocol [4]). Expressed in the syntax of StreamBase [16], the schema of our data would read:

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

To keep matters simple, we look at queries that process a single data stream only. In particular, we disallow the use of joins. To facilitate the allocation of resources on the FPGA, we restrict ourselves to queries with a predictable space requirement. We do allow aggregation queries and windowing; in fact, we particularly look at such functionality in the second half of Section 4.

Our first set of example queries is designed to illustrate a hardware-based implementation for the most basic operators in stream processing. Queries Q_1 and Q_2 use simple projections as well as selection and compound predicates:

```
SELECT Price, Volume
  FROM Trades
 WHERE Symbol = "UBSN"
  INTO UBSTrades
```

(Q_1)

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

(Q_2)

Financial analytics often depend on statistical information from the data stream. Using sliding-window and grouping functionality, Query Q_3 counts the number of trades of UBS shares over the last 10 minutes (600 seconds) and returns the aggregate every minute. In Query Q_4, we assume the presence of an aggregation function `wsum` that computes the weighted sum over the prices seen in the last four trades of UBS stocks (similar functionality is used, *e.g.*, to implement finite-impulse response filters). Finally, Query Q_5 determines the average trade prices for each stock symbol over the last ten-minutes window:

```
SELECT count () AS Number
  FROM Trades [SIZE 600 ADVANCE 60 TIME]
 WHERE Symbol = "UBSN"
  INTO NumUBSTrades
```

(Q_3)

[1]`http://www.eurexchange.com/`

$\pi_{a_1,\dots,a_n}(q)$	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 b, $agg \in \{$`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
$q_1 \oplus q_2$	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).

```
SELECT wsum(Price, [.5, .25, .125, .125]) AS Wprice
  FROM (SELECT * FROM Trades
         WHERE Symbol = "UBSN")
       [SIZE 4 ADVANCE 1 TUPLES]
  INTO WeightedUBSTrades
```

(Q_4)

```
SELECT Symbol, avg(Price) AS AvgPrice
  FROM Trades [SIZE 600 ADVANCE 60 TIME]
 GROUP BY Symbol
  INTO PriceAverages
```

(Q_5)

We use these five queries in the following to demonstrate various features as well as the compositionality of the *Glacier* compiler.

2.1 Algebraic Plans

Input to our compiler is a query representation in an algebra for streaming queries. Our compiler currently supports the algebra dialect listed in Table 1, whereby operators may be composed in an arbitrary fashion.

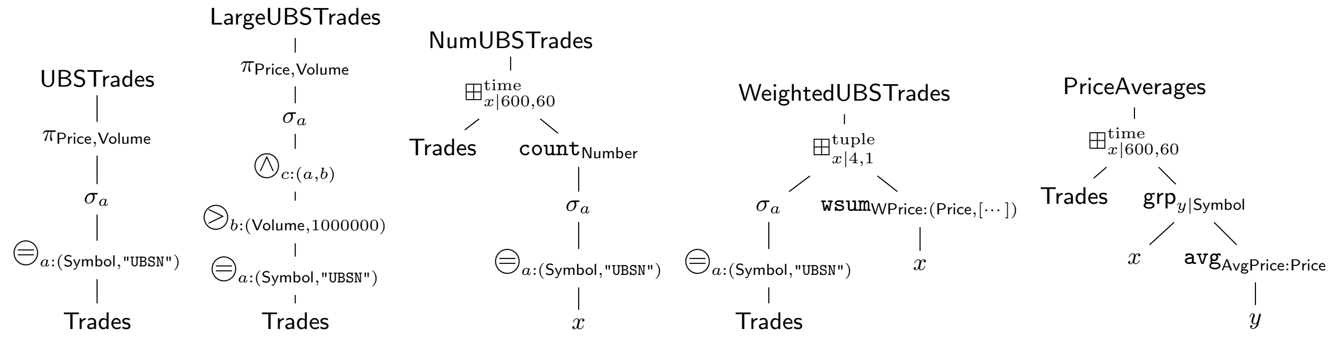
Our algebra uses an "assembly-style" representation that breaks down selection, arithmetics, and predicate evaluation into separate algebra operators. In earlier work, we found a similar representation also suited to express, *e.g.*, the semantics of XQuery [11]. In the context of the current work, the notation turns out to have nice correspondences to the data flow in a hardware circuit and helps detecting opportunities for parallel evaluation.

Figure 1 illustrates how our streaming algebra can be used to express the semantics of Queries Q_1 through Q_5. Observe how in Figure 1(a), *e.g.*, operator $\circeq$ makes the comparison of each input tuple with the requested stock symbol "UBSN" explicit. Its output, the new column a, is used afterwards to filter out non-qualifying tuples (operator σ_a).

The *concatenate operator* $\oplus$ represents what could be called a "join by position". Tuples from both input streams are combined into a wide result tuple in the order in which they arrive. The operator is necessary, *e.g.*, to evaluate and return different aggregation functions over the same input stream.

3. FPGAS FOR STREAM PROCESSING

At its very heart, every FPGA chip consists of three principal types of components. A large number of *lookup tables (LUTs)* provides a programmable type of logic gates.


UBSTrades
$\pi_{\text{Price,Volume}}$
σ_a
$\ominus_{a:(\text{Symbol},\text{"UBSN"})}$
Trades
LargeUBSTrades
$\pi_{\text{Price,Volume}}$
σ_a
$\oslash_{c:(a,b)}$
$\oslash_{b:(\text{Volume},1000000)}$
$\ominus_{a:(\text{Symbol},\text{"UBSN"})}$
Trades
NumUBSTrades
$\boxplus^{\text{time}}_{x|600,60}$
Trades
$\text{count}_{\text{Number}}$
σ_a
$\ominus_{a:(\text{Symbol},\text{"UBSN"})}$
x
WeightedUBSTrades
$\boxplus^{\text{tuple}}_{x|4,1}$
σ_a
$\text{wsum}_{\text{WPrice}:(\text{Price},[\cdots])}$
$\ominus_{a:(\text{Symbol},\text{"UBSN"})}$
x
Trades
PriceAverages
$\boxplus^{\text{time}}_{x|600,60}$
Trades
$\text{grp}_{y|\text{Symbol}}$
x
$\text{avg}_{\text{AvgPrice:Price}}$
y

(a) Query Q_1 (b) Query Q_2 (c) Query Q_3 (d) Query Q_4 (e) Query Q_5

Figure 1: Algebraic query plans for the five example queries Q_1 to Q_5.

6-to-1 lookup tables	69,120
flip-flops (1-bit registers)	69,120
block RAM	296×18 kbit
25×18-bit multipliers	64
typical clock rate	100 MHz

Table 2: Xilinx XC5VLX110T characteristics.

Each lookup table can implement an arbitrary 6 bit $\mapsto$ 1 bit function. Lookup tables are wired through an *interconnect network* that can route signals across the chip. Finally, *flip-flops* (also called registers) provide 1-bit storage units that can directly be wired into the remaining logic.

The behavior of lookup tables, the wiring of the interconnect, and the initial state of the flip-flops can all be configured by software. The actual configuration is typically described using a *hardware description language* (such as VHDL or Verilog) that is then loaded into the FPGA.

Most FPGA chips have additional functionality available as hard-wired silicon components. Examples of this include low-latency on-chip memory (*block RAM* or *BRAM*), hardware multipliers, floating-point units, or even full-fledged CPU cores. The hardware we use to evaluate our work, *e.g.*, includes 666 kByte block RAM. Table 2 shows the characteristics of the FPGA we use in this paper. Configurable *I/O pins* let the FPGA chip communicate with peripheral hardware, such as external RAM, network, or storage bus interfaces.

3.1 Content-Addressable Memory

The main advantage of using FPGAs for data processing is their intrinsic parallelism. Among others, this enables us to escape from the *von Neumann bottleneck* (also called the *memory wall*) that classical computing architectures struggle with. In the common von Neumann model, memory is physically separated from the processing CPU. Data is acquired from memory by sending the location of a piece of data, its *address*, to the RAM chip, then receiving the data back. In FPGAs, flip-flop registers and block RAM are distributed over the chip and tightly wired to the programmable logic. In addition, lookup tables can be reprogrammed at runtime and thus be used as additional distributed memory. As such, the on-chip storage resources of the FPGA can be accessed in a truly parallel fashion.

A particular use of this potential is the implementation of *content-addressable memory* (CAM). Other than traditional memory, content-addressable memory can be accessed by data values, rather than by explicit memory addresses. Typically, CAMs are used to resolve a given data item to the address it has been stored at. More generally, however, the functionality can implement an arbitrary key-value store with constant (typically single-cycle) lookup time.

We refer to the work of Guccione *et al.* [12] or documentation provided by Xilinx [17] for details on FPGA-based CAM implementations. In Section 4.7, we use content-addressable memory to implement lookups during 'group by' execution. The access pattern in this context, frequent lookups with rare updates, suggests the use of a CAM implementation that is based on lookup tables. It excels with very high lookup speeds (fraction of a clock cycle), but has a 16-cycle latency for updates. As an alternative, a block RAM-based implementation would require a full cycle for lookups and two cycles for updates.

3.2 System Setup

FPGAs can mimic arbitrary logic functionality by mere reconfiguration. In contrast to existing special-purpose hardware (such as graphics or floating-point processors), this makes the role of an FPGA inside the overall system not predetermined. By implementing the respective bus protocols, *e.g.*, FPGA can be connected to memory or peripheral buses, communicate with external devices, or any combination thereof.

Two architectures seem particularly relevant to our problem domain at this point, both are illustrated in Figure 2. In the top part of this figure (method (a)), the FPGA is directly connected to the physical network interface, with parts of the network controller implemented inside the FPGA fabric. After reception, data from the network is directly fed into the hardware implementation of a database query plan. The host CPU only becomes involved once result items have been produced for the user query. Using DMA, glue logic of the *Glacier* library writes the result tuples into the system main memory, then informs the host CPU about the arrival of new data (*e.g.*, by raising an interrupt).

Alternatively, the FPGA can also be used in a traditional co-processor setup, as illustrated in Figure 2 (b). Here, the CPU hands over data to the FPGA either by writing directly into FPGA registers (so-called *slave registers*) or it prepares

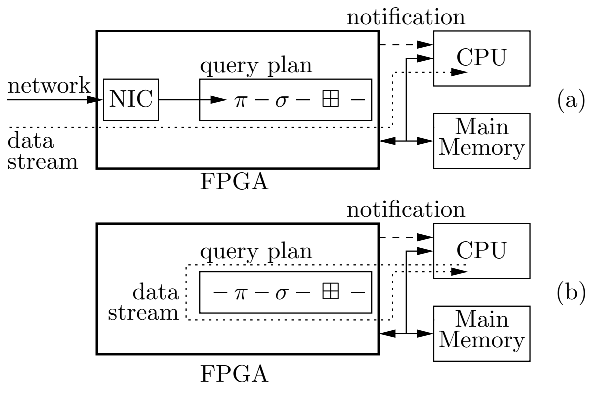


Figure 2: System Architectures: (a) Stream engine between network interface and CPU. (b) Stream engine as a coprocessor to the CPU.

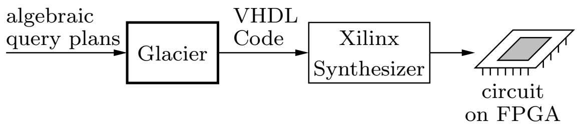


Figure 3: Compilation of abstract query plans into hardware circuits for the FPGA.

the input data into a shared RAM region, then sends a *work request* to the FPGA-based co-processor.

The architecture in Figure 2 (a) fits a pattern that is highly common in data stream applications. Oftentimes, rate-reducing filtering or aggregation stages precede more complex high-level processing (done on the CPU). In algorithmic trading, *e.g.*, early filters discard about 90 % of all input data. After filtering, only the remaining 10 % of the data actually hit the CPU, significantly increasing the applied load that the system can sustain.

3.3 Query Compilation

Figure 3 illustrates the compilation process from algebraic plans to FPGA circuits. As input to our *Glacier compiler*, we are given algebraic query plans as introduced in Section 2.1. The compiler applies the set of compilation rules (Section 4) and optimization heuristics (Section 6), then emits the description of a logic circuit that implements the input plan. The generated circuits are expressed in VHDL hardware description language. The VHDL code is fed to the Xilinx compiler tool which creates the actual low-level, FPGA-specific representation of the circuit (configuration of the LUTs and the interconnect network). The output of the Xilinx compiler tool is then used to program the FPGA. In Figure 3, the compilation of VHDL code into an FPGA configuration follows the usual design flow in traditional FPGAs design. Using *Glacier*, the creation of VHDL code can be fully automated.

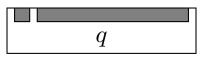
4. FROM QUERIES TO CIRCUITS

Using pre-built components from the *Glacier* library, each operator in Table 1 can be compiled into a hardware circuit in a systematic way. To ensure the full compositionality of the translation rules later in this section, every translated sub-plan adheres to the same well-defined wiring interface.

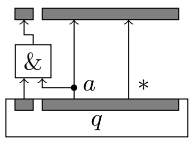
4.1 Wiring Interface

As in data streaming engines, our processing model is entirely push-based. Each n-bit-wide tuple is represented as a set of n parallel wires in the FPGA fabric. On a set of wires, a new tuple can be propagated in every cycle of the FPGA's system clock (*i.e.*, 100 million tuples per second). An additional *data valid* line signals the presence of an actual tuple in a given clock cycle. Tuples are only considered to be part of the data stream if their *data valid* flag is set to `true`, *i.e.*, if the *data valid* line carries a logical "high" signal.

In the following, we use rectangles to represent logic components (with the exception of multiplexers, for which we use the common trapezoid notation). Our circuits are all clock-driven or *synchronized* and every operator in our library writes its output into a flip-flop register after processing. We indicate registers as gray-shaded boxes and make the *data valid* flag explicit as each operator's leftmost output. For instance, we depict the black-box view of a hardware implementation for a query q as shown on the right.



We use arrows to denote the wiring between hardware components. Wherever appropriate, we identify those lines from a tuple bus that correspond to a specific tuple field with a label at the respective input/output port. The label '$*$' stands for "all remaining fields". We are not representing the ordering of fields within a tuple. The corresponding hardware plan for the algebra expression $\sigma_a(q)$, *e.g.*, can thus be illustrated as



.

In this circuit, the logical 'and' gate invalidates the output tuple whenever field a contains `false`.

4.1.1 Circuit Characteristics

The above circuit will compute its output in a single clock cycle and will be ready to consume a new input tuple at every tick of the clock. We say that its *latency* and *issue rate* are both 1. In general, circuits may need more than one cycle until the result of their computation can be picked up at the operator output—they have a latency that is larger than 1. Due to their semantics, circuits that implement grouping or windowing cannot produce output before they have seen the last tuple of the respective query window. For these operators, we define latency to be the number of clock cycles between the closing of the input window and the generation of the first output tuple.

We define the issue rate as the number of tuples that can be processed per clock cycle. The issue rate is always ≤ 1. For example, an operator can accept a tuple every five cycle its issue rate is 0.2.

Some operations can be *pipelined*. The corresponding circuits will be ready to consume new input already *before* the output of the preceding tuple has been fully computed. They have an issue rate with a higher than the recprocal value of their latency. Some readers may be familiar with this terminology from the systems architecture domain [14].

Latency and issue rate are important parameters to determine the performance of a hardware circuit. Latency directly corresponds to the observable response time, whereas the issue rate determines throughput.

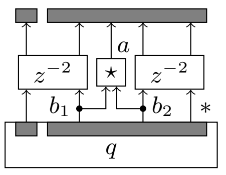
4.1.2 Synchronization

Both properties sometimes also need to be considered during query compilation. For instance, all compilation rules must ensure that a generated circuit will never try to push two tuples in successive cycles into an operator that has an issue rate less than one. We use two types of logic components to implement the synchronization between sub-circuits:

FIFO queues act as short-term buffers for streams with a varying data rate. They emit data at a predictable rate, typically the issue rate of an upstream sub-circuit. Note that, at runtime, the average input rate must not exceed what is achievable with the output rate.

In most practical cases, the depth of the FIFO can be kept very low. This not only implies a small resource footprint, but also means that the impact on the overall latency is typically small.

Delay operators z^{-n} can block data items for a fixed amount of cycles n. This can be used, *e.g.*, to properly synchronize the output of slow arithmetic operators with the remaining tuple flow (circuit implements $\textcircled{\star}_{a:(b_1,b_2)}(q)$; assume that the latency of $\star$ is 2):

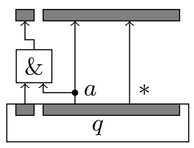


.

4.2 Selection and Projection

We saw earlier how our assembly-style selection operator σ_a can be cast into a hardware circuit. Compilation Rule Sel formalizes this translation in the notation we also use in the remainder of this work. We use the $\Mapsto$ symbol to indicate the "compiles to" relation and, as before, assume that a rectangle labeled q is the circuit that results from compiling q:

$\sigma_a(q) \quad \Mapsto$

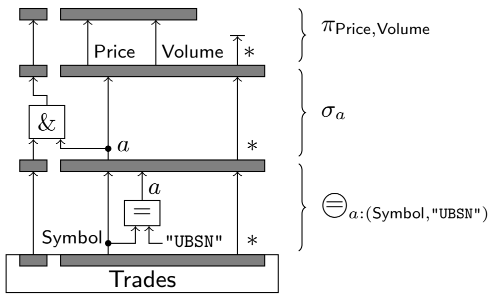


. (Sel)

Note that the resulting circuit leaves all tuples essentially intact, but invalidates discarded tuples by setting their *data valid* flag to false. This is very similar in nature to the "selection vectors" that MonetDB/X100 [13] uses to avoid data copying.

The logical 'and' gate $\boxed{\&}$ completes within a single cycle, *i.e.*, latency and issue rate of the circuit generated for σ_a are both 1.

Here, we use the projection operator $\pi_{a_1,\dots,a_n}$ to discard fields from the tuple flow. Support for field re-naming (often expressed using the π operator) is a straightforward extension of what we present here.

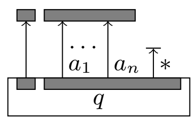


Price
Volume
*
$\pi_{\mathsf{Price,Volume}}$
&
a
*
σ_a
a
=
Symbol
"UBSN"
*
$\textcircled{=}_{a:(\mathsf{Symbol},\texttt{"UBSN"})}$
Trades

Figure 4: Compiled hardware execution plan for Query Q_1. Latency of this circuit is 3, issue rate 1.

Discarding a field from the tuple flow simply means to not wire the respective output ports with any inputs further down the data path, as shown in Rule Proj:

$\pi_{a_1,\dots,a_n}(q) \quad \Mapsto$



. (Proj)

This implementation for $\pi_{a_1,\dots,a_n}$ has an interesting side effect. Our compiler emits the *description* of a hardware circuit that is passed into the *synthesizer* tool to generate the actual hardware configuration for the FPGA. The synthesizer optimizes out "dangling wires", effectively implementing *projection pushdown* for free.

There is no actual work to do at runtime (though fields are propagated into a new set of registers). Latency and issue rate of this implementation for projection are both 1.

4.3 Arithmetics and Boolean Operations

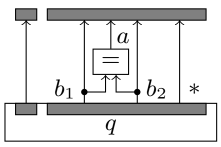
As indicated in Table 1, we use the generic $\textcircled{\star}_{a:(b_1,b_2)}$ operator to represent arithmetic computations, value comparisons or Boolean connectives in relational plans. The instance

$$\textcircled{=}_{a:(b_1,b_2)}(q) \ ,$$

e.g., will emit all fields in q, extended by a new field a that contains the outcome of $b_1 = b_2$.

This semantics directly translates into an implementation in logic (we saw a similar circuit a moment ago):

$\textcircled{=}_{a:(b_1,b_2)}(q) \quad \Mapsto$



. (Equals)

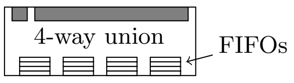
Most simple arithmetic or Boolean operations will run within a single clock cycle. More complex tasks, *e.g.*, multiplication/division, or floating-point arithmetics, may require additional latency. Sometimes, the actual circuit that implements $\boxed{\star}$ can be tuned within the trade-offs latency, issue rate, and chip space consumption. If the latency of $\boxed{\star}$ is greater than one, delay operators have to be introduced to synchronize the operator output with the remaining fields (as shown before in Section 4.1.2).

Example. With the rules we saw so far, we can now translate our first example query into a hardware circuit. In Figure 4, we illustrated the circuit that results from applying our compilation rules to Query Q_1.

The hardware circuit quite literally reflects the shape of the algebraic plan. Each of the operators can individually operate in a single circuit (*i.e.*, have latency and issue rates of one). Since all plan operators are applied sequentially, latencies add up and the circuit in Figure 4 has an overall latency of three. By contrast, the issue rate of a pipelined execution plan is determined by its slowest sub-plan. Since all sub-plans have an issue rate of one, this is also the rate of the complete plan. □

4.4 Union

From a data flow point of view, the task of an algebraic union operator ∪ is to accumulate the output of several source streams into a single output stream. Since, in our case, all source streams operate truly in parallel, a hardware implementation for ∪ needs to ensure proper synchronization. We do so by buffering all input ports using FIFOs:

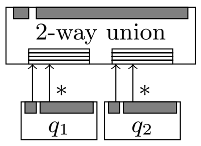


A state machine inside the union component then forwards tuples from the input FIFOs in a round-robin fashion and emits them as the union result.

Though every individual input may feed into the union component at an arbitrary tuple rate (*i.e.*, issue rate 1), the average rate of all input streams together must not exceed more than one tuple per cycle, which is the maximum tuple rate that the union component can forward up-stream the data path. In terms of latency, the state machine inside the operator requires a single cycle to process. The FIFOs at the input, implemented using either flip-flop registers or block RAM (a resource trade-off), add another latency cycle. The overall circuit therefore has a minimum latency of 2. Depending on the input data distribution, however, the observed latency may be higher whenever tuples queue up in an input FIFO.

Strictly speaking, a binary union component is sufficient to implement the algebraic ∪ operator:

$q_1 \cup q_2 \quad \mapsto$



. (Union)

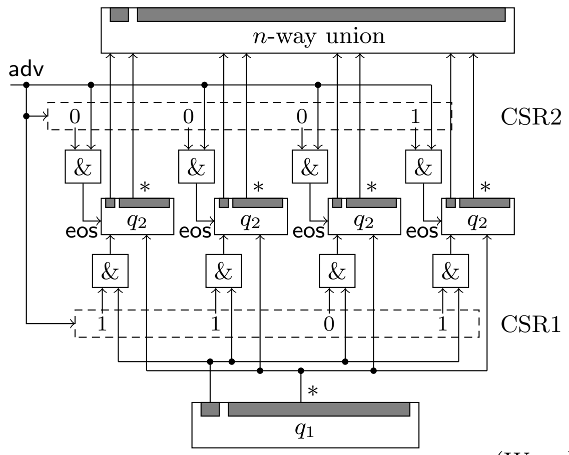
As we will see in the following, however, the availability of a general, n-way union implementation eases the implementation of other functionality.

4.5 Windowing

The concept of windowing bridges the gap between stream processing and relational-style semantics. The operation $q_1 \boxplus_{x|k,l} q_2$ consumes the output of its left-hand sub-plan (q_1) and slices it into a set of *windows.* For each window, $\boxplus_{x|k,l}$ invokes a parameterized execution of the right-hand sub-plan $q_2(x)$, with each occurrence of x replaced by the current window. Sub-plan q_2 thus sees a finite input for every execution and may, *e.g.*, use aggregation in a semantically sound manner.

Our compiler implements this semantics by wrapping q_2 into a template circuit (full compilation rule shown in Figure 5). We introduce an additional input signal eos next to the *data valid.* This is asserted "high" when the window

$q_1 \boxplus_{x|k,l} q_2(x) \mapsto$



(Wind)

Figure 5: Compilation rule for windowing operator ⊞ (shown for an instance with at most three windows open in parallel).

closes and notifies the sub-plan that it has seen all elements of that window. The signal typically triggers the sub-plan to start generating output tuples.

A common use case are *sliding windows*, where input tuples belong to several windows at the same time. Here we can exploit the available *parallelism* on the FPGA chip. We replicate the hardware plan of q_2 as many times n as there may be windows open in parallel during query execution, plus 1. For time- and tuple-based windows, *e.g.*, we have that $n = \lceil k/l \rceil + 1$ (where k is the window size and l is the slide). In Figure 5, we assume $n = 4$ (*i.e.*, at most three windows open in parallel). To keep matters simple, we assume that k is a multiple of l; the extension to the general case is straightforward.

We use the *cyclic shift register* CSR1 (indicated as a dashed box in Figure 5) to keep track of window states. For every instance of the sub-plan q_2, this shift register carries the information whether the instance actively processes an open window. Figure 5 assumes that three windows are open in parallel, *i.e.*, three bits are set in CSR1. Whenever the end of a window is reached, triggering the "advance" signal adv rotates the shift register (to the right), such that the oldest open window is closed and a new one opened. The signal adv may be driven either by a clock (for time-based windows) or by a counter that implements tuple-based windows.

Parallel to advancing CSR1, we send an eos signal to the sub-plan that processes the oldest open window. This sub-plan will then start producing output and feed it to the up-stream plan through a union operator. While doing so, the sub-plan will have the 0-bit in CSR1, *i.e.*, it will not receive any new input while emitting tuples. To communicate the eos signal to the correct sub-plan, we use a second shift register CSR2, shifted in sync with CSR1. The single bit in CSR2 identifies the oldest open window.

Example. The hardware circuit that implements the sliding-window query Q_4 is shown in Figure 6. With the windowing clause `[SIZE 4 ADVANCE 1 TUPLES]`, at most four windows can be open together at any point in time. Hence, we in-

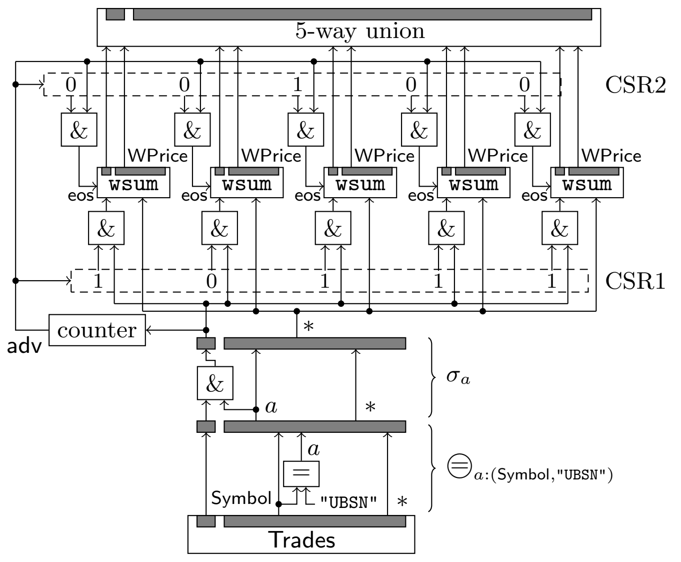


Figure 6: Hardware execution plan for Query Q_4.

stantiate five copies of the `wsum` sub-plan. The window type of this query is tuple-based. The 'counter' component on the left counts incoming tuples and sends the `adv` signal as often as specified by the query's `ADVANCE` clause (in this particular case, `ADVANCE` = 1 and we could simplify our circuit by directly routing *data valid* to the `adv` line). □

Signal processing in the windowing part of the plan is implemented fully asynchronously. It fits into a single clock cycle and is fully pipelineable. The latency of the overall circuit thus is the latency of the inner plan plus 2 (the latency of the n-way union operator). The issue rate is the one of the inner circuit.

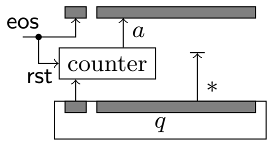
4.6 Aggregation

Other than the previous operators, aggregation functions (`count`, `min`, `max`, `avg`, ...) assume a *finite* input data set. Typically, they are applied on windows. As seen in the previous section, windowing breaks a potentially infinite stream into finite sub-streams.

In practice (and as implemented in the previous section), tuples are streamed into a set of open windows immediately after arrival, rather than batching them up until a window closes. We use `eos` ("end-of-stream") signal to notify the aggregation circuit when a window closes or the end of the current input stream has been reached (for example after a finite input from a persistent database table has be consumed). Note that the window operator itself does not provide storage for data elements. The tuples are directly forwarded and therefore storage needs to be provided by implementation of the aggregation function instead. This has the advantage that each aggregation function needs to provide storage just for the amount of state it requires, rather than maintaining the entire window. As introduced by [9], aggregation functions can be classified as follows:

Algebraic Aggregate Functions. We implement algebraic aggregate functions (*i.e.*, ones that use a fixed amount of state) [9] in a straightforward fashion. To implement `count`, *e.g.*, we use a standard counter component and wire its trigger input to the *data valid* signal of the input stream. Once we reach the end of the current stream, we *(a)* emit the counter value to the upstream data path and *(b)* reset the counter to zero to prepare for the next input stream.

In the translation rule for $\texttt{count}_a(q)$,

$\texttt{count}_a(q) \quad \Mapsto$  , (Count)

we forward the `eos` signal to the *data valid* output register to implement *(a)* and feed the same signal into the reset input of the counter to implement *(b)*. Note that $\texttt{count}_a$ constructs a new output field without reading any particular input value. The operator emits no other field but the aggregate (we handle grouping separately, see next). For the algebraic aggregates we consider `cout`, `sum`, `avg`, `min`, and `max` the latency is one cycle. A tuple can be applied at the input every clock cycle (issue rate=1).

Holistic Aggregate Functions. For some aggregate functions, the state required is not within constant bounds. They need to batch (parts of) their input until the the aggregate can be computed when the end of the stream is seen. The prototype example for such operators are the computation of medians or most frequent items. Our weighted sum operator `wsum` behaves similarly, but needs to remember only the last four input tuples. The use of *flip-flops* is a good choice to hold such small quantities of data. Here we can use them in a *shift register* mode, such that the operator buffer always contains the last four input values.

4.7 Grouping

Semantically, a grouping expression $q_1 \; \mathsf{grp}_{x|c} \; q_2$ evaluates the left-hand sub-plan q_1, then routes each tuple to one of a number of independent evaluations of the sub-plan $q_2(x)$. The grouping column c thereby determines the target sub-plan for every input tuple.

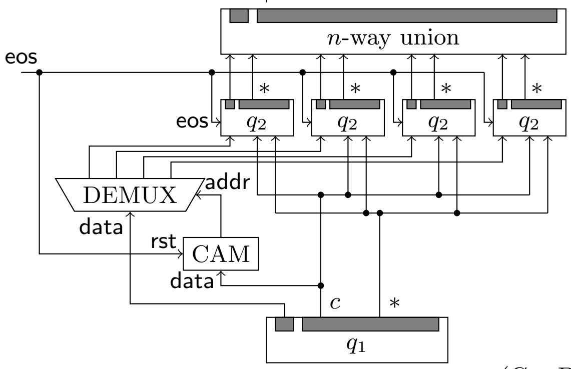
FPGA circuits provide excellent support for such functionality. In Section 3.1, we discussed *content-addressable memory* as an efficient mechanism to implement key-value stores. Here, we use that functionality to identify the matching group for an input tuple. Our CAM returns the index i of the sub-plan that matches the given input tuple. We feed this index into the address port of a *de-multiplexer*, which will then route the signal on the `data` input to the ith output line.

Once again, the *data valid* flag comes in handy here. Rather than routing the entire tuple to the proper sub-plan instance, we use the de-multiplexer only to control the *data valid* flag. The actual payload is sent to all sub-plan instances in parallel.

Following our earlier assumptions, we preallocate a number of sub-plan instances, depending on the number of groups that are going to result at runtime. Typically, the sub-plan is a simple aggregate operation with low complexity. Overestimating the number of groups at compile time thus rarely causes a noticeable effect on the overall chip space consumption.

Grouping is typically used in combination with aggregation. Though grouping itself does not chop an infinite stream into finite subsets, we explicitly indicated the necessary routing of `eos` signals to the sub-plan instances. In addition, we use the signal to clear the content-addressable

$q_1 \; \mathsf{grp}_{x|c} \; q_2(x) \mapsto$



(GrpBy)

Figure 7: Compilation rule to implement the 'group by' operator grp.

memory after each group (rst input).

Our CAM implementation is based on lookup tables with very fast lookup performance. De-multiplexing can be processed fully asynchronously, such that the entire routing circuit can typically be processed within a single clock cycle or two (high-capacity CAMs and high-fanout de-multiplexers may be more complex and require an additional wait cycle). As discussed earlier, LUT-based CAMs have a slow write performance, which we have to pay whenever a group item is seen the first time. Since this makes the issue rate of the circuit data-dependent, we use a FIFO (not shown in the circuit) to buffer all input. The circuit thus has a variable latency. A hit or a miss can be determined with a latency of one cycle. If no entry is found in the CAM, additional 16 wait cycles are necessary to insert a new entry. Thus, the overall performance of a CAM is one cycle on a hit and 17 cycles for a miss. The latency of at the output side is given by the latency of the sub-plan plus one (for the n-way union). The average issue rate is one if we assume that the FIFO is large enough (i.e., at least 16 times the number of groups) to buffer the incoming tuples during the wait cycles when writing the CAM.

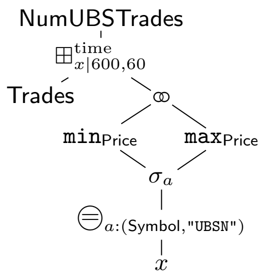
Example. Compiling Query Q_5 would yield a circuit like the one in Figure 7, wrapped into a windowing circuit (as in Figure 5). We omit the plan here because of its obvious complexity. In the actual application, the Trades stream contains market data of a subset of the stock indexes. With less than a hundred different stock symbols per stream, we can easily replicate the `avg` sub-circuit as demanded by Compilation Rule GrpBy. □

4.8 Concatenation Operator

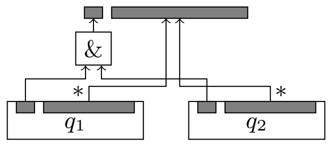
The tuple concatenation operator $\circ\!\!\circ$ is a device mainly intended to express multiple aggregates within the same `SELECT` clause. The query

```
SELECT min(Price),max(Price)
  FROM Trades [SIZE 600 ADVANCE 60 TIME]
 WHERE Symbol = "UBSN"                        ,
```

for instance, could be expressed using the query plan shown on the right. On the hardware side, the semantics of $q_1 \circ\!\!\circ q_2$ is straightforward to implement. We simply direct the signals from all input fields to a common output register set. A tuple generated this way only is meaningful if both input tuples were valid. Hence, we use a logical 'and' gate to combine them:



$q_1 \circ\!\!\circ q_2 \mapsto$



. (Concat)

Again, the 'and' gate easily finishes within a single cycle. Hence, latency and issue rate are both 1.

5. AUXILIARY COMPONENTS

While the previous section provided a compositional scheme to translate a query body into a hardware circuit, actually running the circuit requires some glue logic that lets the execution plan communicate with its environment. *Glacier* includes such logic for commonly used setups.

5.1 Network Adapter

In a *commodity* computing system, the communication between a network interface card (NIC) and its host CPU is performed using a multi-step protocol. In a nutshell, the network card transfers a received packet into the main memory of the host system using DMA, then informs the CPU about the arrival by raising an interrupt. The interrupt lets the operating system switch into kernel mode, where it does all necessary packet decoding, before it hands the data off into user space where the payload can finally be processed.

For latency-critical applications (such as algorithmic trading) or ones with high data volumes, such a long processing stack may be prohibitive. Therefore, we decided to implement our own *network adapter* on the FPGA as a soft-core. The soft-core directly connects to the Ethernet MAC component of the physical network interface. From there, we grab raw Ethernet network frames immediately when they arrive. We implemented a small UDP/IP stack in the soft-core. This allows us to to receive UDP datagrams without the help of the CPU. From the decoded UDP datagrams we can extract the data tuples and feed it to the circuit that represents the compiled execution plans. The host CPU only gets involved for the data that remains after the end of the query pipeline, where it is typically faced with a significantly reduced data load due to filtering and aggregation. In Section 7, we will see how this enables us to process data at gigabit Ethernet wire speed.

Likewise, we could use the same functionality to build a *data sink* that transmits result data over the network without any involvement of the host CPU.

5.2 CPU Adapter

Our system setup in Section 3.2 assumes the host CPU as the other end of the processing pipeline. To send (result) data to the CPU, we use a strategy that is similar to the one used by network cards, as sketched above. We write all data into a FIFO that is accessible by the host CPU via a memory-mapped register. Whenever we have prepared new data, we raise an interrupt to inform the CPU. Code in the host's *interrupt service routine* then reads out the FIFO and hands the data over to the user program.

To different approaches are conceivable to implement a communication into the other direction, *i.e.*, from the CPU to the FPGA. Memory-mapped *slave registers* allow the CPU to push data directly into an FPGA circuit by writing the information into a special virtual memory location. While this provides intuitive and low-latency access to the FPGA engine, the necessary synchronization protocols incur sufficient overhead to fall behind a *DMA-based* implementation if data volumes become high. In this case, the data is written into (external) memory, where logic on the FPGA picks it up autonomously after it has received a *work request* from the host CPU.

5.3 Stream De-Multiplexing

Actual implementations may depend on specialized functionality that would be inefficient to express using standard algebra components. In our actual use case, algorithmic trading, input data is received as a *multiplexed* stream, encoded in a compressed variant of the FIX protocol [4]. Expressed using the StreamBase syntax, the multiplex stream contains actual streams like

```
CREATE INPUT STREAM NewOrderStream (
  MsgType      byte, -- 68: new order
  ClOrdId      int,  -- unique order identifier
  OrdType      char, -- 1:market,2:limit,3:stop
  Side         char, -- 1:buy,2:sell,3:buy minus
  TransactTime long) -- UTC Timestamp

CREATE INPUT STREAM OrderCancelRequestStream (
  MsgType      byte, -- 70: order cancel request
  ClOrdId      int,  -- unique order identifier
  OrigClOrdId  int,  -- previous order
  Side         char, -- 1:buy,2:sell,3:buy minus
  TransactTime long) -- UTC Timestamp
```

We have implemented a *stream de-multiplexer* that interprets the `MsgType` field (first field in every stream) and dispatches the tuple to the proper plan part.

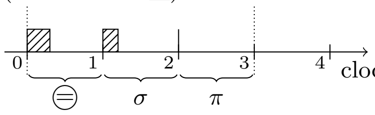
6. IMPROVING PLAN QUALITY

In Section 4 we focused on providing a complete and fully compositional set of compilation rules. With these rules arbitrary stream queries can be compiled into a logic circuit. It is not surprising that "hand crafting" a specific plan sometimes may lead to plans with lower latency and/or better issue rate. It turns out that rather simple optimization heuristics already suffice to make the output of our compiler close to hand-optimized plans.

6.1 Reducing Clock Synchronization

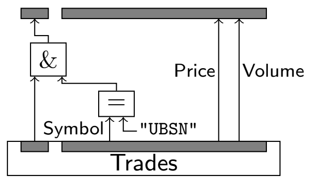
Our compilation rules assume strict *synchronization* of every operator implementation. Every operator is expected to have its result ready after an integer number of clock cycles (the operator's latency). Even though simple computations could finish in less time than a full cycle, their result is always buffered in a flip-flop register, where it waits until the end of the clock cycle.

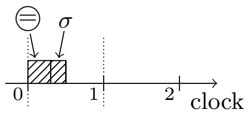
Example. Consider again the compiled circuit for Query Q_1 (Figure 4). As discussed earlier, this circuit requires three clock cycles to execute. Little of that time is used for actual processing, however. In the following timing diagram, we illustrate when each of the three plan parts perform actual processing (indicated as ▨):



Equality comparison takes slightly longer to evaluate than the logical 'and' (which is what σ essentially does). There is no actual work to be done for projection at all, still all three plan parts occupy a full clock cycle each. □

If no components inside a plan step are inherently clock bound (such as access to clocked memory components), a plan optimizer can trivially eliminate intermediate registers and run (part of) a sub-plan *asynchronously.* Applying this idea to the plan for Query Q_1 results in the plan we use in our actual implementation:



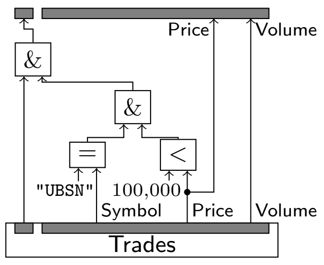


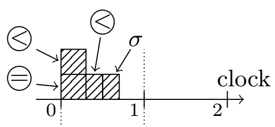
As shown in the timing diagram on the right, this sub-plan now runs both processing steps directly in succession and finishes within a single clock cycle.

The most apparent effect of this optimization is the reduction of latency. The plan for Query Q_1 has now a latency of one. In addition, we saved a small amount of FPGA resources, primarily flip-flops that were needed for buffering before.

6.2 Increasing Parallelism

Eliminating intermediate registers often automatically leads to *task parallelism.* With registers removed, the hardware circuit for Query Q_2 looks as follows:





In this circuit, the two value comparisons run truly in parallel (whereas they would execute sequentially in the non-optimized plan). In the timing diagram on the right, we see how we packed additional work into the same clock cycle. In effect, Query Q_2 executes in a single cycle, too.

6.3 Trading Unions For Multiplexers

When translating the windowing operator ⊞ (Rule Wind), we used the 'union' circuit of Section 4.4 as a convenient tool to merge all window outputs into a single result stream. Except in exotic cases, only one of these outputs is actually producing data at any point in time, and we know which one.

We can take advantage of this knowledge by replacing the union circuit with a *multiplexer* component in such cases. As the name suggests, a multiplexer is the counterpart to the de-multiplexer we saw in Section 4.7. Provided an index i, it routes the signal at the ith input to its output port. In windowing circuits, we know the index of the data-producing sub-plan from the shift register CSR2. Using a multiplexer, we can now feed the output of this sub-plan directly into the output register of the windowing circuit.

As discussed in Section 4.4, the hardware circuit for union uses FIFO queues at each of its n inputs. By using a multiplexer instead, we can free the resources occupied by the n FIFOs. Depending on the FIFO implementation chosen, this may free mostly flip-flop registers or block RAMs, plus the necessary logic (LUTs) that drives the FIFOs. In addition, we save one clock cycle of latency that was originally spent in the input FIFOs. Applied to the plan in Figure 6, this reduces the latency from 5 to 4 (eliminating registers on the bottom half of the plan saves another two cycles of latency).

6.4 Group By/Windowing Unnesting

The $\boxplus$-grp pattern shown in the algebraic query plan for Query Q_5 (see Figure 1(e)) is a common combination in stream processing. A straightforward application of the Wind and GrpBy rules to this pattern will replicate the 'group by' circuit for each of the n sub-plan instances in Rule Wind. The resulting query execution plan will thus use n de-multiplexers and content-addressable memories and route tuples independently for each group.

Typically, all windows will contain roughly the same groups, and all CAM instances will contain roughly the same data items. It therefore makes sense to "pull out" the individual DEMUX/CAM pairs of the replicated sub-plans and use a global instance of each instead. In a sense, we swap the roles of $\boxplus$ and grp in the algebraic plan.

The primary effect of "unnesting" the tuple dispatching functionality of the 'group by' operation is a considerable resource saving. The penalty we pay is a slight increase in the number of groups, since the union of all groups in individual windows is now held in a single CAM.

7. EVALUATION

Compiling stream queries into logic circuits is only meaningful if the resulting circuits solve the problems that we motivated in Section 2. This evaluation section thus focuses on the relevant performance metrics *latency* and *throughput* (our subject for Section 7.1). In Section 7.2, we verify that the integration of an FPGA into the data path of a streaming leads to an actual improvement in *end-to-end performance.*

7.1 Latency and Throughput

Other than in software-based setups, the performance characteristics of hardware execution plans can accurately be derived by solely analyzing the circuit design. Thereby, the performance of a larger circuit is determined by the performance of its sub-plans. In the following, we first concentrate on latency, then investigate throughput.

	Issue Rate		**Latency**	
Query	non-opt.	opt.	non-opt.	opt.
Q_1	1	1	3	1
Q_2	1	1	5	1
Q_3	1	1	5	2
Q_4	1	1	5	2
Q_5	1	1	$6\ldots6+16G$	$5\ldots5+16G$

Table 3: Issue rates and latencies for optimized query plans of Q_1–Q_5.

7.1.1 Latency

We measure the latency of a hardware circuit in the number of clock cycles that occur from the time a tuple enters the circuit until the time a result item is produced. For the case of 'group by' queries, the relevant input tuple is the last tuple of the input stream. Our FPGA is clocked at a rate of 100 MHz. Each cycle of latency thus implies an observable latency of 10 nanoseconds.

In a sequential data flow, the latencies of all sub-plans behave cumulatively: the overall latency of the full plan can be obtained by summing up the latencies of all sub-plans. Parallel circuits (such as the sub-plan instances in a 'group by' plan) are determined by the latency of the slowest sub-plan. Without applying any of the optimization techniques of Section 6, this yields the latencies reported in Table 3 as "non-opt." (we discuss the details of Query Q_5 in a moment).

Non-Optimized Circuits. For the simple circuits (Queries Q_1 and Q_2), the total latency corresponds to the number of flip-flop registers along the data path. For Queries Q_3 and Q_4, the union operators at the plan top add another latency cycle due to their built-in FIFOs (cf. Section 4.4).

In Query Q_5, the difference in read and write speed of our content-addressable memory introduces a data dependence of the circuit latency. Thus, Table 3 reports lower and upper bounds for the latency at runtime. Once the circuit has seen all possible group identifiers (and thus filled its CAM), no write access occurs and the circuit responds after six cycles. By contrast, if G different new groups arrive in succession, their group identifiers queue up in the input FIFO of the 'group by' circuit and each one adds 16 cycles for the CAM write.

After Optimization. The optimizations we described in Section 6 reduced latency by eliminating intermediate flip-flop registers. As listed in Table 3, this reduces latency down to one or two clock cycles for Queries Q_1–Q_4. The use of a multiplexer (cf. Section 6.3) saves one latency cycle for Query Q_5.

Observations. Table 3 reports single-digit latencies for most queries. The latency of Query Q_5 clearly tends toward the optimum case in practice, since the arrival of a large number of new groups right before the end of a window is rare. With a cycle time of 10 ns, our FPGA typically responds in less than a micro-second.

7.1.2 Throughput

The maximum throughput of a circuit is directly dependent on its issue rate. With a 100 MHz clock, an issue rate

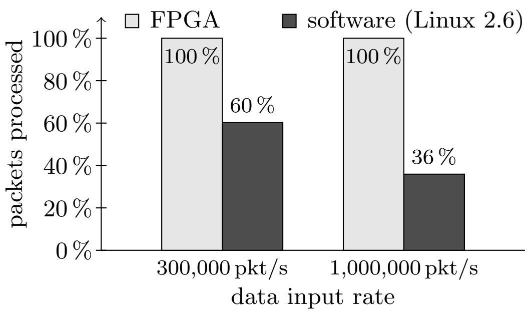


Figure 8: Number of packages successfully processed for two given input loads. The hardware implementation is able to sustain both package rates.

of 1 means that the circuit can process 100 million input tuples per second.

All our plans are fully pipelineable. As can be seen in Table 3, this leads to an issue rate of 1 for all five example queries. In the upcoming section, we are going to demonstrate how this enables us to process very high data rates at wire speed in a network-attached setup.

7.2 End-To-End Performance

A key aspect of using an FPGA for data stream processing is that the hardware circuit can directly be hooked into an existing data path. As already sketched in Section 3.2, we are particularly interested in using the FPGA as a preprocessor that operates between the physical network interface and a general-purpose CPU (though the idea could be applied to other data sources, too). To verify the effectiveness of this setup, we implemented it using an FPGA development board, then measured the data rates it can sustain.

The biggest challenge in commodity systems is to process network data with high *package rates* (as opposed to large-sized packages). Actual application setups in software start suffering at data rates of $\gtrsim 100,000$ packets/s because of the high intra-host communication overhead for every packet. By contrast, our query execution circuit is directly connected to the physical network interface. The experiments in the following show how this enables us to process significantly higher package rates at wire speed.

Our experiments are based on a Xilinx XUPV5 development board that ships with the FPGA mentioned in Section 3 and includes a fast 1 GBit Ethernet interface. We implemented the system configuration shown in Figure 2 (b) as an embedded system by instantiating the necessary hardware components as soft-cores inside the FPGA chip. Our CPU in this setup is a MicroBlaze CPU as provided by Xilinx.

It turns out that it is fairly difficult to generate really high package rates in a lab setting. With a NetBSD-based packet generator, we managed to generate a maximum of 1,000,400 packets/s (all UDP traffic). Still, this was not sufficient to saturate our hardware implementation. As illustrated in Figure 8, no data was lost when processed on the FPGA.

For comparison, we hand-crafted a light-weight network client on top of Linux 2.6, designed to accept and process the same input data at high speed. Yet, as shown in Figure 8, this client was not able to sustain the load we applied. For high package rates, it dropped more than half of all input tuples.

Our results clearly demonstrate that our circuit can meet the expectations we set. This makes FPGAs particularly attractive for common application scenarios. If the FPGA is used as a rate-reducing component in the data input path, the remainder of the system faces only a fraction of the input load. Effectively, this significantly increases the applied load that the system can sustain.

8. RELATED WORK

The idea of using tailor-made hardware for database processing dates back at least into the late 1970s, when DeWitt explored what was called a "database machine" at the time [3]. His DIRECT system used specialized co-processors that operated close to secondary storage and provided explicit database support in its instruction set.

While enormous chip fabrication costs rendered the idea not economical at the time, some companies started to commercialize similar setups recently. Sold as "database appliances", their systems provide hardware acceleration mostly for data warehousing workloads. Documentation about the inner workings of any of the available systems is rare, but it seems that some of the appliances have a lot in common with the setup we looked at.

The *Netezza Performance Server* (NPS) system [2] is built from a number of "snippet processing units" (SPUs). Each of these snippets includes a magnetic disk, tightly coupled with a network card, a CPU, and an FPGA. Similar to the setup we consider, the FPGA is used to filter data close to the data source (the disk in Netezza's case).

The heart of Kickfire's *MySQL Analytic Appliance* [15] is its so-called "SQL Chip." Judging from the product documentation, this chip seems to be bundled with DDR2 memory and connected to the base system via PCI Express. In essence, this appears to coincide with the co-processor setup that we briefly touched in Section 3.2 (cf. Figure 2).

Both systems appear to use FPGAs primarily as customized hardware, with circuits that are geared toward very specific (data warehousing) workloads, but are immutable at runtime. In *Avalanche*, we aim at exploiting the *configurability* of FPGAs. With *Glacier*, we present a compiler that compiles queries from an arbitrary workload into a tailor-made hardware circuit.

We share this aspect with other research projects that use FPGAs to support arbitrary software, written in commodity languages. The Kiwi project [10], *e.g.*, compiles C# code into FPGA circuits. The main challenge in such systems is the detection of independent sub-tasks that can be parallelized on the FPGA. Here we look at a much more constrained source language, with obvious handles for parallelism. In return, we address very high data rates and optimize our plan generation toward that.

Our processing model resembles the MonetDB/X100 system by Héman *et al.* [13]. MonetDB/X100 processes data from a column-wise storage in a pipelined fashion. We borrowed their idea of *selection vectors.* Invalidating tuples rather than physically deleting them avoids expensive in-memory copy operations in MonetDB/X100. Much like MonetDB/X100, our circuits favor narrow input relations/streams. An alternative processing mode tailored to wide tuples is already on our workbench.

Our implementation of 'group by' takes particular advantage of an FPGA-based implementation of content-addressable memory. Bandi *et al.* [1] have looked at a commercial CAM product and its potential applications in a database context. Tough such products can provide high capacity and lookup performance, we think that the missing coupling to a full database infrastructure renders the approach hard to apply in practice. FPGAs, by contrast, provide the flexibility to join an existing infrastructure in a seamless fashion, even at different locations if necessary. The work of Gold *et al.* [7] describes a similar approach with similar drawbacks. They suggest the use of *network processors* for database processing, mainly to exploit the thread-level parallelism inherent to network CPUs.

Others have explored various types of specialized processors for use in a database context (popular examples are GPUTeraSort [8] or stream joins for the Cell processor [6]) and they show promising performance characteristics. Given that specialized processors follow an architecture that is not inherently different to the one used in general-purpose CPUs, it is unclear, however, whether they can indeed overcome the limitations of commodity setups.

The higher-order nature of the 'group by' and windowing operators in our streaming algebra resembles the "Apply" operator that is used inside Microsoft SQL Server and has been discussed by Galindo-Legaria *et al.* [5]. Similar rewrite rules as the ones in SQL Server may also help the *Glacier* compiler to improve the quality of generated plans.

9. SUMMARY

The *Glacier* component library and compiler that we have presented in this paper are part of the *Avalanche* project at ETH Zurich. In *Avalanche*, we aim at building a streaming engine for heterogeneous many-core architectures that combine FPGAs and general-purpose CPUs.

As we have shown in this paper, *Glacier* provides an operator algebra and transformation rules that can be used to convert meaningful continuous queries into FPGA circuit descriptions. Among others, we provide full support for aggregation, grouping, and windowing. Since the performance characteristics of the operators implemented as FPGA circuits are very different than those of software operators, we provide an in-depth analysis of the relevant performance metrics.

Our results indicate that using the FPGA as a co-processor in an engine running on conventional CPUs can have significant advantages. The experiments show that most operators have very low latency and that the FPGA as a whole can sustain a very high throughput. The setup tested in the paper demonstrates that the FPGA can process streams at network speed (the bottleneck is the network interface, not the data stream processing on the FPGA), something that cannot be done in conventional CPUs.

Future work includes adding support for window joins and special-purpose operators like frequent item detection, as well as integration on a hybrid stream processing engine.

10. REFERENCES

[1] Nagender Bandi, Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Fast Data Stream Algorithms Using Associative Memories. In *Proc. of the ACM SIGMOD Int'l Conference on Management of Data*, pages 247–256, Beijing, China, June 2007.

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

[3] David DeWitt. DIRECT—A Multiprocessor Organization for Supporting Relational Database Management Systems. *IEEE Trans. on Computers*, c-28(6), June 1979.

[4] FIX Protocol Specification. `http://fixprotocol.org/specifications`.

[5] César A. Galindo-Legaria and Milind Joshi. Orthogonal Optimization of Subqueries and Aggregation. In *Proc. of the ACM SIGMOD Int'l Conference on Management of Data*, pages 571–581, Santa Barbara, CA, USA, May 2001.

[6] Bugra Gedik, Philip S. Yu, and Rajesh Bordawekar. Executing Stream Joins on the Cell Processor. In *Proc. of the 33rd Int'l Conference on Very Large Databases (VLDB)*, pages 363–374, Vienna, Austria, September 2007.

[7] Brian T. Gold, Anastassia Ailamaki, Larry Huston, and Babak Falsafi. Accelerating Database Operations Using a Network Processor. In *Workshop on Data Management on New Hardware (DaMoN)*, Baltimore, MD, USA, June 2005.

[8] Naga K. Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. GPUTeraSort: High Performance Graphics Co-processor Sorting for Large Database Management. In *Proc. of the 2006 ACM SIGMOD Int'l Conference on Management of Data*, Chicago, IL, USA, June 2006.

[9] Jim Gray, Adam Bosworth, Andrew Lyaman, and Hamid Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing GROUP-BY, CROSS-TAB, and SUB-TOTALS. In *Proc. of the 12th Int'l Conference on Data Engineering*, pages 152–159, New Orleans, LA, USA, February 1996.

[10] David Greaves and Satnam Singh. Kiwi: Synthesis of FPGA Circuits from Parallel Programs. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2008.

[11] Torsten Grust and Jens Teubner. Relational Algebra: Mother Tongue—XQuery: Fluent. In *Proc. of the 1st Twente Data Management Workshop (TDM)*, pages 7–14, Enschede, The Netherlands, June 2004.

[12] Steven A. Guccione, Delon Levi, and Daniel Downs. A Reconfigurable Content Addressable Memory. In *7th Reconfigurable Architectures Workshop (RAW 2000)*, volume 1800/2000 of *Lecture Notes in Computer Science*, Cancún, Mexico, May 2000.

[13] Sádor Héman, Marcin Zukowski, Arjen de Vries, and Peter Boncz. Efficient and Flexible Information Retrieval Using MonetDB/X100. In *3rd Biennial Conf. on Innovative Data Systems Research (CIDR)*, pages 96–101, Asilomar, CA, USA, January 2007.

[14] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 2006.

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

[16] StreamBase Systems, Inc. `http://www.streambase.com/`.

[17] Xilinx Inc. *An Overview of Multiple CAM Designs in Virtex Family Devices. Application Note 201*, September 1999.