Hasso Plattner Institute
Data Engineering Systems Group

Master Thesis

Run-time configurability of stream processing queries on FPGAs

Konfigurierbarkeit von Datenstromverarbeitungsanfragen auf FPGAs zur Laufzeit

Ben-Noah Engelhaupt

Matriculation Number: 790500

Supervisor
Prof. Dr. Tilmann Rabl

2^{nd} **Reviewer**
Prof. Dr. Andreas Polze

Submitted: 31.03.2023

Abstract

FPGAs have proven to be feasible accelerators for various data processing tasks, including stream processing. However, recompiling a design and reconfiguring an FPGA for dynamic queries is unfeasible due to long compilation times. This work proposes a stream processing system that can be compiled to hardware and reconfigured at run-time without recompiling the query or reprogramming the FPGA. Reconfigurable operators are introduced that can change their processing behavior through dynamic configuration signals. To not only make the operators of a query configurable but the query itself, an operator interconnect is proposed that allows dynamic communication paths between operators and therefore to rebuild entire queries without recompilation. This system has been analyzed with a focus on logic usage between different levels of configurability. It has been shown that configurability comes at a significant cost of logic resources and increased compilation time, while there are only small performance implications.

Zusammenfassung

FPGAs haben sich als praktikable Beschleuniger für verschiedene Datenverarbeitungsaufgaben, einschließlich Datenstromverarbeitung, erwiesen. Die Neukompilierung eines Hardwareentwurfes und Rekonfiguration eines FPGAs für dynamische Abfragen ist jedoch aufgrund langer Kompilierungszeiten unpraktikabel. Diese Arbeit stellt ein Datenstromverarbeitungssystem vor, das zur Laufzeit in Hardware kompiliert und rekonfiguriert werden kann, ohne die Abfrage neu zu kompilieren oder den FPGA neu programmieren zu müssen. Es werden rekonfigurierbare Operatoren eingeführt, die ihr Verarbeitungsverhalten durch dynamische Konfigurationssignale ändern können. Um nicht nur die Operatoren einer Abfrage konfigurierbar zu machen, sondern auch die Abfrage selbst, wird ein Operator-Interconnect vorgeschlagen, welches dynamische Kommunikationspfade zwischen Operatoren ermöglicht und damit den Aufbau kompletter Abfragen ohne Neukompilierung ermöglicht. Das vorgestellte System wurde unter dem Schwerpunkt der Logiknutzung bei verschiedenen Konfigurierbarkeitsstufen analysiert. Es wurde gezeigt, dass die dynamische Konfigurierbarkeit einen erheblichen Anstieg an Logikressourcen und erhöhte Kompilierungszeiten mit sich bringt, während es nur geringe Auswirkungen auf die Leistung gibt.

Contents

1 Introduction

Devices based on Field-Programmable Gate Arrays (FPGAs) are becoming increasingly popular in today's data centers with various application areas including smart NICs, specialized audio and video processing hardware, and general-purpose programmable accelerators. With the XEON Gold 6138P, Intel even introduced a processor with an on-package FPGA[1]. As their name suggests, they can be programmed to host custom processing circuitry tailored to an application scenario. Custom circuitry allows for very energy-efficient processing compared to CPUs [20].

FPGAs can be programmed to accelerate common workloads, one of which is stream processing. There has already been some work on offloading stream processing tasks to reconfigurable hardware, citing its low-latency processing capabilities and massive parallelism that custom circuitry makes possible. Contributions range from general-purpose stream processing where algebraic query plans are compiled to hardware [9], over network intrusion prevention systems [27] and multi-query stream processing systems [11, 14] to run-time-reconfigurable stream joining [12].

The end-to-end process of programming an FPGA accelerator consists of creating a circuitry definition ("design") in a hardware description language (HDL) or using high-level synthesis (HLS) to convert iterative and object-oriented C, C++, or even Python code into an HDL. This hardware description is then translated into gate-level macros in a processing step called "synthesis". After this step, the mapping of these macros onto the actual FPGA resources ("placement", "fitting") has to be determined. Furthermore, in a step called "routing", they are interconnected. As the last step, a programming file is generated which is used to program the FPGA.

The placement and routing steps depend on the actual FPGA hardware, such as the available logic cells and timing constraints. The placement is optimized and routing is performed over and over until all timing constraints of the design are met. Meeting these constraints is crucial for the hardware to perform in a stable and predictable way. This process is very compute-intensive and, depending on the design size, can take hours or even days to complete, even when reusing compilation artifacts or splitting the design into smaller modules that can be compiled independently from each other [5, 24, 25].

When using FPGAs for stream processing applications with a query compiled to

[1]https://ark.intel.com/content/www/us/en/ark/products/139940/intel-xeon-gold-6138p-processor-27-5m-cache-2-00-ghz.html

hardware, this greatly limits flexibility, because changing the query incurs long compilation times. This might not be a problem for long-running static queries, but when the behavior of certain stream processing operations has to change dynamically based on parameters (e.g., video compression factor in accelerated video processing; interest rate and risk factors for derivative pricing in algorithmic trading) or the structure of a query has to be modified altogether (e.g., additional selection stage), performing the entire compilation process and re-configuring the FPGA is unfeasible. The same is the case for ad-hoc queries: they cannot benefit from acceleration because compilation for an FPGA would take too long in most cases and they need to be executed on the CPU. Even using mechanisms like partial reconfiguration, where only a part of the FPGA chip is programmed, requires a query (or a part of it) to be already compiled.

This work will present a stream processing system that can be compiled to hardware and that can be reconfigured at run-time without the need to recompile the query and without the need to reprogram the FPGA. The configuration is transmitted within the data stream and applied instantly, so it allows to change query behavior without any additional interruption in processing. Depending on the size of the query to be configured, this can mean an end-to-end reconfiguration time within nanosecond range. This is more than ten orders of magnitude faster than an hour-long recompilation process.

Three stages of query configurability can be defined:

- **Non-configurable query** The behavior of the query is fixed at compile time and no configuration is possible.
- **Configurable query** Each of the query's operators can be configured at run-time but the structure of the query itself is fixed at compile time. E.g., change the window size of a word count query; change the predicate of a query's selection operator.
- **Composable query** A collection of operator modules is present on the FPGA which can each be configured. These operator modules can be chained together at run-time to compose a query. The configuration therefore acts as a query description language. E.g., transform a single query to a multi-query consisting of multiple independent sub-queries; create an ad-hoc query.

These stages have different logic complexity. For a configurable query, configuration registers need to be added. For a composable query, dynamic operator wiring logic has to be included additionally. These added layers of abstraction take away optimization potential. An FPGA's maximum clock speed is determined by the length of datapaths in the design. Because of more complex decision logic, these datapaths can be longer, possibly resulting in a reduction of the FPGA's clock speed. This in turn could reduce its throughput and increase latency. This work will investigate the impact that a configurable query and composable query have on throughput, latency, design size, compilation time, and other factors.

The presented system is built with configurability included, so the actual logic generated for the FPGA is static. As an outlook, such a static design could be transferred to an ASIC for general-purpose stream processing acceleration. As a rule of thumb, ASICs are an order of magnitude faster than FPGAs [7, 8, 23]. The results of this work could help in finding out whether such ASICs are feasible.

Section 2 will introduce basic concepts around stream processing and FPGAs. Section 3 describes how the streaming system works and how its operators are implemented. It also lays out how these operators can be made configurable and how they are chained together in a composable system. Section 4 explains how the presented system is integrated as an acceleration unit into an actual machine and how it interfaces. Section 5 will provide a static analysis of the streaming system that outlines properties such as logic usage and performance metrics of its operators. It also gives insights how the system performs in an integrated environment and what impact configurability has on the system's metrics. Section 6 will discuss the presented results and proposes points for further research. Other FPGA-based streaming systems are presented and compared with in Section 7, while Section 8 will conclude the work.

2 Background

This section will introduce the background knowledge necessary to understand this work. It will first explain FPGAs, their fields of application, and some concepts of digital logic. Stream processing as a data processing concept will be introduced thereafter.

2.1 FPGAs

FPGA is the short form for *Field-Programmable Gate Array*, a type of integrated circuit onto which digital circuitry can be programmed after the chip has been manufactured (*programmable in the field*). FPGAs can be used in a variety of dynamic application scenarios such as database acceleration [16] or as smart storage [6]. Historically they were used to prototype digital designs before they were going into manufacturing. This is still the case today with the largest CPU manufacturers Intel and AMD having acquired the largest FPGA manufacturers Altera and Xilinx respectively. Applications that can leverage pipelined and parallel execution can benefit most from FPGA acceleration, while iterative execution, complex data dependencies, branching logic, and much state are not suited well [7].

2.1.1 Compute flexibility and energy efficiency

On a flexibility range, FPGAs lie between ASICs and CPUs. ASICs are *Application-Specific Integrated Circuits* that are fixed in the functionality they can provide after manufacturing, unlike FPGAs which can be reprogrammed, and unlike CPUs, which usually allow flexible computation based on an instruction set. It has to be noted that CPUs are also a form ASICs since they are integrated circuits and cannot be reprogrammed, but they differ in their computation model. On an efficiency range, ASICs can perform tasks with very high energy efficiency since their circuitry can be optimized for the fixed functionality they provide. The computing flexibility of a CPU is paid for by a low energy efficiency since they allow for general computation. In this regard, FPGAs are in the middle ground between ASICs and CPUs. They achieve around an order of magnitude better energy efficiency than CPUs but are also around one to two orders of magnitude less energy efficient than ASICs for the same computation [7].

2.1.2 Components

Modern FPGAs are built upon four main resource types [7, 19]:

Look-up tables These components allow representing combinational logic functions. They compose of multiple input wires and usually one output wire. For each combination of input wire values, they can be programmed to output a specific value on the output wire. Modern FPGAs include look-up tables (*LUTs*) of multiple sizes, i.e., the number of input wires, to allow for efficiently mapping different levels of function complexity. Given the two input wires (x_1, x_2), a 2-to-1 LUT component to represent the logical *and* function would be programmed the following way: $(0, 0) \rightarrow 0; (0, 1) \rightarrow 0; (1, 0) \rightarrow 0; (1, 1) \rightarrow 1$.

Flip-flops Flip-flops are memory elements designed to hold state. They are a single-bit storage and save the value on their input wire on each rising edge of the clock cycle. They can be combined to form registers of different widths.

Hard components In addition to LUTs and flip-flops, which form the basis of reprogrammable logic, many FPGAs include additional components that are more complex and serve a specific purpose. This includes block random-access memory (*BRAM*) and certain digital signal processing units (*DSPs*) such as multipliers. Their circuitry is fixed and always present on the chip, that's why they are called "hard" components. While the same functionality could be built using LUTs and flip-flops (i.e., a "soft" component), hard components are more efficient and achieve better timing than their soft counterparts. Hard components, like BRAM blocks, are distributed across the FPGA fabric.

Interconnect LUTs and flip-flops are often present in combination as a configurable logic block (*CLB*) with the configuration stored in SRAM [19]. Interconnects allow to wire up these CLBs and hard components to form a complete circuit. Interconnects are based on "switches" that can route a signal from component to component. This routing and the configuration of the CLBs is determined when the FPGA is programmed, i.e., it does not change in operation.

All implemented functionality occupies a part of the available resources, so the complexity of the functionality will always be limited by the amount of logic resources available. Partial reconfiguration allows to swap out parts of the logic dynamically, but it is still not possible to have functionality that does not fit entirely on the FPGA. This is especially the case for applications that require to keep a lot of state since the amount of BRAM on an FPGA is typically less than one hundred Megabytes and off-chip DRAM has a high access latency, degrading performance [7].

2.1.3 Logic circuits

One can make a distinction between combinational logic circuits and sequential logic circuits. Combinational logic does not rely on a clock signal and constitutes only of LUTs and no flip-flops as it does not hold state. Sorting networks are an example of combinational logic: given a collection of input elements on its input wires, the circuitry will after some time have the sorted collection on its output wires [10]. The time it takes for the signal to propagate from the input wires through the logic gates to the output wires is effectively determined by the length of this signal path and the propagation speed of the signal.

More complex processing, or processing that involves state, might not be able to be represented in only combinational logic, but in sequential logic. To preserve for instance the output of the sorting network, one can use registers constituted of flip-flops. On the rising edge of the clock, a flip-flop will store the value of its input wires, which means that the output of the sorting network has to be stable at that time. If the sorting network is also driven from a register, then the maximum time the signal can take to propagate through the combinational circuit is determined by the time difference between two rising edges of the clock - the duration of a clock cycle. The maximum clock frequency that an FPGA can be driven at is therefore the inverse of the propagation time of the longest combinational signal between registers. This abstraction level is called the ”register-transfer level” (RTL). If an FPGA is driven at a clock frequency that is too high, then the register input signal might not have fully stabilized, leading to wrong results being stored (e.g., a wrongly sorted list). The excess time it takes the signal to stabilize in this case is called ”slack”.

2.1.4 Programming process

The first step in programming an FPGA is creating the desired functionality in a digital circuitry design. Such a design can be created using a hardware description language (HDL) like VHDL or Verilog. It is also possible to use High Level Synthesis (HLS) to compile C-style code into an HDL. HDLs exhibit a declarative programming style and functionality is usually represented as combinational circuits or state machines for more complex operations.

In a step called ”synthesis”, the hardware description is then converted into a representation on the logic-gate level. This representation is universal and compatible with multiple FPGAs because it does not rely on the actual FPGA resources yet. First optimizations are already carried out during synthesis, like the removal of logic that is unused, the resolution of constant signals, or the optimization of state machines [18].

The next step determines how the logic elements of the synthesized gate-level design map to actual resources on the FPGA chip. This process is often called ”placement” or ”fitting” and has to take constraints such as the desired clock frequency into account. Since the maximum clock frequency is determined by the longest combinational signal path, this step has to place components belonging to the same combinational path in close proximity to each other.

The same level of optimization is necessary for the ”routing” step which defines how the placed components are wired together using the interconnects. The fitting and routing are optimized until the timing constraints are met, i.e., the longest combinational signal path fits within one clock cycle. This optimization process is very compute intensive and can, depending on design size, take multiple hours [5, 19, 24]. *Incremental compilation* describes the approach of reusing compilation intermediates, just like modern C++ compilers for example do. But since the designs on FPGAs are highly optimized, even small changes can require a full compilation run to achieve timing again [5].

The last step is to generate a ”bitstream” which is the device-specific representation of the configuration of the chip’s components. This bitstream can then be used to configure the FPGA. Using *partial reconfiguration*, only a part of the FPGA chip is configured while the rest retains its configuration.

2.2 Stream processing

Akidau et al. define a stream processing system as a ”data processing engine that is designed with infinite datasets in mind” [1]. Database systems operate on tables, which constitute a dataset at a specific point in time. Streaming systems on the other hand operate on streams, which encompass the evolution of a dataset over time on an element-by-element basis [1]. The elements of a stream can therefore have a temporal relation to each other.

Records Within this work, the elements of a stream are called *records*. They are of a certain size and can contain arbitrary data. While not strictly required, one piece of data included within a record can be its *event time* timestamp: the reference time of the record at which for example the event occurred that led to the ingestion of this specific record into the stream.

Operators The stream can be processed using operators. Operators modify the stream by adding records, removing records, or modifying records. Some operators are stateless, i.e., they process a record agnostic of the stream, while other operators keep state. The system presented in this work keeps only local (i.e., operator) and no stream-global state. This work calls a collection of operators that process a stream a *query*.

Bounded processing Streams are thought of as infinite but some processing only makes sense on bounded data. Windowing is a concept to define bounded parts (*windows*) within a stream, which can also overlap with each other. These bounds can be based on the number of records within a window or the duration of a window based on the record event times. The simplest windowing concept are tumbling windows which have a fixed number of records or duration. They are non-overlapping as the end of one tumbling window is the start of the next window. Sliding windows are also of fixed length but are overlapping based on a *slide*. A window will start every *slide* many records or when the *slide* duration has surpassed. This means that there will always be $\lceil length \div slide \rceil$ many sliding windows at a time. Tumbling windows are a special case of sliding windows where $length = slide$ [1].

Multi-query processing When referred to in this work, *multi-query stream processing* covers two meanings: (1) processing the same stream using multiple different queries or (2) processing multiple streams using multiple queries. While in (1) there is one input stream and multiple output streams, in (2) there are multiple input and output streams.

3 System description

A stream processing system has been designed in which queries can be reconfigured at run-time. Different levels of query configurability can be defined. In a non-configurable query, the query structure and operator behavior are static and cannot be changed. In a configurable query, the query structure is static but the individual operator behaviors can be changed. In a composable query, both the query structure and operator behavior can be changed at run-time, allowing for flexible processing. This section provides a description of this system, how it functions, how it is implemented, and what queries can be realized. Section 3.1 describes the high-level design and introduces fundamental concepts such as tuple propagation. Section 3.2 provides implementation details of certain stream processing operators, how they can be made configurable, and how they integrate into the system. Section 3.3 demonstrates how the configuration is propagated to operators and Section 3.1.2 presents sample queries based on the introduced operators. Section 3.5 describes how these configurable operators can be composed into a query of which the structure itself can be changed without recompilation.

System bounds The "system" being referred to in this chapter shall be limited to the data stream processing part that forms a query on the FPGA and has a record-in-record-out interface, excluding source and sink (which may be outside the FPGA). How records are transmitted to and from the FPGA and what intermediate processing is necessary on the FPGA to realize such an interface, is outlined in Section 4.

3.1 General

3.1.1 Record representation and propagation

On the hardware level, a record of n bits width is represented by n parallel wires. That means a record is always propagated in its entirety instead of, e.g., a byte-wise representation. This has the advantage that a record is always entirely available in a single clock cycle and not just parts of it. This approach does not scale well for records of *very* large size as all other circuitry like operators needs to be designed to also handle whole records, which can lead to a high usage of logic resources.

These data wires can propagate one record per clock cycle. Since there might not be a record available each clock cycle, an additional "data valid" signal indicates whether the record currently on the data wires is considered part of the stream. If this single wire carries an electrical "high" signal, then the record is part of the stream, else not.

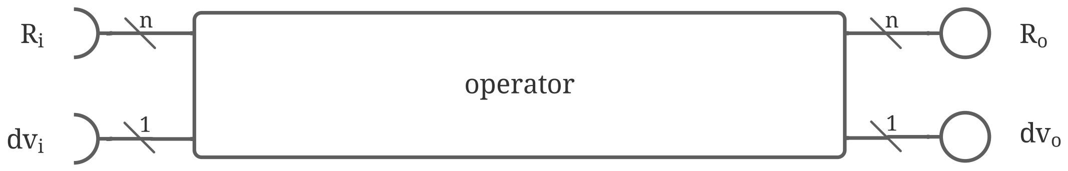


Figure 1: Operator interface for record propagation.

As visualized in Fig. 1 the minimum interface of an operator consists of n input and n output wires for the bits of the record and one input and one output data valid wire. If the input data valid wire dv_i is high, an operator processes the record on the input data wires R_i. To signal output, the operator puts a record on the output data wires R_o and sets the output data valid wire dv_o to high. At all other times, dv_o must be set to low to indicate that whatever is on R_o should not be considered output. Throughout this section, the interfaces of operators are extended further to achieve certain functionality, e.g., the *eow* signal used for windowing.

3.1.2 Queries

The push-based record propagation sets the foundation for communication between operators. By connecting the output wires of an operator to the input wires of another operator, individual operators can be chained together to form a query. Since stream source and sink are not considered part of the query, it therefore has the same record-based interface as an operator, as depicted in Fig. 2. The query as a collection of operator stages is fully pipelined because the operators can accept and propagate a new record in each clock cycle.

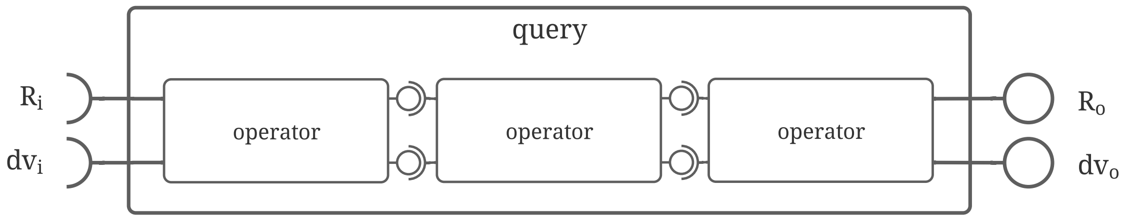


Figure 2: Query formed by combining operators.

It has to be noted that operators are not limited to having only one input and output stream. Such operators include windowing, stream fork and stream join, which will be presented in the upcoming Section 3.2.

3.1.3 Handling back pressure

Some operators, like sorting, are not able to accept a record in each clock cycle. That means an operator might have an output record but its downstream operator cannot accept it. In this case, the downstream operator has to be able to signal upstream that it is "busy" so it does not receive any more records. There are two main concepts on how to realize such functionality and they differ in how they influence the stream source. For both, the respective operator has an additional output wire "busy" that is set to high if it cannot accept records at the moment.

Local buffering This concept does not influence the source. A signal buffer is placed directly upstream to the operator that cannot accept records all the time. This signal buffer has an additional input wire "wait" which is connected to the downstream "busy" signal. This signal acts as a switch in the behavior of the buffer. If it is high, then the buffer will not output records and instead queue up all its incoming records (FIFO) in BRAM. Once the signal is low, the buffer starts outputting these queued-up records and once this queue is empty, incoming records are just passed through without buffering. All operators upstream to the buffer are not influenced by this process. Therefore the buffer only acts as a mechanism to alleviate small spikes in record frequency but is not suited for controlling the overall query throughput. A source that produces a higher consistent throughput than what the operators in the query can handle, will eventually lead to overflowing buffers.

Global buffering Contrary to local buffering, global buffering does influence the source by throttling it when an operator within the query cannot accept any more records. For this to function, all "busy" signals within a query are combined with an *or*-gate and produce a query-global "busy" signal which becomes part of the output interface of the query. The source has to react to this signal and stop delivering records to the query when it is high. Since the source is outside of the system bounds, please see Section 4 how this can be achieved on a system-integration level. This concept allows for controlling overall query throughput but will react over-sensitive to small spikes in record frequency as input for the entire query is halted and not just the busy operator.

Combining local and global buffering Both concepts can be combined to allow for alleviating small spikes in record frequency but also allowing for flow control. A "full" signal that is high when the queue is full (or almost full) is added to the local signal buffers. The "full" signals of all signal buffers in the system, combined with an *or*-gate, form the query-global "busy" signal. That means as long as the local buffers still have capacity, the source continues to deliver records. Only when one of the buffers becomes full, the source is throttled. This approach combines the advantages of local buffering and global buffering by allowing for controlling overall query throughput, while not reacting over-sensitive to spikes of record frequency because these are mitigated by the local buffers.

3.2 Operators

This subsection presents operators and their interfaces. An operator's interface can be separated into three main parts:

Processing interface. Consists of all signals related to the data stream and processing logic. The minimum processing interface of an operator is depicted in Fig. 1. These signals change frequently as they carry the stream's data. They comprise of input and output signals.

Configuration interface. Consists of input signals that define the concrete behavior of an operator. In a non-configurable system, configuration signals change not at all and are resolved at synthesis time. In a configurable system, they only change infrequently in comparison to the data stream.

System interface. Includes the clock and reset input signal which all operators receive. The reset signal is necessary since operators may hold internal state that has to be reset, e.g., after a reconfiguration. To trigger such a reset, the signal must be held high for one clock cycle.

For each operator, relevant performance figures will be presented alongside their implementation. For Glacier, Mueller et al. introduced "latency" and "issue rate" as basic performance metrics, which this work will adopt. However, this work will use the more common term "throughput" instead of "issue rate". Latency for an operator defines how many clock cycles pass between receiving a record and that record affecting the operator's output. Throughput on the other hand defines how many records on average an operator can accept per clock cycle. The system presented in this work can propagate one record per clock cycle, therefore the maximum achievable throughput is 1 record per clock cycle.

3.2.1 General concepts

Configurability An operator can have a configuration defined through its configuration interface signals. This configuration defines the concrete behavior of the operator. In a configurable system, this configuration, and therefore the behavior of an operator can change at run-time. In a non-configurable system, these configuration signals are assigned constant values. Since the behavior cannot change at run-time, the synthesizer will optimize unused logic away. Taking an aggregation operator as an example, in a configurable system, logic for all modes of operation (*sum* aggregation, *count* aggregation, ...) is present, while in a non-configurable system, only the predetermined functionality is available (e.g., a *sum* aggregation). In a configurable system that allows changing record sizes, resources need to be overprovisioned. One can define a maximum record size that determines the width of the R_i and R_o signals of an operator. This maximum record size can be set to the number of bits that can be ingested by the host system into the FPGA in a single clock cycle (see Section 4).

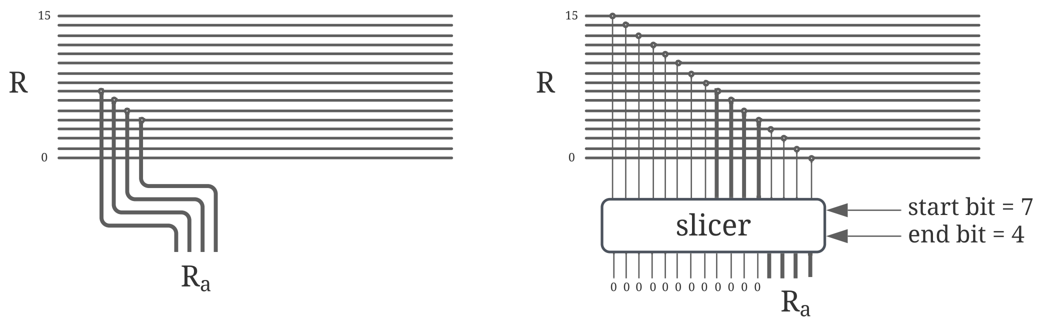


Figure 3: Extracting a field R_a consisting of bits 7 to 4 in a non-configurable system (left) vs configurable system (right).

Record fields Some operators do not work on the entire record but only a part of it. An example of this is a selection operator where only a field of the input record may be used for the comparison. A *field* can therefore be defined as a contiguous range of bits of a record. In a configuration interface, a field can be represented by two numbers: the index of the start bit within the record and the index of the end bit within the record. While custom logic like in an FPGA allows for such bitwise addressing, it is rarely practical and bytewise-addressing is used instead (i.e., start byte and end byte). On the hardware level, a field is represented by the respective subset of record wires. In a non-configurable system, this subset of wires is known at synthesis time and the resulting logic for extracting the field from the record is fairly simple as seen in Fig. 3. In a configurable system, the field could be arbitrarily defined. A "slicer" component takes the entire record as input and uses multiplexers to extract the specified range of wires. Since the field can be arbitrary in length, the slicer output is always the same width as the input record. Only the lowest bits are set to represent the field and the rest of the output is padded with logic lows. When synthesizing such a slicer with constant field bounds, the resulting logic is equivalent to what can be seen in Fig. 3 on the left. By using byte-wise addressing instead of bit-wise addressing for fields, the logic complexity of such a slicer is reduced significantly.

3.2.2 Selection

The selection operator allows comparisons based on a binary operator θ of a field a with another field b ($\sigma_{a\theta b}(R)$) or with a value constant v ($\sigma_{a\theta v}(R)$).

Processing interf.		**Configuration interface**	**System interface**
Inputs	**Outputs**		
R_i	R_o	$mode \in \{field, constant\}$	clk (unused)
dv_i	dv_o	$function \in \{0, 1, =, \neq, <, \leq, >, \geq\}$	rst (unused)
eow_i	eow_o	*field_a* (*_start*, *_end*)	
		field_b (*_start*, *_end*)	
		$constant \in \mathbb{N}$	

Table 1: Interface of the selection operator. The *eow* signals are used for windowing.

Interface Table 1 shows the operator interface. It is first to note that selection can be implemented as a combinational logic circuit, i.e., it is not reliant on a clock signal. Without a clock signal, no state can be held, so the operator further leaves the reset signal unused. The configuration interface determines the concrete operator behavior. The *mode* signal specifies whether field a is compared with another field b or a *constant*. The *function* signal determines the binary operator θ to use. Given that not the whole input record R_i but its field a is input to θ, the configuration signals *field_a_start* and *field_a_end* determine a as a part of R_i.

The field b is extracted in the same way. See Section 3.2.1 how this field extraction is achieved.

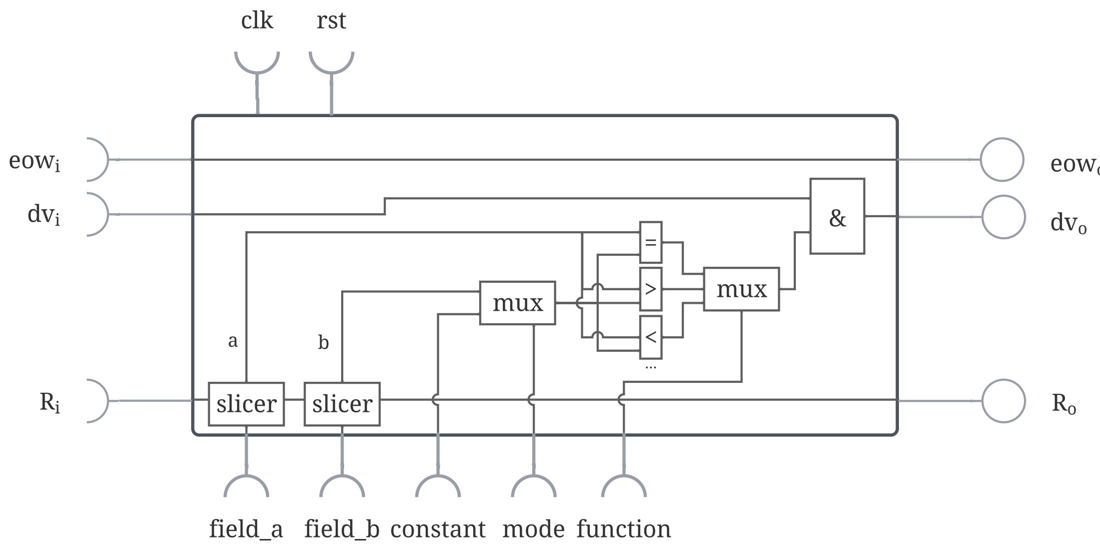


Figure 4: Configurable selection operator schematic.

Implementation Figure 4 shows a simplified schematic of the implementation of the operator. Fields a and b are extracted by "slicer" components. The *mode* is input to a multiplexer that switches between the *constant* and b, while $function$ is input to a multiplexer that switches between the results of all possible θ's. The θ's are implemented using a logic comparator for each of the input bits. The output is a single wire that is low when the comparison is false, and high when it is true. For contradiction ("0") this wire is always low and for tautology ("1") it is always high, so no comparators are needed there.
The selection operator's purpose is to determine whether an element of its input stream shall be part of its output stream. This is exactly what the dv signals in the operator interface are used for. Given dv_i and the output of the θ-multiplexer, to determine dv_o, these two wires just have to be combined by a logic *and* (&) gate: iff there is an input record as determined by dv_i and the selected θ outputs high, then the selection operator signals an output record. The output record R_o is equivalent to the input record, so R_i's wires can simply be passed through the operator.
If the configuration is constant, i.e., in a non-configurable system, then the synthesizer will optimize unused logic resources away. Figure 5 shows the logic that effectively remains when doing an equals-comparison with the constant 9. One

can see that the logic usage in a non-configurable system is significantly lower.

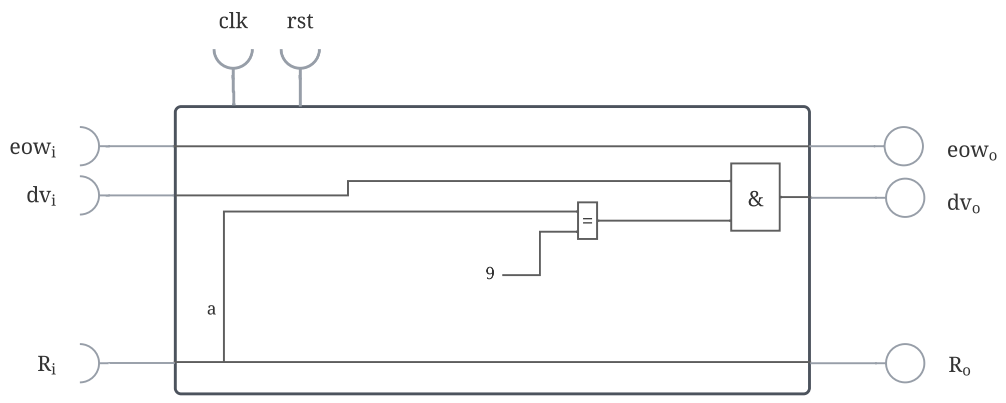


Figure 5: Circuitry that is left when comparing field a with the constant 9 for equality.

Performance Selection is implemented as a combinational circuit, therefore resulting in a special case regarding latency and throughput. Latency is a fraction of a clock cycle while throughput is larger than one record per clock cycle. To illustrate this, imagine four selection operators in sequence (and assuming the resulting signal path fits within one clock cycle), then the latency of a single operator would technically be a quarter of a clock cycle and the throughput would be 4 records per clock cycle. If a single selection operator is placed between two non-combinational operators, then its latency is technically zero and throughput is one record per clock cycle. For simplicity, this work assumes that as the general case and will refer to the operator having a latency of zero clock cycles and throughput of one record per clock cycle throughout the rest of this work.

3.2.3 Projection

The projection operator allows for selecting a subset of bytes of the input record to be present in the output record ($\pi_{b_1,b_2,\ldots,b_n}(R)$). Common CPU-based stream processing and database systems also allow ordering those bytes in a certain way. Implementing reordering of all bytes in hardware would not only entail an overly complex configuration, but the resulting logic would also require a lot of logic resources. Therefore the projection operator allows for moving *one* range of bytes of the record to a new position within it. Should multiple reorderings be necessary, multiple projection operators can still be placed in sequence. Table 2 shows the operator's interface. Similarly to the selection operator, the entire projection operator is based on only combinational logic.

Processing interf.		**Configuration interface**	**System interface**
Inputs	**Outputs**		
R_i	R_o	$move\_field$ ($\_start$, $\_end$)	clk (unused)
dv_i	dv_o	$move\_dest$	rst (unused)
eow_i	eow_o	$keep\_bytes$	

Table 2: Interface of the projection operator.

Interface The $move\_field\_start\_byte$ and $move\_field\_end\_byte$ configuration fields determine the range of the record to move, while the $move\_dest$ field specifies the destination byte index to move it to. The $keep\_bytes$ bitmask determines which bytes of the input record shall be part of the output record.

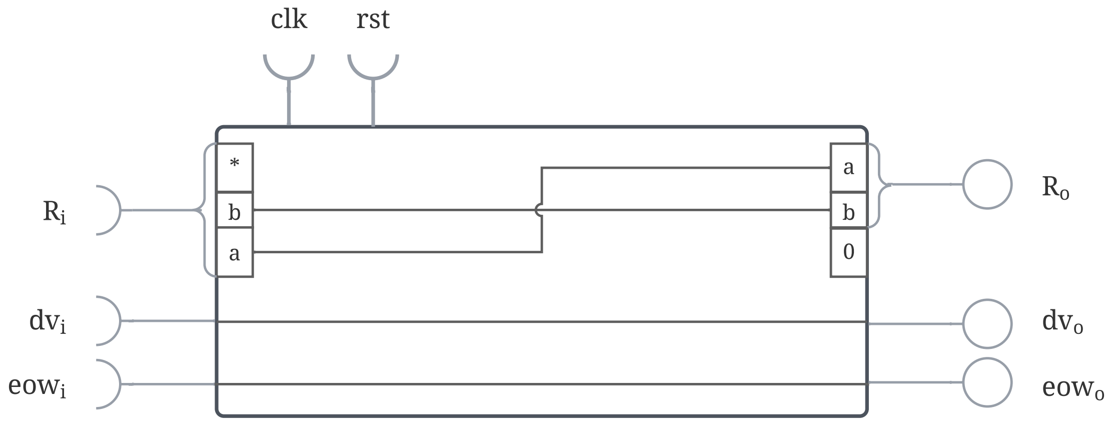


Figure 6: Circuitry generated when moving field a in front of b and discarding the rest of R_i.

Implementation The operator will first perform the "move" operation and apply the projection operation on the move's result. The $move\_field$ will be moved to the destination byte $move\_dest$ with the replaced bytes being shifted to the field's original position. The bytes to keep, based on the bitmask $keep\_bytes$, are placed at the end of the record with the rest of the record being padded with logic lows ("0").
Figure 6 shows circuitry for an example projection. The operator only has an effect on the output record R_o, so all other signals such as dv can simply be wired

through. One can see that a move operation will only switch wire positions in the output record relative to the input record. The wires of bytes discarded in the projection are simply not connected to the output record. In a non-configurable system, this implementation has the effect that the synthesizer will observe that the discarded wires of R_i have no effect on downstream operators (and therefore the query output). As already laid out by Mueller et al., the synthesizer can optimize away these ”dangling wires” and effectively realize a projection pushdown. In a configurable system, the output record may change at any time given the configuration parameters, rendering such optimizations impossible.

It might seem like the operator violates the *single responsibility principle* of software engineering because it provides multiple modes of operation and/or functionalities at once. But in the context of operator reconfigurability, allowing for more functionality within a single operator entity creates greater flexibility in reconfiguration.

Performance For the projection operator, the same special case as for the selection operator applies. In the general case of a projection operator being placed between two non-combinational operators, latency is zero and throughput is one record per clock cycle.

3.2.4 Windowing

The general windowing principle is based on signaling the end of windows in-stream, a concept also used by Mueller et al. in their streaming system Glacier[9]. Similarly to how the *dv* signal is used to indicate whether a given record belongs to the data stream or not, an *eow* (”end of window”) signal is used to indicate whether a window ends with the current clock cycle. This signal is propagated throughout the system in the same way as the *dv* signal and record are - from operator to operator. Usually, an aggregation operator will then use this signal as an indicator as to when to emit window aggregates. A windowing operator therefore opens a window domain, while an aggregation or window synchronization operator closes it. This concept avoids having to keep large state, such as collecting a window's records in memory. On the other hand, this introduces the limitation that the stream has to be in-order when windowing is performed based on event time.

In the most general case, tumbling windows, there is only one open window domain at a time. Using sliding windows there are $\lceil window\_size \div window\_slide \rceil$ many open window domains at a time: for instance, with a window size of 8 and window slide of 2 there will be always four open window domains at once. Because of the inherent parallelism an FPGA design can make use of, these multiple open window domains can be processed in parallel. A windowing operator splits the stream into multiple processing branches, each branch only operating on one window domain. As seen in Fig. 7 these branches are later joined by a window synchronization operator after the window domains are over (i.e., usually after an aggregation).

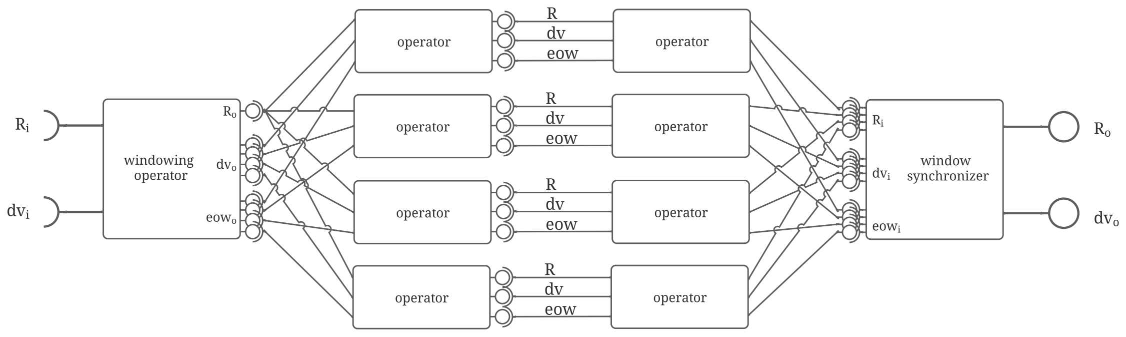


Figure 7: Windowing mechanism consisting of four parallel processing branches.

The windowing operator outputs the same record to all processing branches but sets the data valid signal *dv* for a branch depending on whether this record belongs to the respective window. It also outputs an *eow* signal to each branch to indicate the end of the window. This concept allows for flexible windowing schemes because a record can belong to multiple windows and multiple windows can end at the same time. Now two sliding window operators will be presented: fixed-size windowing and timestamp-based windowing.

Fixed-size windowing This operator produces sliding windows with a fixed number of elements within each window. The processing is therefore time-agnostic as it does not rely on any time information present in the record. For this operator, the configuration interface only consists of the *window_size* and *window_slide* signals as can be seen in Table 3. It counts the elements that have already been emitted for each window and once the *window_size* is reached, the *eow* signal will be driven high for one clock cycle and the counter is reset to 0. These counters are initialized with negative values of multiples of the *window_slide* to achieve the correct window begins. A record will only be emitted to a certain window when its counter is larger than or equal to 0 to avoid windows being longer than *window_size* in this initialization phase.

In a non-configurable system, the windowing configuration is static and therefore

Processing interf.		**Configuration interface**	**System interface**
Inputs	**Outputs**		
R_i	R_o	$window\_size$	clk
dv_i	$dv_o[1..n]$	$window\_slide$	rst
	$eow_o[1..n]$		

Table 3: Interface of the fixed-size sliding windowing operator. n denotes the number of provisioned processing branches.

the number of parallel window branches is defined. But when this configuration changes at run-time, the number of required processing branches also changes. Hence processing branches can be over-provisioned in a configurable system to allow such flexibility, up to a defined number of maximum supported parallel processing branches.

Since the operator simply passes records onto their respective processing branches in parallel, it can accept one record per clock cycle, resulting in a throughput of 1. It does so with a latency of one clock cycle.

Timestamp-based windowing Contrary to the fixed-size windowing operator, this operator determines windows based on the record's event-time. As can be seen in Table 4, the configuration interface contains additional signals that define the field of the record which contains the event timestamp. The $window\_size$ and $window\_slide$ parameters are in the same duration unit as this timestamp. The operator assumes the stream to be in-order, i.e., these timestamps to monotonically increase. This allows to follow an approach similar to fixed-size windowing but instead of keeping track of the number of records within each window, the window start timestamps are stored. When a record with a timestamp larger than $window\_start + window\_size$ arrives, then the eow signal is driven high for the respective window branch. The sliding is achieved by initializing the window starts when the first record of the stream arrives: the start timestamps are set to the record's timestamp incremented by multiples of the $window\_slide$.

If the stream is out-of-order, there could theoretically be infinitely many open window domains at once. In the current windowing architecture, infinitely many processing branches would be needed, which the limited resources of an FPGA cannot account for. Another approach would be to queue up records of windows in memory as many CPU-based systems do, but FPGAs are generally equipped with very little memory resources. A practical approach would be to create fixed-size sliding windows on the stream and to sort the records in these windows. These

Processing interf.		**Configuration interface**	**System interface**
Inputs	**Outputs**		
R_i	R_o	$window\_size$	clk
dv_i	$dv_o[1..n]$	$window\_slide$	rst
	$eow_o[1..n]$	$timestamp\_field(\_start, \_end)$	
	$busy_o$		

Table 4: Interface of the timestamp-based sliding windowing operator. n denotes the number of provisioned processing branches.

windows can then be processed as an in-order stream to get approximate results.

A window is empty when there are no records between two *eow* signals. These windows also need to be processed to produce correct aggregation results (e.g., an element count of 0 within a window). Since the timestamp-based windowing operator relies only on record timestamps to detect the end of windows, there is the case that two successive records have a timestamp difference that requires multiple empty windows to be emitted. While the operator is performing this process it cannot accept any more incoming records. This blockage lasts as many clock cycles as there are windows to be emitted divided by the number of parallel processing branches, as one *eow* signal can be transmitted on each processing branch per clock cycle. The $busy_o$ signal is set to high until all these empty windows are emitted to signal upstream that records need to be buffered. This is one example of the local buffering mechanism described in Section 3.1.3.

The throughput of the operator therefore depends on how often empty windows have to be emitted. If this is never the case, then its throughput is one record per clock cycle, otherwise it is $1 - P_{empty\ window}$ where $P_{empty\ window}$ is the probability of an empty window having to be emitted in a clock cycle. The latency is unaffected by this and is always one clock cycle.

Window synchronization Each window is processed by one of the parallel processing branches. The streams that these branches output need to be merged into a single stream again. A component called "window synchronizer" is responsible for that. It takes the records $R_i[1..n]$, data valid signals $dv_i[1..n]$, and end of window signals $eow_i[..n]$ of all parallel processing branches as input and outputs a single stream based on R_o and dv_o, as depicted in Table 5. Its configuration interface only contains the number of processing branches that are actually in use (since they can be over-provisioned in a configurable system) to not try to merge unused branches.

Processing interf.		**Configuration interface**	**System interface**
Inputs	**Outputs**		
$R_i[1..n]$	R_o	*num_windows*	*clk*
$dv_i[1..n]$	dv_o		*rst*
$eow_i[1..n]$			

Table 5: Interface of the window synchronizer. n denotes the number of provisioned processing branches.

The merge process has to take the semantics of the windowing itself into account. Since the windowing operator creates windows in a certain order, the same order has to be preserved after the merge. Given windows w_1 and w_2 for example, where w_2 is a window that starts after w_1. If w_2 is processed faster than w_1, the output records of w_2's processing could arrive at the merge operator earlier than those of w_1. If the merge process would merge records only on the basis of when they arrive, window processing results can get out-of-order. The window synchronizer therefore has to perform a semantic merge, where it preserves the window order.

This synchronization is achieved by keeping track of which processing branch is expected to output records next and only passing those records onto the output stream. Since both presented windowing operators effectively assign windows to processing branches in a round-robin fashion, the synchronizer expects branches to output records in this same order. Once a processing branch indicates that the window currently processed has ended, i.e., using the *eow* signal, the window synchronizer expects records from the next processing branch. While the window synchronizer waits for the expected processing branch to output records, the outputs from all other processing branches need to be buffered. The window synchronizer has one FIFO-style buffer for each processing branch. As in local buffering for handling back pressure, a *wait* signal on the buffer is used to control whether it should output or queue up records. This *wait* signal will be set to output records only for the buffer belonging to the processing branch of which records are expected. The buffers will not only store records but also the *eow* signal to not lose the association between both. Otherwise, given the contents of a buffer, it could not be determined whether all the records belong to the same window or multiple windows, again breaking window order.

As for performance metrics, the operator has a throughput of one record per clock cycle as it buffers the incoming records of processing branches that it would not be able to accept otherwise. The latency of a record depends on the input stream. If all records arrive in the order of their respective windows, no buffering is necessary and the operator has an overall latency of one clock cycle for the processing part and three clock cycles for the buffering part, since records still need to be passed through the buffers (see Section 3.2.9).

3.2.5 Grouping

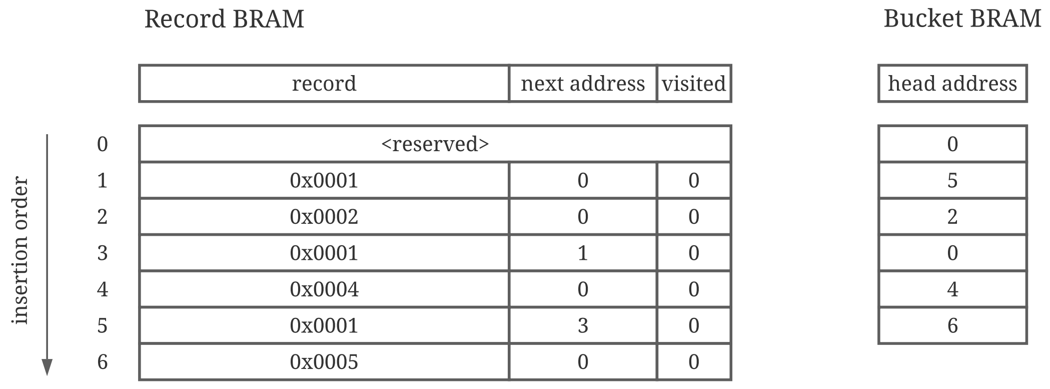
The grouping operator reorders the records of a window so that records with the same value of their field a appear successively (γ_a). A simple approach to achieve this functionality is sorting the records by this field a using the sorting operator. A more efficient solution in terms of algorithmic complexity can be achieved by making use of a hash map, albeit it incurs a higher memory usage. There is only a limited amount of BRAM on an FPGA. Since all records of a window need to be stored in the grouping operator, this puts an effective maximum processable records per window limit into place.

Interface As with all other operators, the grouping operator receives records using the R_i and dv_i signals. In comparison to other operators, it does not output records in the order in which they arrive. In fact, it needs to wait until it has seen the last record of a window, as indicated by eow_i, to determine the final output ordering. The records are then emitted using the R_o and dv_o signals and after all records of the window have been emitted, the eow_o signal is driven high for one clock cycle. The implementation of the operator requires local buffering upstream which the $busy_o$ signal is for. The field after which to group by is determined through the $field\_start$ and $field\_end$ signals in the configuration interface.

Processing interf.		**Configuration interface**	**System interface**
Inputs	**Outputs**		
R_i	R_o	$field(\_start, \_end)$	clk
dv_i	dv_o		rst
eow_i	eow_o		
	$busy_o$		

Table 6: Interface of the grouping operator.

Implementation The grouping operator relies on a hash map to group records. The field a to group after is extracted using a slicer component for each record. This field is then input to combinational circuitry based on multipliers and shifters, which resembles a hash function. The hash of the field determines the bucket in which to place the record. Each bucket has a linked list of records associated with it. The operator utilizes two BRAMs, one for storing the records (”record BRAM”) and one for storing the heads of the linked lists (”bucket BRAM”) as depicted in Fig. 8. An entry in the record BRAM consists of the record itself, the record BRAM address of the next record with the same hash, and a ”visited” bit to determine whether a record has already been emitted. Entries in the bucket BRAM only consist of the record BRAM address of the first record in that bucket. These heads are initialized with zero, indicating a ”null pointer” and that there are no records belonging to this bucket. That is also the reason why the record BRAM address 0 is reserved.



	record	next address	visited
0	<reserved>		
1	0x0001	0	0
2	0x0002	0	0
3	0x0001	1	0
4	0x0004	0	0
5	0x0001	3	0
6	0x0005	0	0

head address
0
5
2
0
4
6

Figure 8: BRAM layout and content after inserting some records. The records are hashed here using the identity function, so $hash(0x000n) = n$.

If a record is received, the hash of its field is calculated. Then the current head of the corresponding bucket is retrieved. This retrieval consists of setting the BRAM read address to the hash and stalling for one clock cycle until the head address is on the BRAM’s data wires. The operator signals upstream that it cannot accept a new record by setting $busy_o$ high for this clock cycle.

The record is then inserted at the beginning of the linked list, i.e., it becomes the new head. Records are inserted into the record BRAM in the order they arrive, so the record BRAM’s write address is set to the next empty address and the data wires contain the record, the current bucket head, and a zero visited bit. In the same clock cycle, the bucket BRAM is updated by writing the address of the just-inserted record as the head of the bucket. Writing to record BRAM and bucket BRAM during the same clock cycle is the reason the two are separated. If a single BRAM would have be used, one could only perform one write operation in each clock cycle, requiring more clock cycles for an insertion. The operator has to stall for one more clock cycle and indicates that it is ”busy” until the data has been written into the BRAMs and a new record can be accepted.Figure 8 shows the

BRAM contents after some insertions. One can observe that the head address of a bucket will always point to its latest inserted record.

This process continues until the window end is indicated by a high eow_i signal. As the operator has to emit the groups now, it cannot accept incoming records so as to not overwrite the BRAM contents. The $busy_o$ signal is set to high until all records have been emitted. The output process consists of traversing the bucket BRAM and following their linked lists, setting the visited bit of each record BRAM entry to 1 when emitted. There can be collisions when two field values produce the same hash and are therefore placed in the same bucket, even though the records do not belong to the same group. When following the linked list of a bucket and a collision is detected, then the head of the bucket will be set to the address of the colliding entry and the colliding record will not be emitted (and its visited bit will not be set to 1). If no collisions occur, the bucket head address is set to 0, indicating that there are no more records to output in that bucket. When a collision happened, it will not be 0 and therefore the linked list will be traversed again. This process continues until no more collisions occurred and thus all groups have been emitted. This happens for each entry in the bucket BRAM until all records have been emitted. Subsequently, the eow_o signal is driven high to signal that the window has ended and $busy_o$ is driven low to receive the next records.

This operator is implemented as a finite state machine to account for the different states of processing it can be in. On the hardware level, the current state is stored in a register. The logic of a given state is active when this register is set to the respective state. States are transitioned between by updating the state register.

Another approach towards hash-map-like implementations in hardware is content-addressable memory (CAM). CAM realizes associative storage in which entries can be looked up by a prefix (thus content addressable). It is used in networking devices for fast routing table lookups[13]. For FPGA designs, CAM has to be synthesized using LUTs and cannot make use of BRAM blocks. Realizing large amounts of memory is therefore not possible.

Performance The throughput and latency of the operator greatly depend on the input stream and window size. Only after all records of a window have arrived, the groups can be emitted. The latency of the operator is therefore the number of clock cycles it takes to reorder the records. The throughput of the operator is also dependent on the number of elements that fit into the bucket BRAM since it is being traversed when emitting the grouped records. A detailed analysis is deferred to Section 5.1.3. The analysis puts numbers to latency and throughput for different operator configurations and stream characteristics.

3.2.6 Group delimiter

If operations are to be performed on record groups, the start and end of these groups need to be represented within the stream. Given an input stream of grouped records (e.g., from the grouping operator), the group delimiter operator marks the group bounds by setting an ”end-of-group” (*eog*) signal to high once a group is over, a concept borrowed from windowing. The operator’s functionality is decoupled from the grouping operator itself because a stream of grouped records can also be obtained using sorting, or the stream could already consist of grouped records. Figure 9 shows an exemplary output stream of the operator.

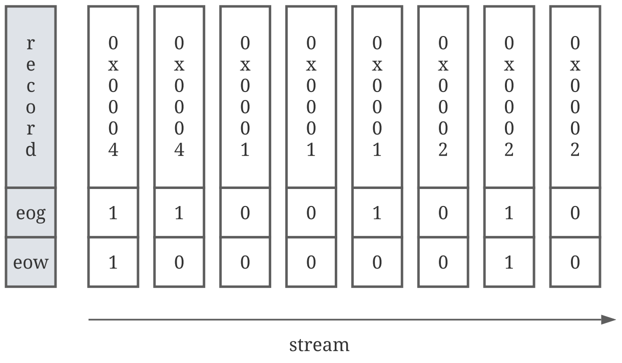


Figure 9: Exemplary output stream of the group delimiter operator.

Interface The operator receives records using R_i and dv_i, outputs records using R_o and dv_o, and signals the end of windows using eow_i and eow_o. The eog_o signal is set high for one clock cycle when the last record belonging to a group has been emitted. This can be in the same clock cycle as when outputting the last record of the group, or any time before the first record of the next group is emitted. The field after which the records are grouped by is determined by the $field\_start$ and $field\_end$ configuration signals.

Implementation The operator has to set the eog_o signal to high when it emits the last record of a group. But it only knows a record is the last one of the group when it has seen the next record. It will therefore store the incoming record and only emits it when the next record has arrived. It determines whether two records

Processing interf.		**Configuration interface**	**System interface**
Inputs	**Outputs**		
R_i	R_o	$field(\_start, \_end)$	clk
dv_i	dv_o		rst
eow_i	eow_o		
	eog_o		

Table 7: Interface of the group delimiter operator.

belong to the same group by comparing their respective fields, which have been extracted using a slicer based on the configuration. One can see in Fig. 9 that the end of a window is automatically the end of a group. In this case, the operator will output the record right away and not wait for the next one.

Performance The operator's throughput is one record per clock cycle as it can accept a record all the time. Its latency is two clock cycles because it stores the last received record and only emits it when the next record (or end-of-window) has arrived.

3.2.7 Aggregation

The aggregation operator allows the calculation of aggregates on a collection of records. It does that based on an aggregation function and a given record field. This work has so far introduced two record collections: windows and record groups. Both are represented using a delimiter signal (*eow* and *eog* respectively). By tying onto that mechanism, the aggregation operator can be used on windows as well as on groups.

Interface The operator uses a generic "end-of-collection" signal (eoc_i) to which an *eow* or *eog* signal can be wired to. Regardless of the eoc_i signal, it relays window information using the *eow* signals. For the functions *min*, *max*, and *median*, the operator output is the corresponding record and R_o is the same width as R_i. For all other functions, the output is only the aggregate. This aggregate (and therefore R_o has the width of the field which is aggregated on (except for *count* where it is a 64-bit unsigned integer). These different widths of R_i and R_o only apply in a non-configurable system. Since the function and record field can change in a configurable system, R_o has the same width as R_i there.

Only outputting the aggregate is different from Glacier [9], where the aggregation operator places the aggregation result into a field of the input record. In a configurable system where the record size is limited, this has the disadvantage that the record needs to have spare space to accommodate the aggregation result. For aggregations that produce large results or multiple aggregations that are performed in sequence, such an approach is unfeasible as it limits processing capabilities. In the system described here, the record is carried on a separate stream, and aggregates are merged back at a later stage. This process is described in when sample queries are presented in Section 3.4.

To calculate a *median* aggregation, the operator expects a sorted input stream and the $collection\_size_i$ signal to be set. This signal is part of the output of the sorting operator, allowing to chain an aggregation operator directly after a sorting operator to calculate median aggregates. It is irrelevant for all other aggregation functions.

Processing interf.		**Configuration interface**	**System interface**
Inputs	**Outputs**		
R_i	R_o	$function \in \{count, min,$	clk
dv_i	dv_o	$max, avg, median, sum\}$	rst
eoc_i	eow_o	$field(\_start, \_end)$	
eow_i			
$collection\_size_i$			

Table 8: Interface of the aggregation operator.

Implementation The aggregation operator calculates aggregates incrementally as records arrive, by storing an intermediate aggregate in a register. When the collection has ended, indicated by eoc_i, it emits the aggregate and resets the register. Each time a new record arrives, it updates the intermediate aggregate (e.g., for *sum* it adds the current record to the intermediate sum). By additionally counting the number of elements in the collection, all aggregation functions from Table 8 can be realized, except for *median*.

For median aggregation, the operator expects an input sorted by the record $field$ to aggregate on. Furthermore, the $collection\_size_i$ signal has to be set to the number of elements belonging to the collection currently being processed. This allows the operator to know the element count of the collection before it has received all its records, and therefore the index of the median record. This signal is part of the output of the sorting operator to allow for placing a sorting operator directly upstream of an aggregation operator to realize median aggregation.

In a configurable system, the aggregation function can change and therefore the logic to realize all functions need to be present. In a non-configurable system, the synthesizer will retain only the logic necessary to implement a single function.

Performance The throughput of the operator is one record per clock cycle since it can accept a record in each clock cycle. The latency naturally depends on the window size, since the first record of the window only has an effect on the output stream (aggregate) once all elements of that window have arrived. More interesting is the latency when the last record of the window is received. It then takes the operator only one clock cycle to calculate and output the aggregate since the aggregate is calculated incrementally.

3.2.8 Sorting

Sorting the records of a window can be useful for grouping or for realizing a median aggregation. Two successive sorting operators also allow performing grouping on multiple fields of the record which are not next to each other. This is not achievable using the grouping operator since it accepts only a contiguous range as the field to group by. The group delimiter operator can then be used downstream of the sort operator to ingest the end-of-group signal into the stream.

Interface Similar to the grouping operator, the sort operator does not output records in the order in which they arrive. It also has to wait until it has seen the last element of a window, as indicated by eow_i, to perform the sorting process. When the sorted records have been emitted, the end of the window is also signaled using the eow_o signal. The $busy_o$ signal communicates upstream that the operator cannot accept any more incoming records. During the sorting phase, this signal will be driven high. The field after which to sort by is determined through the $field\_start$ and $field\_end$ signals in the configuration interface. The sort order is given through the *order* configuration signal. The $collection\_size_o$ signal indicates the number of elements in the current window. It allows to place a median aggregation directly downstream to the sort operator since the aggregation needs to know the collection size to extract the median element from it.

Implementation There exists a plethora of sort algorithms that could be implemented in hardware. For this work, the focus was on low memory requirements,

Processing interf.		**Configuration interface**	**System interface**
Inputs	**Outputs**		
R_i	R_o	$field(\_start, \_end)$	clk
dv_i	dv_o	$order \in \{asc, desc\}$	rst
eow_i	eow_o		
	$busy_o$		
	$collection\_size_o$		

Table 9: Interface of the sort operator.

since memory is a scarce resource on FPGAs. Furthermore, the sort algorithm should be able to make use of the parallelism that custom hardware can provide.

The algorithm implemented is an optimized insertion sort since it sorts in-place. The optimization lies in a bitonic sorting network to create already sorted parts of the collection. A bitonic sorting network is combinational circuitry that sorts a fixed number of elements. The logic resource usage of a sorting network grows quadratically[10], so one cannot create a sorting network to sort an entire window for instance.

The sort algorithm would queue up incoming records in batches of 16 and then use a bitonic sorting network to sort these 16 records combinationally. The sorted batch is then stored in BRAM. When all records of a window have arrived, the BRAM contains sorted sub-collections. The operator then iterates of the heads of all sub-collections, finding the lowest element and emitting it. This process is repeated until all records have been emitted. One has to note here that no actual BRAM contents are changed during this output process. The only thing modified are the addresses of the sub-collection heads. This is possible since it is not required to have the entire sorted window in BRAM at the end of the sorting process because, in a streaming system, the operator can simply emit records in the correct order one by one.

Performance As for the grouping operator, throughput and latency depend on the input stream characteristics. Unlike the grouping operator, the latency is not dependent on the maximum number of records in a window but only on its algorithmic complexity of $(\frac{records}{2})^2$. A detailed analysis and comparison with the grouping operator are deferred to Section 5.1.3.

3.2.9 Stream management operators

The operators presented in this section do not serve data processing purposes but are utilities for handling streams.

Stream fork The stream fork operator duplicates its input stream by splitting the wires of the input record and other stream signals (*dv*, *eow*, *eog*) into two. This process is combinational.

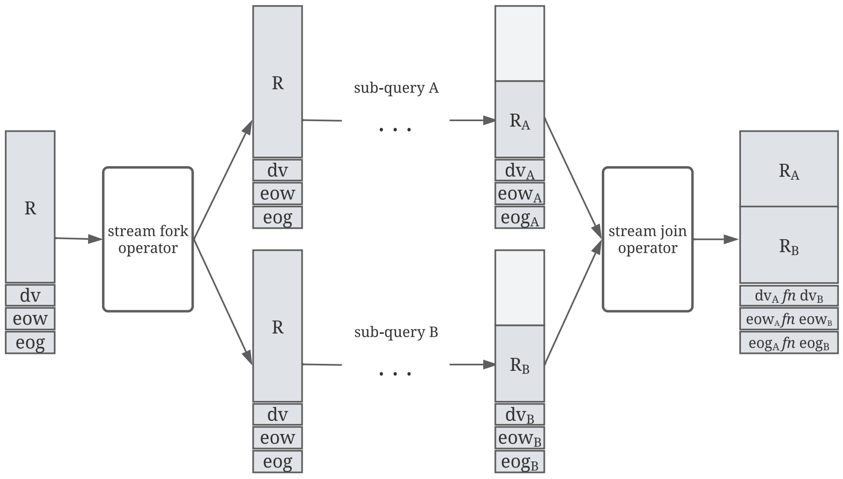


Figure 10: Signal layout between the stream fork and stream join operators. The fork operator creates two identical streams that are processed in sub-queries of which the lower half of the output is merged by the join operator into the output stream. fn denotes the $join\_function$.

Stream join The stream join operator does the opposite of the stream fork but is also combinational. It combines two input streams into one without much semantics (in contrast to a relational join). To achieve this, the input wire pairs cannot be simply merged with each other like the stream fork operator splits them. Given a configurable system that can handle a maximum record width of N, because it has N wires to propagate records between operators: while the stream fork can simply create two streams each of width N, the stream join cannot merge them into a stream of width $2N$, because it would violate the maximum record width limit. Without creating operators that can handle and propagate records of different widths, one has to perform a merge where parts of the input are discarded.

Processing interf.		**Configuration interface**	**System interface**
Inputs	**Outputs**		
R_i	R_o	$join\_function$	
dv_i	dv_o		
eow_i	eow_o		
eog_i	eog_o		

Table 10: Interface of the stream join operator.

Figure 10 shows the signal layout when splitting and merging streams. The stream join operator will take only the lower half of the record wires R_A and R_B to form the record on its output stream. The other signals are determined using the $join\_function$ configuration signal. This signal is four bits wide and can therefore represent all 16 binary boolean functions. The bits represent the output given a boolean assignment, i.e., the bits "*abcd*" stand for the boolean assignments "11", "10", "01" and "00" respectively. The logical *and* function would therefore be represented using the four bits "1000", *or* would be "1110" and *xor* would be "0110". Given the logical *and* function for instance, the stream join operator will only produce an output record when both of its input streams carry a valid record. The same goes for the *eow* and *eog* signals, which are merged using the same function. This flexibility allows for combining multiple parallel selection operators to form complex boolean expressions.

Stream delay The stream delay operator is used to synchronize data dependencies in streams that are processed in parallel. It increases the latency of a record and associated *eow* and *eog* signals by a predefined number of clock cycles. With a delay of n the record is propagated n clock cycles after it has been received. In hardware, this is realized by inserting records into a queue made out of registers and shifting the queue contents forward in each clock cycle.

The necessary delay is inherently defined by the structure of the query. To synchronize a processing pipeline with a latency of 5 with another pipeline with a latency of 2, an operator with a delay of 3 has to be added to the latter. In a configurable system where the query structure does not change, the delay will therefore also not change, requiring no configurability of this operator. In a composable system where the query structure can change, the highest flexibility is achieved by only providing stream delay operators with a delay of one clock cycle, because these can then be chained to create variable delays. Because of these reasons, there is no need to allow the delay of this operator to be reconfigured at run-time.

Stream buffer As already lined out in Section 3.1.3, stream buffers are used for local buffering. They are used upstream to operators that cannot accept records in each clock cycle. These operators signal that they are busy using an $busy_o$ signal. In normal operation, the stream buffer will just pass through incoming records, but when this busy signal is high, then the stream buffer will stop passing through records and will queue them up. If the busy signal falls low again, then the buffer will output these queued-up records and continue passing records through while maintaining record order.

The queue is implemented using BRAM. Since BRAM cannot be dynamically allocated at run-time, there is no re-configuration possible for this operator and the BRAM size has to be defined at synthesis time. In practice, this is not a large problem, since global buffering will come into effect once the allocated BRAM is full. The size of stream buffers only determines the capability to alleviate small spikes in record frequency.

The throughput of the buffer is one record per clock cycle, while the latency is three clock cycles. It constitutes of one cycle for writing the record, one cycle for setting the read address, and one cycle for reading and outputting the record. There is no latency difference between queuing up and passing-through records.

Stream metadata A stream consists of data signals (the record R) and control signals (dv, eow, eog). For multi-query processing (see Section 3.4.3) it is necessary to make control signals part of the data signal. The metadata operator allows placing the values of these signals onto the record wires. Which byte of the record to place them is defined through the configuration signals that can be seen in Table 11. This process is completely combinational and makes the operator not require a clock or reset signal.

Processing interf.		**Configuration interface**	**System interface**
Inputs	**Outputs**		
R_i	R_o	*dv_byte*	
dv_i	dv_o	*eow_byte*	
eow_i	eow_o	*eog_byte*	
eog_i	eog_o		

Table 11: Interface of the stream metadata operator.

The operator drives dv_o always high, i.e., the operator produces an output record each clock cycle, even when there was no input record. Otherwise, the operator would only output records where the value of the byte denoted by *dv_byte* is 1. If there is no input record, then this information is represented through a value of 0 in the byte denoted by *dv_byte*. The signals eow_i and eog_i on the other hand can be passed through. Figure 11 shows how its output signals are set based on its input signals.

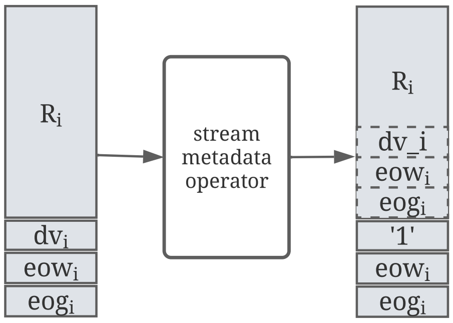


Figure 11: Output of the stream metadata operator based on its input signals.

3.3 Run-time reconfiguration

Configurable operators have a set of input signals whose values represent the configuration of the operator. The configuration is not stored within the record, so a valid configuration has to be present on these signals in each clock cycle. This can be achieved by storing the configuration in registers and wiring their outputs directly to the configuration signals. The configuration can then be updated by writing new values to the registers. Figure 12 shows the configuration register for an exemplary selection operator.

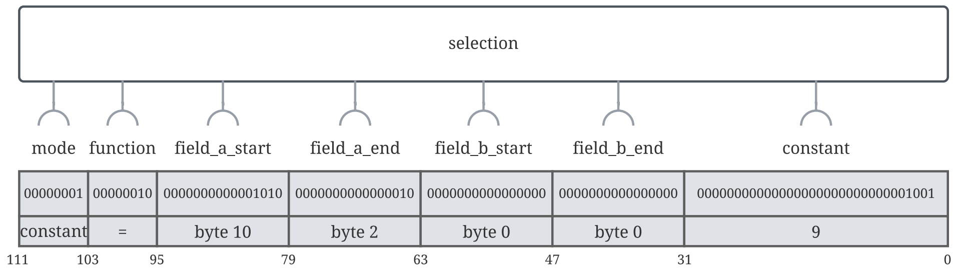


Figure 12: Configuration register value for selection that compares the field consisting of bytes 10 (excl.) down to byte 2 for equality with the constant 9.

The collection of operator configurations forms the configuration of the query. To reconfigure the query and therefore all operators, the query interface is extended by a configuration signal C with width n. This width equals the width of all operator configurations combined. By driving the *cv* ("configuration valid") signal high, the configuration registers are updated and all operators can process based on the new configuration in the following clock cycle. This leads to an overall theoretical reconfiguration latency of one clock cycle. How the configuration is received from a host is described in Section 4.

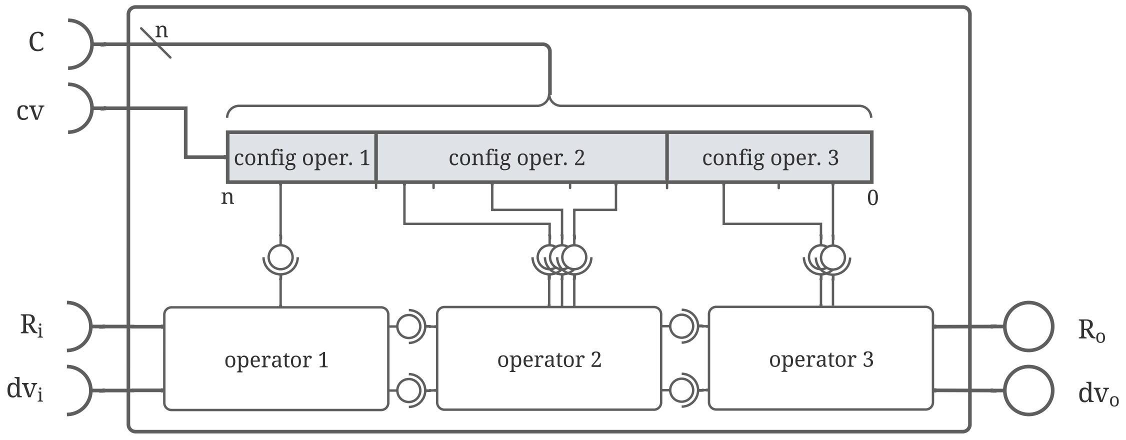


Figure 13: Operator configuration stored in registers (grey) which are wired to the individual operator configuration signals.

One can see in Fig. 13, which shows how parts of the configuration register are wired to operators, that the query configuration is tightly coupled with the query structure. When setting a configuration, one has to make sure that the individual operator configurations are in the correct order. Furthermore, in the presented design, all operator configurations are always updated together and at once. Theoretically, it would also be possible to only reconfigure a single operator (or even a single configuration signal) which would reduce the amount of configuration information that has to be received from the host. But this would require additional control information as to which part of the query configuration register should be written, requiring additional logic resources. Since the query configuration register can be very wide (thousands of bits wide in large queries), the associated signal routing logic would be very complex.

3.4 Sample queries

This section presents queries and describes how they can be realized through the combination of operators.

3.4.1 Timestamp-based tumbling window word count

This query takes a stream of records consisting of an 8B timestamp in microseconds and a 32B word as input. It creates tumbling windows of 1 second in length and counts the number of occurrences of each word in each window. It outputs a record for each word of the window that includes the word itself and how often it occurred.

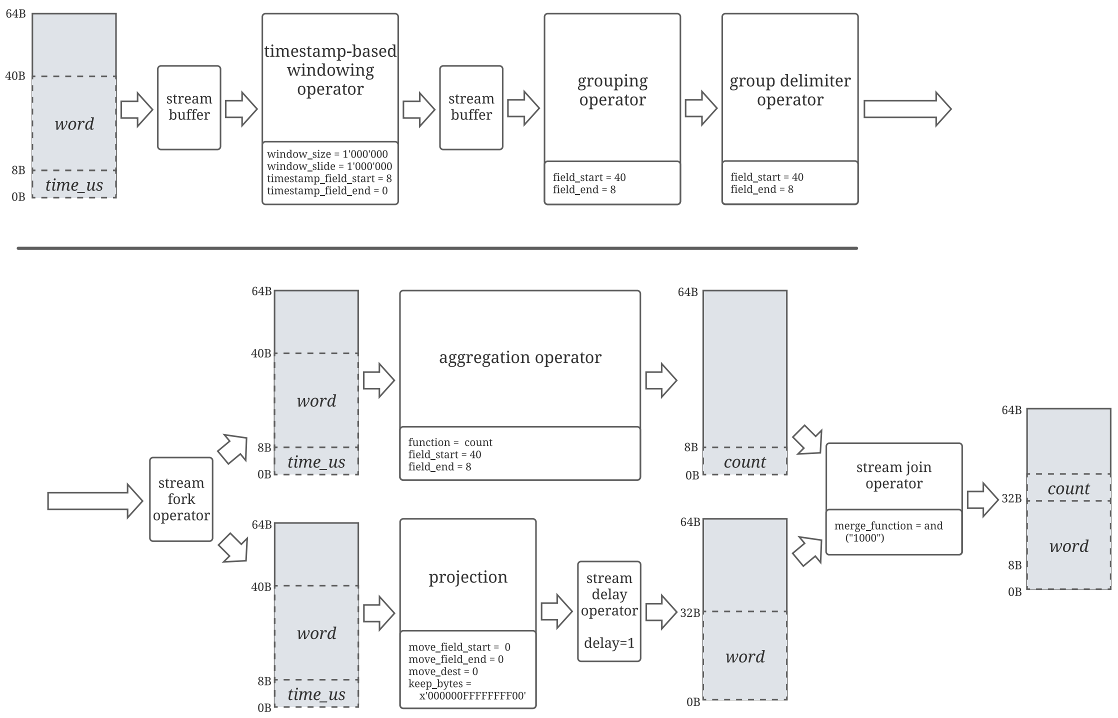


Figure 14: Timestamp-based tumbling window word count query. White boxes represent operators and grey boxes indicate the record layout.

The processing stages are as follows and are visualized in Fig. 14:

1. The first operator in the query is a timestamp-based windowing operator with an upstream buffer because the operator can be in a state where it cannot accept incoming records, requiring them to be buffered. The windowing operator then emits the end-of-window signal according to the record timestamp.

2. The subsequent operator is a grouping operator, also with an upstream buffer. The grouping operator reorders the records in the window such that

they form consecutive groups in which the *word* is the same.

3. The group delimiter operator adds the end-of-group signal to the stream that delimits these consecutive groups of records.

4. The stream is now split using a stream fork operator into two parallel processing branches.

   (a) The first processing branch performs a count aggregation based on the end-of-group signal to count the records within each group. Since each group consists of records of the same *word*, it will output the number of occurrences of each word.

   (b) The second processing branch uses a projection operator to only retain the *word* field in the record. The aggregation operator has a latency of one clock cycle, while the projection operator is combinational ('zero' latency), so an additional delay of one clock cycle has to be added to this processing branch to synchronize them.

5. The two processing branches are merged by a stream join operator using the logical *and* function that makes it only output a record when both input branches carry a record. While the first processing branch emits the aggregate, the second processing branch emits the *word* identifying the group. The join operator will then output a single record for the group which contains the result of the first processing branch (aggregate) and second processing branch (word). If these are the last elements of the window, it will also forward this end-of-window signal to not lose the association of groups to windows.

3.4.2 Fixed-size sliding window sum aggregation

This query takes a stream of trades that consist of a trade id and the profit or loss ("PnL") as input. It creates sliding windows of size 10 and slide 1 to calculate a rolling sum of the profit. If this PnL sum is greater than 1,000 or is lower than -1,000 within a window, the query outputs a record that contains this cumulative PnL, the trade with the lowest PnL within the window, and the trade with the highest PnL within the window.

The processing stages are as follows and are visualized in Fig. 15:

1. The first operator is the fixed-size sliding window operator that assigns

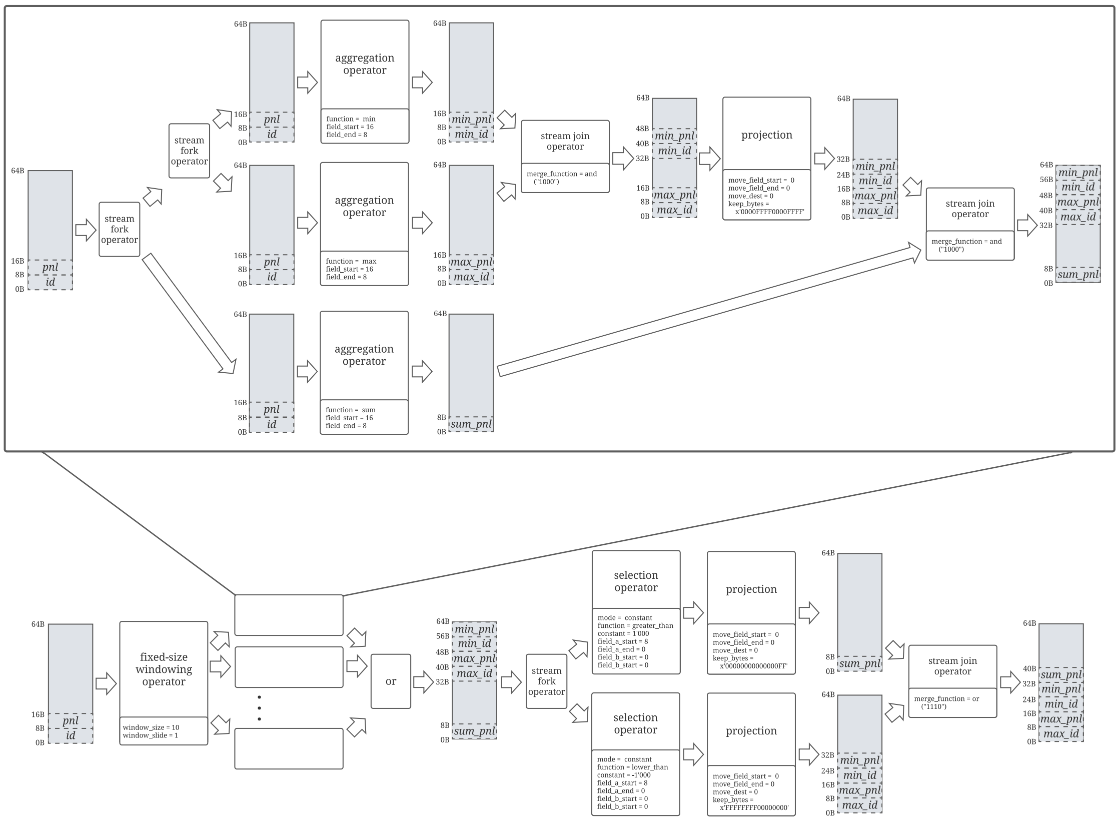


Figure 15: Cumulative PnL outlier detection query. White boxes represent operators and grey boxes indicate the record layout.

records to 100 parallel processing branches and emits the end-of-window signal accordingly.

2. These processing branches all contain the same logic.

   (a) First, two stream fork operators separate the stream into three parallel processing branches.

      i. The first branch uses an aggregation operator based on the end-of-window signal to extract the record with the lowest PnL within the window.

      ii. The second branch uses an aggregation operator based on the end-of-window signal to extract the record with the highest PnL within the window.

      iii. The third branch uses an aggregation operator based on the end-of-window signal to calculate the sum of the PnLs of the records within the window.

   (b) The first two branches are then merged using a stream join operator based on the logical *and* function since both branches always carry a record at the same time.
   (c) A subsequent projection operator compresses the join result.
   (d) The third processing branch is now merged with the first two, also based on the logical *and* join function. All three processing branches have the same latency of one clock cycle. The resulting record now consists of the trade with minimum PnL, trade with maximum PnL, and the sum of PnLs.

3. Each time the windowing operator ingests a record into any of the processing branches, exactly one window will be finished. That means that no multiple processing branches will ever output a record at the same time, making window synchronization needless. So all outputs of the branches can just be *or*-ed together.

4. A stream fork will now create two parallel processing branches.
   (a) The first branch uses a selection operator to only retain records for which the PnL sum is greater than 1,000. A subsequent projection operator prepares the record for merging by only retaining a single field that would otherwise be discarded by the stream join.
   (b) The second branch uses a selection operator to only retain records for which the PnL sum is lower than -1,000. A subsequent projection operator prepares the record for merging by compressing its field layout and omitting the field output by the first branch.

5. A stream join operator now merges both branches. Since the requirement is to emit records where the PnL is greater than 1,000 *or* lower than -1,000, the merge happens based on the logical *or* function.

3.4.3 Multi-query system

The presented operators are also capable of producing a system that processes multiple queries at once. There are two concepts that differ in how records are ingested into these multiple queries. In a record-concurrent multi-query system only one record, which can belong to either of the queries, is ingested in one clock cycle. The record contains a field indicating which query it belongs to. For record-parallel multi-query processing, multiple records can be ingested in the same clock cycle.

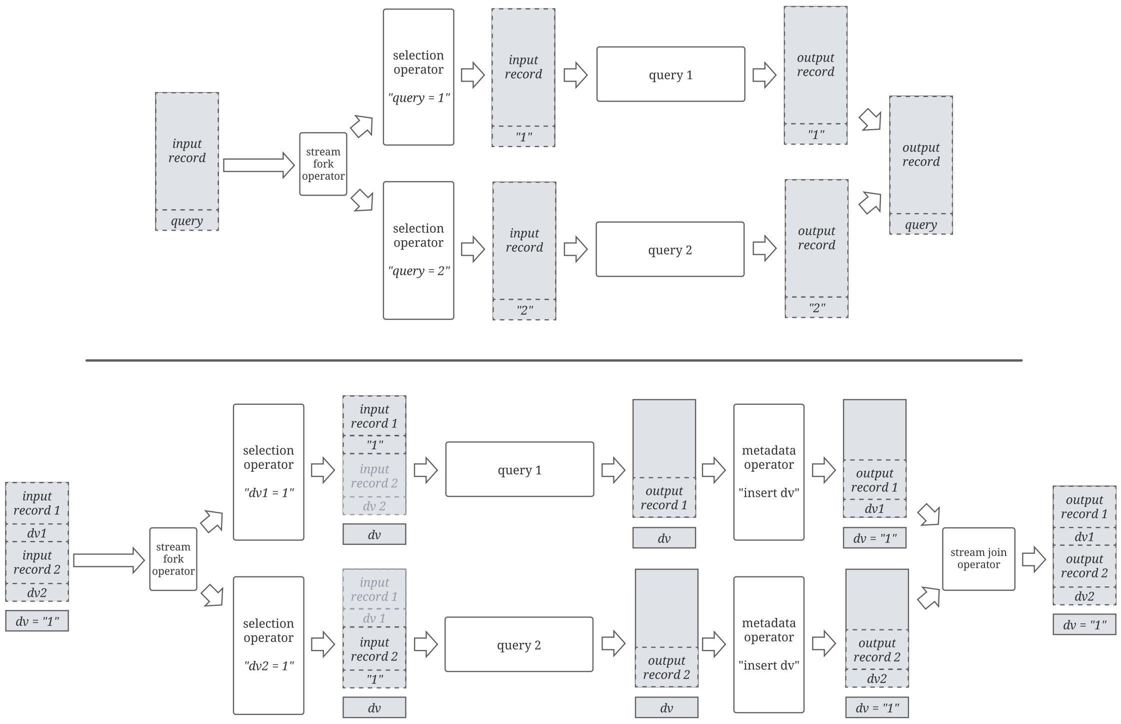


Figure 16: Comparison of record-concurrent (above) and record-parallel (below) multi-query approaches with two queries.

Figure 16 shows the architecture of both multi-query approaches. In a record-concurrent multi-query, the record contains an additional field that indicates which query it belongs to. The stream is then split by a stream fork operator into two identical streams. A selection operator now filters records based on this additional field to only pass the correct records into the queries. The output records of the queries can simply be merged into the output stream with the additional field again indicating which query this output record belongs to.

In a record-parallel multi-query system multiple records are ingested at the same time. That means that the record signal has to be split up to accommodate two smaller records. Since there is only one data valid signal but two records, a per-stream data valid signal is transmitted within the record and the ”global” data valid signal is always high, because now the per-stream data valid signals are used to indicate whether the record data is valid. A stream fork operator again splits the stream up into two identical sub-streams. Each stream now filters based on the per-stream data valid signals, e.g., only when the $dv1$ is high, query 1 will process the record. This essentially means that the $dv1$ and dv signals of the sub-stream now have the same value. The query then outputs an output record and associated data valid signal. A stream metadata operator now allows placing this data valid signal back into the record to obtain a per-stream data valid signal again. A stream join operator then merges the records of both streams into one output stream, now containing both records again. One can see that the output record has the same signal layout as the input record.

Both approaches seem to be equivalent in the amount of data they process. While a record-concurrent system processes one full-size record per clock cycle, a record-parallel system processes two half-size records per clock cycle. When having a closer look, the amount of data processable with a record-parallel system is lower than that of a record-concurrent system since part of the record signal has to be reserved for the per-stream data valid signals.

It has to be noted that these multi-query approaches are just examples. While these examples realize two sub-queries, by using more stream forks and joins, one can obtain more sub-queries. Furthermore, there does not have to be an exclusive query selection. It is also possible to design a system in which a record is processed by multiple queries instead of just one. One has to be cautious here to not exceed maximum query throughput since multiple output records would be generated from one input record.

3.5 Query composition

This work has so far explained how configurable operators can be implemented and how they can be configured. This allows for creating configurable queries, i.e., queries that can be reconfigured at run-time, but that have an immutable query structure. This section will show how the next step of configurability can be achieved in which also the query structure can be changed at run-time through reconfiguration - a composable query.

3.5.1 Operator interconnect

An operator has a certain set of input signals and output signals. In a non-composable system, the output wires of an operator are hard-wired to its downstream operator. For a composable query, this wiring needs to be made dynamic in a way that allows communication from one operator to any other operator. An operator interconnect is introduced that allows for n:n communication paths instead of doing 1:1 connections between operators. This interconnect is essentially a signal matrix that allows routing signals from one operator to all other operators. This also has the advantage that an operator can have multiple downstream and upstream operators, allowing for flexible communication schemes. There is one such signal matrix for *dv*, *eow*, *eog*, and *R* (which is technically a signal cubus because it is a matrix of logic vectors). A contrasting design approach to an interconnect is to have multiple fixed "query templates" into which operators can be "plugged", as presented by Najafi. This comes at the cost of less flexibility but higher optimization potential.

Operators are addressed by unique IDs that correspond to indices in the signal matrices. If operator 1 wants to send the *eow* signal to operator 3, it would set `eow_matrix(1)(3) <= '1'`. Operator 3 receives signals from all operators on the orthogonal axis of the matrix `eow_matrix(1..n)(3)`. One can see that an operator would technically be able to output to all other operators simultaneously. That means that a downstream operator has to decide how to interpret all incoming signals. Using a logical *or* gate to, e.g., combine all record input signals, is not feasible, since the validity of the record signal is tied to the *dv* signal. Instead one can take the record signal at the index where there is a logic high in the *dv*-matrix.

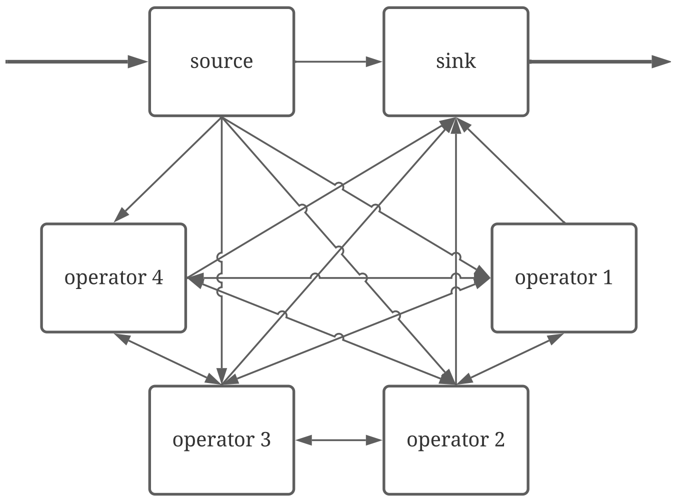


Figure 17: Interconnect schematic including source and sink nodes. Each arrow represents signal wires for *R*, *dv*, *eow* and *eog*.

Figure 17 shows the schematic of signal routes between operators. In addition to the operators, there is also one source and one sink node. The source node receives the query input stream and has an outgoing route to all other operators. The sink node only has incoming signals from other operators and relays them onto the query output. The source is associated with id 0, while the sink has the id corresponding to the last index in the signal matrices. For *n* operators, the signal matrices will therefore be of size $n + 2 \times n + 2$, because they include source and sink.

3.5.2 Interconnect configuration

In a composable query, each operator not only has its operator configuration but also an interconnect configuration that essentially determines the query operator graph. This configuration includes information as to which operators to receive data from and to which operators to output data, with the operators being identified by their IDs. Depending on the kind of operator, this configuration can differ. Figure 18 shows a simple 1:1 operator interconnection on the left and a more complex n:m interconnection on the right of the figure. Operators with a 1:1 interconnection are all operators only having one input and one output stream. The windowing operator is 1:n, the stream fork operator is 1:2, the stream join operator is 2:1, and the window synchronizer is n:1. The interconnect input and output switching is implemented using multiplexers that determine which of the input wires to receive data from and which of the output wires to put data on.

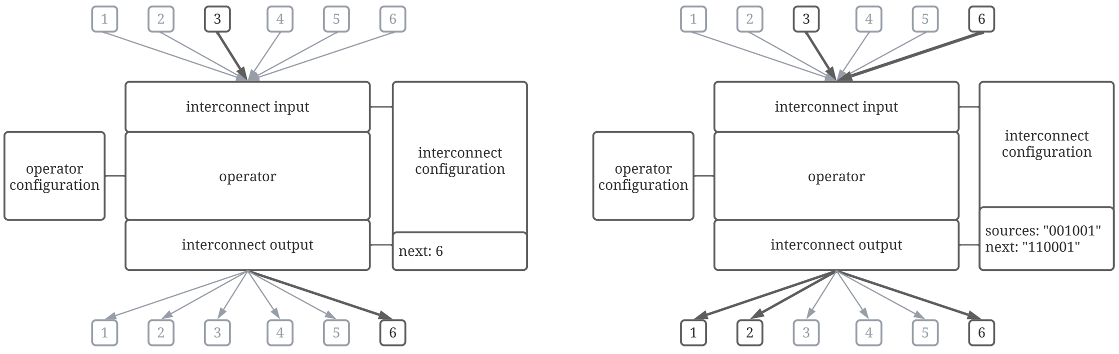


Figure 18: 1:1 operator interconnect with configuration (left) and n:m operator interconnect with configuration (right).

3.5.3 Operator provisioning

Composable operators allow for restructuring queries at run-time, but the operators to be incorporated need to be present in the FPGA design. This raises the interesting question which operators should be provisioned and in which quantity. This work will not answer this question in detail, since more research is necessary to investigate the operator requirements of common streaming tasks. But as a general rule, it makes sense to fully utilize the available logic resources on the

FPGA with composable operators. Furthermore, many streaming tasks will not require more than one windowing operation, but as Section 3.1.2 has shown, will heavily rely on stream management operators and record manipulation operators.

Section 3.3 has introduced the way a query is configured. As it is always configured in its entirety, for a composable query, all present operators have to be configured, whether used in the query or not. This is necessary to route their eventually undefined output signals away from operators that belong to the query. The larger the number of composable operators is, the larger the query configuration is, which has an impact on reconfiguration time. Figure A.1 shows VHDL code that resembles the configuration of a composable system.

4 System integration

This section outlines how the system presented in the previous section can be interfaced with a host to receive records and configurations. Furthermore, the practical approach towards query compilation is described.

The system presented in Section 3.1.2 assumed the stream source and sink not to be part of the query, resulting in a record-based interface. Section 3.1.3 described how global buffering can be achieved using a query-global "busy" signal and Section 3.3 extended the interface with input signals for reconfiguration. A holistic view of the queries' interface is shown in Fig. 19.

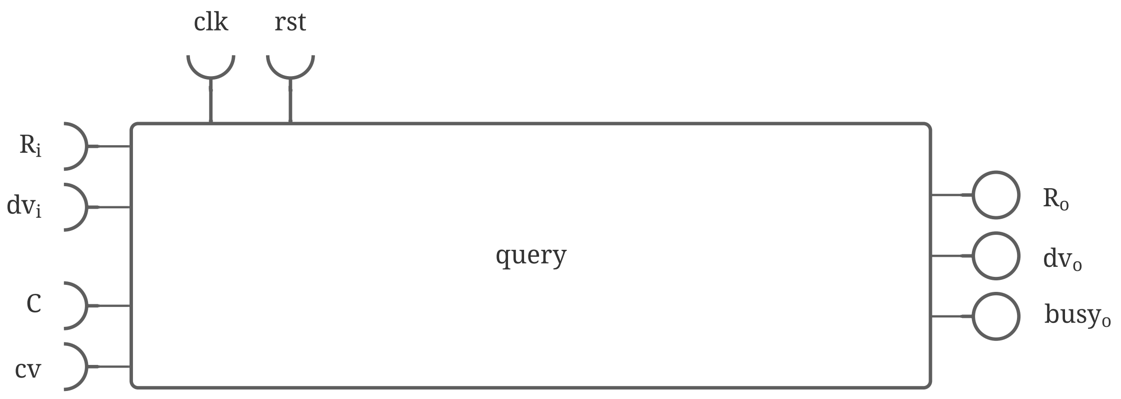


Figure 19: Holistic view of the interface of the query.

4.1 AXI-Stream interface

To make the system universally usable, the interface of the query is translated into the AXI4-Stream protocol[2] which has been developed by ARM and is used for transporting data streams. The protocol requires three signals for each the primary and secondary side:

- **DATA**: The data bus of n bits (user-defined) width. Output of a primary and input to a secondary.
- **VALID**: Single wire that indicates whether the *DATA* wires contain valid data. Output of a primary and input to a secondary.

[2] `https://developer.arm.com/documentation/ihi0051/a`

- **READY**: Single wire that indicates whether data can be received. Input to a primary and output of a secondary.

One can see parallels between these protocol signals and the R, dv and $busy$ signals used for communication between operators in the query. The $DATA$ signals can propagate n bits in each clock cycle, while the $VALID$ and $READY$ signals are used for flow control. When having a composable query, it makes sense to set the maximum record size to the width of the $DATA$ signal since this is the amount of data that can be received in a single clock cycle. The actual width of the $DATA$ signal is usually constrained by the hardware interface to the host.

4.2 Stream protocol

The AXI-Stream only represents one data stream, which can be used to transport records, but the query also expects the configuration as an input. A stream protocol can be employed that allows to transport records as well as configuration information within a single AXI-Stream. A simple approach would be to annotate each piece of data transmitted in the stream with an annotation of whether it represents a record or a (part) of a configuration. Given that the configuration usually only changes very infrequently in comparison to the stream of records, this would incur an overhead of one ”control byte” per clock cycle. A more efficient solution is using a stateful protocol where messages determine the state the protocol is in, e.g, receiving records or receiving a configuration. This also makes it possible to define more message types such as a ”query reset” signal that allows to reset the state of the query, e.g., after a reconfiguration. Each such message has exactly the width of the $DATA$ signal and one message can be received per clock cycle. A ”protocol interpreter” component is introduced that has an AXI-Stream $DATA$ and $VALID$ signal as input and has a query reset signal rst, record signal R, record valid signal dv, configuration signal C and configuration valid signal cv as outputs towards the query.

The state machine of the protocol is depicted in Fig. 20. In the initial ”control” state, either the ”records control” (”r” byte), ”configuration” (”c” byte) or ”reset” (”0” byte) message are expected next. If the ”reset” message is received, the rst output signal is held up for one clock cycle and the protocol goes back into the ”control” state. If the ”records control” message is received, the next expected message is supposed to include the number of records to receive. This number is stored in the num variable. The protocol will then be in the ”records” state as long as there are still records left that have been announced. Each time a

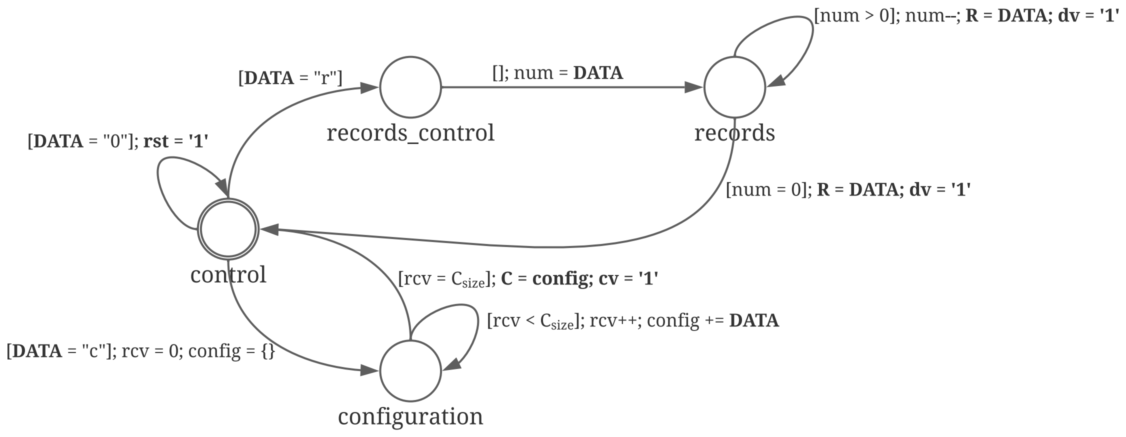

[num > 0]; num--; R = DATA; dv = '1'
[DATA = "r"]
[]; num = DATA
records_control
records
[DATA = "0"]; rst = '1'
[num = 0]; R = DATA; dv = '1'
control
[rcv = C_size]; C = config; cv = '1'
[rcv < C_size]; rcv++; config += DATA
[DATA = "c"]; rcv = 0; config = {}
configuration

Figure 20: State machine depicting the protocol for receiving records and configuration on the same stream. Bold variables represent signals. The $VALID$ signal has been omitted from transition conditions.

record is received, it is output using the R and dv signals. Once all num expected records have been received, the protocol goes back into the "control" state. If the next incoming message is a configuration message, the protocol stays in the "configuration" state until all parts of the configuration have been received. Since the configuration size C_{size} is static, there is no message necessary that indicates the configuration size. After all parts have been collected, the entire configuration is output using the C and cv signals.

When the configuration changes infrequently compared to the data stream and when larger batches of records are announced, the overhead of this protocol design is negligible. It is also extensible, as more messages can be added, similar to the reset message. A possible addition could be the ability to also send end-of-window signals from the host.

4.3 AXI-Stream-to-query

Figure 21 shows how the query and protocol interpreter are integrated into an AXI-Stream environment. The AXI secondary input signals $DATA_i$ and $VALID_i$ are used by the protocol interpreter to extract record, configuration and reset signals. These are then passed into the query. For global buffering (see Section 3.1.3), the query emits a query-global "busy" signal that indicates that the query is not able to accept any more incoming records. To halt the stream source (which is essentially the input AXI-Stream) from delivering more records, this busy signal is

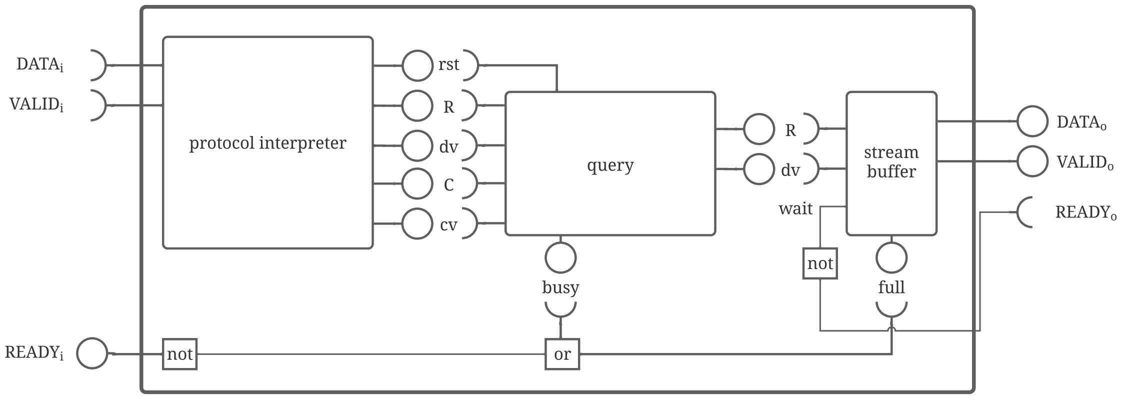


Figure 21: Integration of query and protocol interpreter into AXI-Stream system.

negated and wired to the $READY_i$ signal. If the busy signal is high, then the ready signal will be low, indicating that data delivery has to be halted. There is also the $READY_o$ signal that indicates whether the stream sink is able to accept data. If this signal is low this indicates that the system has to stop emitting records. The stream is buffered to alleviate small spikes in record frequency, but if this buffer becomes full, then the source is also instructed to stop delivering data using the $READY_i$ signal. This concept is the same as the combination of local buffering and global buffering described in Section 3.1.3.

4.4 SNAP host memory interface

The system now has an AXI-Stream input and output interface which makes it easy to integrate with other hardware components and frameworks. The stream processing system has been integrated into the IBM CAPI SNAP framework[2], which is a framework for building FPGA-accelerated, so-called, ”actions”. It allows cache-coherent direct host memory access using the IBM ”Coherent Accelerator Processor Interface” (CAPI) present in POWER hosts. The framework is geared towards software engineers as it allows to create actions using HLS but also RTL-languages like VHDL and Verilog[22]. It can be used with a variety of FPGAs of different vendors. The processing model is job-based, i.e., the user submits a job that includes the parameters for a given action which is then executed.

A SNAP action has been developed that reads records from host memory and writes them to an AXI-Stream. This AXI-Stream is input to the stream processing system, which in turn outputs an AXI-Stream of which the action writes records back into host memory. The overall system therefore allows stream processing from host memory into host memory. The SNAP framework version 1 allows to receive 64 bytes every clock cycle. Therefore in a composable system the maximum record size has been set to that number. The parameters that the action gets comprise of an input memory region, the number of bytes to read, and an output memory region. This allows for "true streaming" when setting the number of bytes to read to the size of one record. Since there is an overhead associated with starting an action and reading only a single record from memory, it is also possible to do batching by placing multiple records in the input memory region and selecting a higher number of bytes to read. The host memory access is then done in bursts, where data is requested in batches of 64 records. This results in better performance, as outlined in Section 5.3.

4.5 Query generation

Query generation is the process of obtaining an FPGA design from a query description. A query description can be an operator graph and a design could be obtained by translating that query description into VHDL code which can then be synthesized. The composable query presented in Section 3.5 has the interesting property of being completely defined by a query configuration. This query configuration is in fact just a query description. By combining this property with the synthesizers' ability to optimize unused logic resources away, one can generate queries of varying configurability. When hard-coding a configuration into the composable query, the synthesizer will essentially generate a non-configurable query. When only hard-coding the interconnect configuration, the query structure will be static, but the operators will still be configurable (i.e., a configurable query). This query generation process is entirely synthesizer-based and does not require to develop a tool that converts a query description into synthesizable VHDL code. The impact that these additional processing and optimization steps have on synthesis time is analyzed in Section 5.2.4.

5 System evaluation

This section evaluates the stream processing system presented in this work. The evaluation can be roughly separated into two parts: static analysis and execution analysis. The static analysis evaluates the system from an execution-less perspective, i.e., an analysis of the system and operator design in itself. Section 5.1 will present statically-inferred performance metrics of some of the queries that have been presented in Section 3.1.2, based on operator implementation and query structure. The impact that different levels of configurability have on logic usage, performance, and synthesis time is presented thereafter in Section 5.2. As part of the execution analysis, Section 5.3 will evaluate the host integration which has been described in Section 4 by presenting benchmark results.

5.1 Static analysis

This section will present how the performance properties of a query can be inferred from its operators. It will do this based on two example queries: the ”timestamp-based tumbling window word count” (see Section 3.4.1) and the ”fixed-size sliding window sum aggregation” query (see 3.4.2). These two queries have been chosen for this evaluation because they exhibit different structures. While the word count query has tumbling windows, the aggregation query has 10 parallel processing branches for its sliding window. On the other hand, the word count query has to group records, which is a bottleneck, as seen in Section 5.1.3. This does not apply to the aggregation query, but it has eleven times as many operators as the word count query.

Generally, the latency of a query is the latency of the longest processing path. The latency of a processing path is in turn the sum of the latencies of the operators within that path. Section 3.2 presented operators and their latencies. Sometimes the latency of an operator is dependent on stream characteristics, e.g., for the grouping operator. For that case one uses the worst-case latency of an operator to derive the worst-case latency of a query, which is also measured in clock cycles (C).

The throughput of a query is the minimum of the throughputs of all processing paths. The throughput of a processing path is the minimum of the throughputs of the operators in the path. Therefore the query throughput is determined by the ”slowest” operator. Query throughput is measured in records per clock cycle ($1/C$). Based on this, one can also infer actual execution properties. An FPGA clocked at 250MHz that runs a query with a throughput of 1, can process 250 million records per second (assuming that records can be ingested fast enough). With a query throughput of 0.1, the number of records per second is 25 million.

5.1.1 Fixed-size sliding window sum aggregation

To defer the latency and throughput of this query, one has to look at the individual operator metrics, which are shown in Table 12. The windowing operator creates 10 fixed-size windows, which all share the same logic and therefore the same performance characteristics. Within one window, three processing paths are created using two stream fork operators: Path 1a, 1b, and 2. Each of these paths has a latency of 1 clock cycle, which is caused by the aggregation operator in each. Since these paths are parallel, the window processing has an overall latency of one clock cycle. Apart from this window processing and the windowing operator, all other logic is combinational, i.e., having a (theoretical) latency of zero. The overall query thus has a latency of two clock cycles. Investigating throughput, all operators within the query have a throughput of one record per clock cycle. This means the overall query can also process one record per clock cycle.

5.1.2 Timestamp-based tumbling window word count

Table 13 shows individual operator latencies and throughputs for the word count query. In comparison to the fixed-size sliding window sum query, the operators exhibit other performance characteristics. The windowing operator has varying throughput because it has to stall when empty windows have to be emitted. The grouping operator has a varying latency and throughput because its complexity is dependent on the number of maximum records per window and the actual input stream of which the records have to be reordered. This section also includes a detailed performance analysis for this operator. This analysis assumes records to be equally distributed over the grouping operator's buckets and that windows always contain the maximum number of elements.

The windowing operator creates tumbling windows, so there is only one processing path and no parallel processing of windows. Within the processing of this single window, a stream fork creates two parallel processing paths, the first one including an aggregation and the second one including a projection and stream delay. The latency of this parallel processing is one clock cycle. The sole reason for the existence of the stream delay in the second path is to bring both parallel processing

Operator	Latency (C)	Throughput ($1/C$)
fixed-size windowing	1	1
stream fork	0	1
1: stream fork	0	1
a: aggregation	1	1
b: aggregation	1	1
1: stream join	0	1
1: projection	0	1
2: aggregation	1	1
stream join	0	1
or	0	1
stream fork	0	1
1: selection	0	1
1: projection	0	1
2: selection	0	1
2: projection	0	1
stream join	0	1
Query	2	1

Table 12: Individual operator latencies and throughputs for the fixed-size sliding window sum aggregation query described in Section 3.4.2. Indentations indicate parallel processing paths.

paths to the same latency to avoid data races. Adding up the latencies of all operators, the query has an overall latency of 10 clock cycles plus the latency of the grouping operator, which is variable and will be analyzed in detail in the upcoming section.

The throughput of the query is either limited by the windowing or grouping operator. If the probability that the windowing operator has to emit an empty window within a clock cycle ($P_{empty\ window}$) is greater than 8/9 ($\approx$ 88.89%), then the query bottleneck is the windowing operator, otherwise it is the grouping operator. The query cannot accept a record in each clock cycle and once the stream buffers become full, the query-global busy signal halts the source.

5.1.3 Grouping performance

In comparison to other operations, the process of grouping is computationally heavy and constitutes a bottleneck in query throughput. Therefore this section

Operator	Latency (C)	Throughput ($1/C$)
stream buffer	3	1
timestamp-based windowing	1	$1 - P_{empty\ window}$
stream buffer	3	1
grouping	$R_{max\ per\ window} * 9$	1/9
group delimiter	2	1
stream fork	0	1
1: aggregation	1	1
2: projection	0	1
2: stream delay (delay=1)	1	1
stream join	0	1
Query	$10 + R_{max\ per\ window} * 9$	$min(1 - P_{empty\ window}, 1/9)$

Table 13: Individual operator latencies and throughputs for the timestamp-based tumbling window word count query described in Section 3.4.1. Indentations indicate parallel processing paths. Assuming equally-distributed records and fully-utilized windows for the grouping operator.

will analyze the grouping operator in detail and compares it to the approach of using sorting instead of a hash map to create groups.

Grouping operator latency The grouping operator's performance depends on two variables: the maximum number of records per window $R_{max\ per\ window}$ and the actual number of records R_{window} within the window. This distinction is necessary since the operator traverses the bucket BRAM, of which the size is based on $R_{max\ per\ window}$, to emit groups. In the worst case, it has to traverse the entire BRAM, regardless of the actual number of records. The latency of accessing one entry in the bucket BRAM is two clock cycles: one cycle to set the BRAM address to read from and one cycle to read the result. Traversing the entire (empty) bucket BRAM therefore takes $2 * R_{max\ per\ window}$ clock cycles plus a negligible number of cycles for initialization. Writing a record takes two clock cycles. If there are multiple records in a bucket, it takes four clock cycles to process one record. When records are equally distributed across buckets, it takes seven clock cycles to read one record. Taking this worst-case scenario and assuming that the BRAM of the operator is fully utilized, the latency per record would be 2 (write) + 7 (read) = 9 clock cycles. The overall formula to approximate the operator's worst-case latency L is therefore $L \approx 2 * (R_{max\ per\ window} - R_{window}) + (2 + 7) * R_{window} = 2 * R_{max\ per\ window} + 7 * R_{window}$. As indicated, the actual latency depends on the distribution of records across buckets.

Grouping operator throughput The throughput of the grouping operator is the fraction of clock cycles it is able to accept records in. It takes two clock cycles to write a record to memory and it takes up to seven cycles to read and emit it again. That means the operator has a throughput of $\frac{1}{2}$ records per clock cycle in the write phase and zero in the read and emit phase because the operator cannot accept new records during that time. To approximate the overall throughput T, these phases need to be weighted by the fraction of total clock cycles they constitute: $T \approx \frac{1}{2} * \frac{2*R_{window}}{L} + \frac{1}{2} * 0 = \frac{1}{2} * \frac{2*R_{window}}{2*R_{max\ per\ window} + 7*R_{window}}$. When $R_{window} = R_{max\ per\ window}$, then $T \approx \frac{1}{9}$, or when $R_{window} = \frac{1}{4} * R_{max\ per\ window}$, then $T \approx \frac{1}{15}$, being able to accept a new record every 15 clock cycles on average.

Sort operator performance The sorting operator implements an optimized insertion sort algorithm of which the complexity is only based on the number of records in the window but not its maximum number of records. The latency L of the sorting algorithm is the number of clock cycles it takes to sort the records in the window, which is approximately $(\frac{R_{window}}{2})^2 = \frac{1}{4} * {R_{window}}^2$. The average throughput of the operator is the inverse of the number of clock cycles it takes to sort because the operator has to block incoming records during sorting.

Comparison Since grouping is dependent on the maximum number of records within a window, but sorting is not, it is worthwhile to investigate which approach has the best performance characteristics in which scenario. Both operator implementations have been simulated using random records and different values for R_{window} and $R_{max\ per\ window}$, of which the results can be seen in Table A.1. First of all one can see that the approximation formulas, especially for the grouping operator, produce values close to the measured numbers. This shows that reasoning based on static analysis is a feasible task. When $R_{max\ per\ window}$ is large and R_{window} is small, the grouping operator performs poorly in comparison to the sorting operator. When the window contains 16 records and the maximum window size is 2^{20} $(= 1,048,576)$, the sorting operator produces results 16,000 times faster than the grouping operator. While the sorting operator is better suited in all instances where R_{window} is 16, when R_{window} is 256, the sorting operator is only better when $R_{max\ per\ window}$ is larger than 2^{16}. For 2^{12} $(= 4,096)$ records in a window, the grouping operator is consistently better suited, because the complexity of the sorting algorithm rises quadratically with the number of elements in a window. For 2^{16} $(= 65,536)$ records, the sorting simulation did not even succeed.

Using the approximation formulas, one can calculate the break-even point between grouping and sorting performance for different levels of "window utilization" (i.e., $\frac{R_{window}}{R_{max\ per\ window}}$). For a utilization of 100%, sorting is better than grouping for less than 36 records, for a utilization of 10%, the break-even point is 108 records, for 1% it is ca. 800, for 0.1% it is ca. 8,000, and for 0.01% it is ca. 80,000 records.

One has to note that there are also sorting algorithms with better algorithmic complexity. The implementation presented in this work sorts in-place and uses minimal amounts of memory, which is a scarce resource on FPGAs. Faster algorithms like QuickSort require more memory and would be outperformed by a hash-map implementation in most cases anyway. The memory overhead of the grouping operator in comparison to the sorting operator is the additional bucket BRAM and control structures in record BRAM. For 2^{20} ($= 1,048,576$) records and a record size of 64 bytes, the grouping operator uses 35% more memory than the sorting implementation.

5.1.4 Reconfiguration latency

Section 3.3 described how operators are configured and Section 4 presented how a configuration can be received using an AXI-Stream. The reconfiguration latency can be defined as the number of clock cycles it takes to receive the new configuration and to apply the configuration.

The stream processing system has been integrated using a 512-bit wide AXI-Stream. It can therefore receive 64 bytes of data within a single clock cycle. Taking the configuration of the timestamp-based tumbling window word count query as an example, which is 67 bytes large, it would take two clock cycles to receive it. But there is also overhead induced by the stream protocol. Assuming we are in the "control" state, there is one additional message (i.e., one additional clock cycle) that announces the start of the configuration. Therefore it takes three clock cycles in total to receive the configuration. The configuration is stored in the configuration registers in the same clock cycle as the last configuration part arrived and the next incoming record can be processed based on it. This brings the total reconfiguration latency to 3 clock cycles or in general $1 + \lceil \frac{|C|}{64B} \rceil$ clock cycles, where $|C|$ is the size of the configuration in bytes. At 250MHz clock speed, the latency would be 12 nanoseconds.

5.2 Configurability analysis

This section will investigate the impact that different levels of configurability have on logic usage, performance characteristics, and synthesis time of the design. This can give insights into which level of configurability is feasible for a certain application scenario.

5.2.1 Methodology

A query design has been synthesized using Xilinx Vivado 2019.2 for the Xilinx Alveo U200 card[3] as a non-configurable, configurable and composable query. The query is a timestamp-based sliding window word count with 4 parallel window processing branches. The query requires one timestamp-based windowing operator, one window synchronizer, 4 grouping operators, 4 group delimiter operators, 4 aggregation operators, 4 stream forks, 4 stream joins, 4 stream delay operators, and 4 projection operators. The composable query would technically allow to over-provision operators, i.e., there are more operators available than needed for the query. In order to be able to precisely extract the logic resource overhead between the three configurability levels, there are only as many operators as needed provisioned for the composable variants of the queries. The record size of 64 bytes and the amount of BRAM for operators, e.g. for grouping, was also the same for the configurable and composable query. The non-configurable query was configured to expect a 5-byte input record and to output a 9-byte output record. The synthesized design only includes the query logic and no components for host interfacing.

5.2.2 Logic usage

	Operators			Configuration			Interconnect		
	LUTs	FFs	BR	LUTs	FFs	BR	LUTs	FFs	BR
Non-config.	1,734	2,003	30	0	0	0	0	0	0
Configurable	513,818	23,838	200	0	189	0	0	0	0
Composable	513,818	23,997	200	0	1,314	0	444,264	2,261	0

Table 14: Logic resource utilization of the sample query by operator, configuration and interconnect logic.

[3] `https://www.xilinx.com/products/boards-and-kits/alveo/u200.html`

Query logic usage Table 14 shows the queries' overall logic resource usage separated into operator, configuration, and interconnect logic as output by Xilinx Vivado's hierarchical utilization report. It has to be noted that these numbers are not 100% accurate, since it is not entirely possible to trace the synthesized components back to their design sources. Especially when optimizations have already been conducted, the logic boundaries of operators are fluid. Nonetheless, the numbers give a rough idea of how many logic resources can be attributed to a certain design entity.

One can see that the non-configurable query has the lowest resource usage: It uses 7,134 look-up tables (LUTs), 2,003 flip-flops (FFs), and 30 BRAM blocks in total. As it has no configurability and interconnect logic, its resource usage in those categories is zero. The configurable queries' operators have a 296 times higher LUT usage, use 12 times as many FFs, and 6.6 as many BRAM blocks. 189 additional FFs can be attributed to storing the configuration. It does not have any interconnect resource usage as the query structure is static. In comparison to the configurable query, the composable query uses the same logic resources for the operators, but 7 times as many FFs for the configuration, which can be traced back to the configuration also having to represent query structure information. The interconnect logic uses almost as many LUTs as the operators. The BRAM usage between the composable and configurable query is the same. In total, the composable query uses 86% more LUTs than the configurable query, and 55,152% more LUTs than the non-configurable query. The composable query has a 15% higher flip-flop usage than the configurable query and a 1,276% higher usage compared to the non-configurable query.

Operator logic usage One can also analyze the resource usage on an operator level. Table 15 shows the LUT, FF, and BRAM usage of the sample query's operators by configurability level. The first thing to note is that the stream fork and stream join operator have no logic usage at all in the non-configurable and configurable queries. This is because these operators are used to represent query structure, which is static for both these configurability levels. Similarly, the projection operator has been optimized out of the non-configurable query, because the operator's configuration was static and therefore the projection could be resolved at synthesis time. Section 3.2.3 has shown that projection is essentially just a reordering of wires. In general, all operators use significantly less resources in the non-configurable query, because of the optimization possible given a configuration that is already known at synthesis time. As mentioned before, this configuration represented a word count query that uses 5 bytes out of the 64-byte input record and uses 9 bytes from the 64-byte output record for the result. The synthesizer can optimize out all logic that does not reflect in these 9 output bytes. This is the case for all logic resource types, including BRAM: While the grouping operator uses 40 BRAM blocks in total in the configurable and composable query, in the non-configurable query only 5 BRAM blocks are used because not the entire 64-byte record has to be stored, but only the part of the record that reflects in the output. The aggregation operator only has to perform the count aggregation and the synthesizer can optimize all other aggregation logic away, as can be seen in the mere 11 LUTs that the aggregation operator uses in the non-configurable query.

	Non-configurable			Configurable			Composable		
	LUTs	FFs	BR	LUTs	FFs	BR	LUTs	FFs	BR
Windowing	449	365	2	16,120	1,804	8	25,559	2,481	8
Window synchr.	225	282	8	2,179	2,208	32	4,066	2,229	32
Grouping	245	291	5	33,468	2,133	40	66,875	2,768	40
Group delimiter	8	22	0	55,129	1,030	0	66,875	1,060	0
Aggregation	11	24	0	34,971	1,069	0	44,638	1,103	0
Projection	0	0	0	37,268	72	0	48,557	101	0
Stream delay	1	2	0	1	2	0	7,987	542	0
Stream fork	0	0	0	0	0	0	9,515	32	0
Stream join	0	0	0	0	0	0	5,136	26	0

Table 15: Logic resource utilization per operator for the sample query. The numbers include both operator, configuration, and interconnect logic.

The orders of magnitude higher logic usage of the configurable and composable query can mainly be traced back to the larger record size that can be processed. While for the non-configurable query, the synthesizer can optimize based on the configuration, for the other query types, the record fields that the operators operate on might change at any time. Therefore the operators have to be synthesized to handle records up to the maximum record size (64 in this case). The logic resource difference between the composable and configurable query is almost entirely caused by interconnect logic.

Interconnect complexity The resource usage of the interconnect logic greatly depends on the number of operators to interconnect. For example, the composable form of the timestamp-based tumbling window word count query (Section 3.4.1) has an operator usage of 24,972 LUTs and interconnect usage of 166,600 LUTs, since it only has 10 operators. The sample word count query presented in this section has 45 operators, a 513,818 LUTs operator usage, and 444,264 LUTs interconnect usage. With 89 operators the biggest query presented in this work, the fixed-size sliding window sum aggregation query (Section 3.4.2) uses 1,585,581 LUTs for operators and 1,926,010 LUTs for interconnect logic, because of its ten parallel window processing branches.

FPGA utilization Based on the measured logic resource usage, one can interpolate the maximum query size with regard to the number of operators for a given FPGA device. The Xilinx Alveo U200 for example has 892,000 LUTs. Based on the sample word count query LUT usage, one could fit the non-configurable query 514 times on there. On the other hand, the configurable query already occupies 57.6% of available LUTs and the composable query would not even fit. This of course greatly depends on the actual query: a composable version of the timestamp-based tumbling window word count query could fit almost 5 times onto the FPGA. It has to be noted that these are estimations that can vary because of actual logic layout requirements on the chip, but nonetheless, give a decent idea of maximum query complexity. Furthermore, research has shown that not more than 70-80% of available LUTs in an FPGA should actually be occupied by logic[3].

Table 15 also gives a good idea as to how many individual operators can fit onto an FPGA for different levels of configurability. One can also see that logic usage greatly differs between operators. While aggregation and grouping are expensive in terms of logic usage, stream management operators require fewer resources because they are of lower complexity.

5.2.3 Performance

The performance properties of operators do not differ between composable, configurable and non-configurable instances. The observations made in 5.1 apply to all configurability levels, i.e., a fixed-size windowing operator will always have one clock cycle latency and one record per clock cycle throughput. But the effectively realizable clock speed of an FPGA is tied to the longest signal path in the synthesized design. How long this signal path is, depends on the actual query and the used operators. Since some operators are combinational, such as selection or projection, one cannot chain an arbitrary amount of them together in a query, since eventually, the resulting combinational signal path will not allow clocking the FPGA at the desired frequency. Instead, a register (i.e., a signal delay operator) has to be inserted at a suitable position to buffer the result between clock cycles. While this helps to achieve timing, the latency is increased by one clock cycle.

While the sample word count query achieves the desired timing closure of 250MHz for its non-configurable and configurable versions, it becomes tricky for composable queries. The issue here is that the communication paths are not defined at synthesis time, since the interconnect can be configured dynamically. When there are combinational operators within a composable query, there technically exists one long signal path through all of them. There can also exist combinational loops since operators can form loops through the interconnect. Without knowing the query configuration at synthesis, it is hard to analyze whether the design will achieve timing. A sub-optimal solution for this could be to prohibit combinational operators, i.e., every operator has an output register. This will increase latency and prohibit optimizations in which multiple combinational operators fit in the same clock cycle.

5.2.4 Synthesis time

The main argument for reconfigurable stream processing queries is the avoidance of long synthesis times on query changes. The sample word count query has been synthesized as a non-configurable, configurable, and composable query on a machine with an AMD EPYC 7742[4] processor with 64 cores. It is noteworthy that Xilinx Vivado does not utilize all available cores for synthesis. Some parts of the synthesis process are even only single-threaded[5]. Synthesizing the non-configurable query took 114 seconds, while the reconfigurable query took 710 seconds to synthesize. The composable query had a significantly higher synthesis time, which was 2,319 seconds.

The synthesis time of a composable query depends on a number of factors, such as the number of operators and which operators have been provisioned. More complex operators such as sorting or grouping can take longer to synthesize than simple stream management operators. For contrast, a composable version of the timestamp-based tumbling window word count query (Section 3.4.1) took 3 hours and 32 minutes to synthesize, while the fixed-size sliding window sum aggregation query (Section 3.4.2) took 12 hours and 50 minutes to synthesize.

Query generation through composable query optimization One can obtain a non-configurable query either by writing code that combines operators or by synthesizing a composable query with a non-changing configuration, as described

[4]`https://www.amd.com/de/products/cpu/amd-epyc-7742`

[5]`https://docs.xilinx.com/v/u/2019.2-English/ug904-vivado-implementation`

in Section 4.5. To evaluate the optimization effort necessary when synthesizing using a composable query with constant configuration, the sample word count query has been over-provisioned with operators: 4 additional selection operators, 4 additional sorting operators, 4 additional stream metadata operators, and one additional fixed-size windowing operator. Synthesis took 13 hours and 38 minutes, which is approximately 430 times as long as synthesizing the non-configurable query from code.

5.3 Execution analysis

This section evaluates the performance of the timestamp-based tumbling window word count query (see Section 3.4.1) and fixed-size sliding window sum aggregation query (see Section 3.4.2) when executed on actual hardware.

5.3.1 Methodology

The non-configurable versions of the queries have been integrated into the SNAP framework as described in Section 4 and synthesized for the Nallatech N250S card that includes a Xilinx Kintex US KU060 FPGA. The FPGA card was connected to an IBM S824L Power 8 host and has been clocked to 250MHz. The host interface width is 64 bytes, allowing in theory a throughput of 250 million 64B-records per second. In practice, the host integration cannot satisfy these speeds, as seen in this section. The burst size has been set to 64, so the system requests records in batches of 64 from host memory. This increases throughput at the expense of having to perform batching. One experiment has also been carried out where the burst size was set to 1. To be able to compare the query performances with a baseline, a design with an empty query has been synthesized additionally, i.e., a query that just passes records through.

5.3.2 Stream characteristics

The records passed to the query were randomly generated, except for the timestamp fields for the word count query, which have to be steadily increasing. As this query performs grouping, the maximum number of records per window for this query has been set to 65,536 and all windows were utilized 100%, i.e., the actual number of records per window was also 65,536. This will reflect in the throughput of the query, as described in Section 5.1.3.

5.3.3 Results

The latency of the system has been obtained by measuring the processing time of one single record. As the system streams from host memory into host memory, this latency is the end-to-end latency. The baseline latency measured is 55 µs. The aggregation query has the same latency because its latency is only two clock cycles, as presented in Section 5.1. The word count query on the other hand has a measured latency of 586 µs, because of the large latency of the grouping operator. Using the grouping operator's latency approximation formula from Section 5.1.3, a latency of 131,089 clock cycles can be calculated for the query. Divided by the clock frequency of 250MHz, the operator's latency is 524 µs. Adding the 55 µs baseline latency, approximately sums up to the 586 µs of measured latency. It has to be noted here that this higher latency is only obtained because the window to which the record belongs has been ended, triggering the reordering logic of the grouping operator. If the windows were not ended, the latency would be the same as the baseline, as the grouping operator would only write the record to memory and not do further processing, resulting only in a dozen clock cycles latency.

Remember that the processing model of the host integration framework is job-based and requires specifying a host memory region with the input records. The latency reflects how long it takes to start a job and to process one record, so throughput depends on the number of records submitted per job. In a ”true” streaming case this would be one record per job. The baseline throughput, in this case, is 18,000 records per second or 9.3Mbit/s, as can be seen in Fig. 22. Transmitting ten records per job also has a ten times higher throughput since the number of records read from host memory is still smaller than the burst size of 64. This means it roughly takes as long to transmit one record as it takes to transmit ten. Transmitting 100 records per job takes two bursts, i.e., two read requests to the host. The throughput only increases by a factor of 8.9 now in comparison to ten records per job. The more records per job the more negligible the job start overhead becomes. When transmitting 10,000 records per job, the throughput is 16.2 million records per second or 8,311.7 Mbit/s respectively. At the maximum number of records per job, which is ca. 32 million, throughput is 18.9 million records or 9,668.1 Mbit per second.

Similarly to latency, the aggregation query has the same throughput as the baseline, because its throughput is one record per clock cycle, resulting in no stalling. The word count query on the other hand consistently has around nine times lower throughput, caused by its grouping operator being a bottleneck. Section 5.1.3 has shown that the approximate throughput of the grouping operator is one-ninth when windows are fully utilized. This is now also resembled in the actual execution

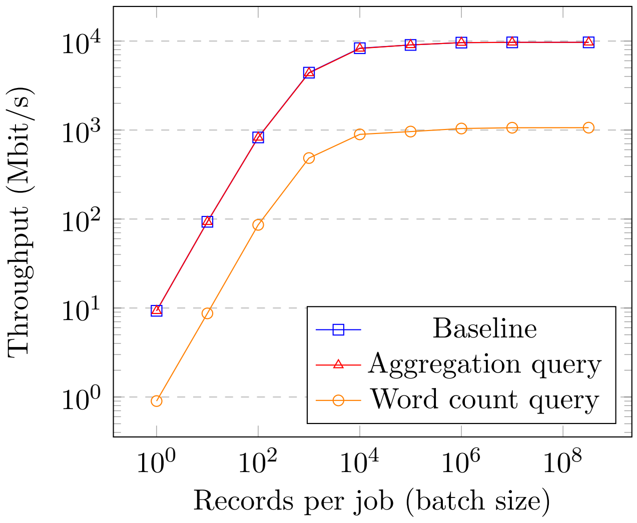


Figure 22: Throughput comparison between baseline, fixed-size sliding window sum aggregation, and timestamp-based tumbling window word count queries. See sources in Table A.2, Table A.4, Table A.3.

throughput.

The burst size specifies how many records are requested from the host at once. This also affects throughput. With the maximum records per job, requesting only one instead of 64 records at once results in a throughput of 649.9 Mbit/s, which is a 93.3% reduction. This shows that the burst size and number of records per job have to be chosen sensibly because both have an impact on throughput when batching is reduced.

6 Discussion

This section discusses some of the results of the previous section and provides directions for further research.

Partial configurability A configurable query has a significantly higher logic resource usage than its non-configurable counterpart. The difference can be orders of magnitude higher, depending on the constant ”configuration” of the non-configurable query, which essentially defines the optimization potential. A way to limit the impact the configurability has, one could reduce the range of configuration options. The observations made in this work are based on the configurable query allowing all parameters of all operators to be configured and the composable query allowing each operator to be interconnected with every other operator. If it is in advance known that certain parameters will not need to be configured, this could be indicated at synthesis time. This would allow for more optimization and possibly lower resource usage and synthesis time.

Interconnect optimization This work has shown that the interconnect complexity is significant. This is due to it providing a connection from all operators to all operators, i.e., it has quadratic complexity. This could be alleviated by lowering the number of operator interconnections. A heuristic could be developed that gives an answer as to which operators should be interconnected and which not based on whether it makes sense to have them as successive operators in a query. There might be, e.g., no reason why two windowing operators or two aggregation operators should be interconnected. Interconnecting two projection or two selection operators on the other hand seems reasonable. Nonetheless, an operator not necessarily has to be interconnected with all other operators. It might be a reasonable limitation to interconnect an operator with just a fraction of all operators as long as there is at least a connection with one operator of each type possible. This comes close to the ideas of Najafi, where there are query templates in which operators can be ”plugged” in, albeit limiting flexibility less. One could also think of not allowing combinational operators in composable queries, since combinational operators cause long signal paths in combination with the interconnect logic. This could aid in determining the timing properties of composable queries better but would increase latencies.

Partial reconfiguration for composable operators Even when using a composable query, one is limited by the set of operators available in the design. What operators are available is determined at synthesis time of the composable query. When other operators than the currently provisioned are required, one has to synthesize a new composable query. This work has shown that this is a lengthy process. A solution to this could be partial reconfiguration of the FPGA, which is the process of just replacing parts of the design on it. One could have multiple precompiled partially-reconfigurable modules that contain a certain set of operators. When another set of operators is required in the composable query, these modules could be swapped in and out of the design. Dennl et al. have already presented such a "module library" for SQL acceleration. Performing a partial reconfiguration is faster than recompiling an entire design but slower than the query reconfiguration mechanism presented in this work.

Multiple stream widths Composable queries presented in this work have a globally-defined maximum record size that determines the width of interconnects and what record size operators can process. It might make sense to have operators and interconnects of different sizes within one system. This would allow wider or shallower local processing pipelines, leading to better resource utilization and removing processing limitations. An example where this could bring a benefit is the stream join operator that merges two input streams of width N into one stream of width N, having to discard parts of the input streams. With multiple stream widths, the stream join operator could output a stream of width $2N$ to downstream operators that can handle this record width. Eventually, a 2:1 projection operator could shrink the stream of width $2N$ back into a stream of width N. While this can greatly increase processing capabilities, it also limits flexibility as now there are incompatibilities between different operators. Stream adapters would be necessary to convert between stream widths.

Multiple reset domains All operators have a reset input signal that allows resetting their state. This is useful after a reconfiguration so that new incoming records can be processed immediately based on this new configuration. The reset signal in this work is global, i.e., there is one reset signal that is distributed to all operators. If a reset is triggered this would reset all operators, but there are instances where a global reset is not desired. This work has shown in Section 3.4.3 how multi-queries can be realized. If only one of the queries shall be reset, a global reset signal is impractical. Local reset signals are needed to facilitate targeted resets. This could be achieved by having a reset signal for each operator that can be controlled individually. To partially reset the query one would have to send a bitmap that indicates which operators should receive the reset signal and which not as part of the host interface protocol.

HBM for large state Limitations of the presented system are that it only supports in-order streams and that there is a maximum number of records per window. Both stem from little memory resources on FPGAs. Some FPGA boards come with HBM (high bandwidth memory) equipped that is of significantly larger size than the available BRAM. For instance, the Xilinx Alveo U200 card[6] comes with 35MB of on-chip RAM and up to 64GB of off-chip memory. There has already been work around using HBM for data processing showing promising results[15, 26]. A problem with HBM in comparison to on-chip RAM is its lower throughput and higher latency. On the U200 card, on-chip RAM can be accessed with 31TB/s bandwidth, while off-chip RAM only has a 77GB/s bandwidth.

[6]`https://www.xilinx.com/products/boards-and-kits/alveo/u200.html`

7 Related work

There has already been some work around stream processing on FPGAs, some of which is outlined in this section.

Glacier[9] is a system developed by Mueller et al. that can compile SQL queries into FPGA-based stream processing queries. Many of its principles are similar to the system presented in this work, such as record propagation or logic replication for parallel processing branches. In comparison to the system presented in this work, Glacier only generates non-configurable queries. A major difference arises in the concept of grouping. Glacier uses content-addressable memory (CAM), which has very fast lookup speeds, to determine one of multiple processing branches on which to process the record. This concept requires a certain number of processing branches to be provisioned, which also means the number of groups has to be known beforehand. This concept may not scale well for a large number of distinct groups. There is also a difference as to where to place the result of operations. Glacier allocates a record bus of fixed width, which is propagated between operators. An operator, e.g. selection, places its result within that bus. That imposes a limit on the state that can be carried along a record because the bus is fixed in width. A query in this work would split the stream before an aggregation operator and merge the branches after, realizing more flexible computation, at the expense of being more complicated.

Teubner et al. have also pointed out that compilation time poses a limiting factor when adaptations to hardware-based processing are necessary[21]. They introduced a hardware implementation for XML projection that is configurable at run-time. They build upon finite state automata for parsing XML nodes. Similar to query reconfiguration in this work, the processing behavior is based on parameters that can be updated at run-time. The new parameters come immediately into effect, resulting in low reconfiguration times. They store these parameters in BRAM, which could also be a viable option for the system presented in this work.

The Flexible Query Processor[12] (FQP) has been introduced by Najafi, which is a configurable streaming processor for FPGAs tailored around joining streams. It is based on operator blocks that support bidirectional data flow, which is required for joining. The operator blocks can be reconfigured using an instruction set. The system allows coarse-grained control for composing queries. It provides query templates of which the operator blocks can be configured, in contrast to the fine-grained query composition mechanism presented in this work in which operators can be arbitrarily connected.

As a more general parametrizable processing system, IBM Netezza[17] provides reconfigurable circuits for data processing in data warehousing environments. Netezza uses a commodity FPGA to push processing closer to the data source and designs for it can be compiled quickly. But there are limitations with regards to the queries that can be realized as it only supports selections and projections.

8 Conclusion

This work has presented a stream processing system in which operators can be reconfigured at run-time with minimal latency impact. Furthermore, a configurable operator interconnect has been introduced that allows to also change query structure at run-time.

Based on the results presented in this work, one can conclude that the level of configurability has a significant impact on the logic resource usage and synthesis time, but less on performance characteristics. Naturally, a non-configurable query shows a lower resource usage than a configurable or composable version. But the actual difference is significant because a non-configurable query allows for intensive optimization, while for configurable and composable queries, optimization is traded for flexibility. For a sample word count query, it has been demonstrated that its configurable version uses almost 300 times as many look-up tables when synthesized. A composable query consisting of the same operators would even use approximately 550 times as many look-up tables. Given that FPGA resources are not endless, this puts limits to the maximum query complexity that can be realized on an FPGA. In a composable system, where query structure is defined dynamically, the logic resource usage is also dependent on the number of provisioned operators. Since the presented operator interconnect connects each operator with every other operator, it has a quadratic complexity in terms of switching logic.

One large factor here is also the maximum record size for which operators have to be provisioned since it defines the amount of logic within operators to handle this records size as well as the amount of wiring necessary to transfer records between operators. In non-configurable queries, the synthesizer can optimize out all logic that does not have an effect on output pins, reducing the necessary logic resources drastically. This also applies to BRAM blocks, since these will only have to store the actually necessary parts of the record instead of entire records. In a configurable query, BRAM has to be designed to span the full width of the record. This is especially wasteful considering that scarce memory resources are often a limiting factor of FPGA-based data processing[7].

The synthesis of the configurable query took over six times as long as for the non-configurable query. For a composable query, the effect is even more severe, resulting in synthesis times over 19 times higher. It is not feasible to acquire a non-configurable query by synthesizing a composable query with a constant configuration, as synthesis times can now extend into hours. Synthesizing a non-configurable query from code was 430 times faster than from a composable query. The performance characteristics of operators do not differ between a non-configurable, configurable, and composable query. It depends on the specific query whether timing closure is achieved for a non-configurable or configurable query, where the query structure is static. In a composable query, it is hard to determine timing properties since there could be long combinational paths in the design, caused by combinational operators.

A static evaluation has shown that performance characteristics of a hardware-based query can be inferred from its operators, of which the characteristics can in turn be inferred from their design. This is an advantage over CPU-based systems, in which it is incredibly hard to reason over performance metrics on a clock cycle level. Comparing the statically-inferred throughput of the grouping operator with actual benchmark execution results, one can see that both approaches yield similar results: the grouping throughput for full windows can be approximated to $\frac{1}{9}$, and the fraction of baseline throughput that the word count query achieves in benchmarks is also similar to that number. On the other hand, theoretical observations do not always translate into reality. While an FPGA clocked at 250MHz could theoretically process 250 million records per second using the presented system, in reality, the setup was limited by the host integration, only achieving around 18.9 million records per second. But the static analysis has also demonstrated very low reconfiguration latencies, within double-digit nanoseconds, depending on the query configuration size.

In summary, there is a significant cost to pay for configurability, since far less optimization is possible than for non-configurable queries. A compromise could be to limit configurability only to parts of the query or operators, gaining optimization potential but again losing flexibility.

References

[1] T. Akidau, S. Chernyak, and R. Lax. *Streaming Systems: The What, Where, When, and How of Large-Scale Data Processing.* O'Reilly Media, 2018. ISBN 978-1-4919-8382-9.

[2] Alexandre Castellane and Bruno Mesnet. Enabling fast and highly effective FPGA design process using the CAPI SNAP framework. In *High Performance Computing: ISC High Performance 2019 International Workshops, Frankfurt, Germany, June 16-20, 2019, Revised Selected Papers 34*, pages 317–329. Springer, 2019.

[3] Jason Cong and Kirill Minkovich. Optimality study of logic synthesis for LUT-based FPGAs. In *Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, pages 33–40, 2006.

[4] Christopher Dennl, Daniel Ziener, and Jurgen Teich. On-the-fly Composition of FPGA-Based SQL Query Accelerators Using a Partially Reconfigurable Module Library. In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, pages 45–52, 2012. doi: 10.1109/FCCM.2012.18.

[5] Marcel Gort and Jason Anderson. Design re-use for compile time reduction in FPGA high-level synthesis flows. In *2014 International Conference on Field-Programmable Technology (FPT)*, pages 4–11, 2014. doi: 10.1109/FPT.2014.7082746.

[6] Zsolt István, David Sidler, and Gustavo Alonso. Caribou: Intelligent Distributed Storage. *Proc. VLDB Endow.*, 10(11):1202–1213, August 2017. ISSN 2150-8097. doi: 10.14778/3137628.3137632. Publisher: VLDB Endowment.

[7] Zsolt István, Kaan Kara, and David Sidler. *FPGA-Accelerated Analytics: From Single Nodes to Clusters.* 2020. Publication Title: FPGA-Accelerated Analytics: From Single Nodes to Clusters.

[8] Ian Kuon and Jonathan Rose. Measuring the Gap Between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):203–215, 2007. doi: 10.1109/TCAD.2006.884574.

[9] Rene Mueller, Jens Teubner, and Gustavo Alonso. Streams on Wires: A Query Compiler for FPGAs. *Proc. VLDB Endow.*, 2(1):229–240, August 2009. ISSN 2150-8097. doi: 10.14778/1687627.1687654. Publisher: VLDB Endowment.

[10] Rene Mueller, Jens Teubner, and Gustavo Alonso. Sorting networks on FPGAs. *The VLDB Journal*, 21:1–23, 2012. Publisher: Springer.

[11] Kaspar Mätas, Kristiyan Manev, Joseph Powell, and Dirk Koch. Automated Generation and Orchestration of Stream Processing Pipelines on FPGAs. In *2022 International Conference on Field-Programmable Technology (ICFPT)*, pages 1–10, 2022. doi: 10.1109/ICFPT56656.2022.9974596.

[12] Mohammadreza Najafi. *Hardware Accelerated Stream Processing.* PhD thesis, Technische Universität München, June 2019. URL `https://mediatum.ub.tum.de/doc/1486617/62468.pdf`.

[13] Kostas Pagiamtzis and Ali Sheikholeslami. Content-addressable memory (CAM)

circuits and architectures: A tutorial and survey. *IEEE journal of solid-state circuits*, 41(3):712–727, 2006. Publisher: IEEE.

[14] Mohammad Sadoghi, Rija Javed, Naif Tarafdar, Harsh Singh, Rohan Palaniappan, and Hans-Arno Jacobsen. Multi-query Stream Processing on FPGAs. In *2012 IEEE 28th International Conference on Data Engineering*, pages 1229–1232, 2012. doi: 10.1109/ICDE.2012.39.

[15] Runbin Shi, Kaan Kara, Christoph Hagleitner, Dionysios Diamantopoulos, Dimitris Syrivelis, and Gustavo Alonso. Exploiting hbm on fpgas for data processing. *ACM Transactions on Reconfigurable Technology and Systems*, 15(4):1–27, 2022. Publisher: ACM New York, NY.

[16] David Sidler, Zsolt Istvan, Muhsen Owaida, Kaan Kara, and Gustavo Alonso. DoppioDB: A Hardware Accelerated Database. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1659–1662, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 978-1-4503-4197-4. doi: 10.1145/3035918.3058746. event-place: Chicago, Illinois, USA.

[17] Malcolm Singh and Ben Leonhardi. Introduction to the IBM Netezza warehouse appliance. In *Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research*, pages 385–386, 2011.

[18] V. Sklyarov, I. Skliarova, A. Barkalov, and L. Titarenko. *Synthesis and Optimization of FPGA-Based Systems*. Lecture Notes in Electrical Engineering. Springer International Publishing, 2014. ISBN 978-3-319-04708-9.

[19] Jens Teubner. FPGAs for data processing: Current state. *it - Information Technology*, 59(3):125–131, 2017. doi: doi:10.1515/itit-2016-0046.

[20] Jens Teubner and Louis Woods. Secure Data Processing. In *Data Processing on FPGAs*, pages 83–88. Springer, 2013.

[21] Jens Teubner, Louis Woods, and Chongling Nie. Skeleton automata for FPGAs: reconfiguring without reconstructing. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 229–240, 2012.

[22] Lukas Wenzel, Robert Schmid, Balthasar Martin, Max Plauth, Felix Eberhardt, and Andreas Polze. Getting started with CAPI SNAP: Hardware development for software engineers. In *Euro-Par 2018: Parallel Processing Workshops: Euro-Par 2018 International Workshops, Turin, Italy, August 27-28, 2018, Revised Selected Papers 24*, pages 187–198. Springer, 2019.

[23] Sven Woop, Erik Brunvand, and Philipp Slusallek. Estimating Performance of a Ray-Tracing ASIC Design. In *2006 IEEE Symposium on Interactive Ray Tracing*, pages 7–14, 2006. doi: 10.1109/RT.2006.280209.

[24] Yuanlong Xiao, Dongjoon Park, Andrew Butt, Hans Giesen, Zhaoyang Han, Rui Ding, Nevo Magnezi, Raphael Rubin, and Andre DeHon. Reducing FPGA Compile Time with Separate Compilation for FPGA Building Blocks. In *2019 International Conference on Field-Programmable Technology (ICFPT)*, pages 153–161, 2019. doi: 10.1109/ICFPT47387.2019.00026.

[25] Yuanlong Xiao, Eric Micallef, Andrew Butt, Matthew Hofmann, Marc Alston, Matthew Goldsmith, Andrew Merczynski-Hait, and André DeHon. PLD: Fast FPGA Compilation to Make Reconfigurable Acceleration Compatible with Modern Incremental Refinement Software Development. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, pages 933–945, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 978-1-4503-9205-1. doi: 10.1145/3503222.3507740. event-place: Lausanne, Switzerland.

[26] Yang Yang, Sanmukh R Kuppannagari, and Viktor K Prasanna. A high throughput parallel hash table accelerator on hbm-enabled fpgas. In *2020 International Conference on Field-Programmable Technology (ICFPT)*, pages 148–153. IEEE, 2020.

[27] Zhipeng Zhao, Joseph Melber, Siddharth Sahay, Shashank Obla, Eriko Nurvitadhi, and James C. Hoe. Exploiting the Common Case when Accelerating Input-Dependent Stream Processing by FPGA. *IEEE Transactions on Computers*, pages 1–13, 2022. doi: 10.1109/TC.2022.3200576.

Erklärung (Declaration of Academic Honesty)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe. Die selbstständige und eigenständige Anfertigung versichert an Eides statt:

I hereby declare that the thesis submitted is my own, unaided work, completed without any unpermitted external help. Only the sources and resources listed were used. The independent and unaided completion of this thesis is affirmed by affidavit:

Potsdam, 31.03.2023

Ben-Noah Engelhaupt

A Appendix

	R_{window}	$R_{max\ per\ window}$	$L_{measured}$	L_{approx}	$T^{-1}_{measured}$	T^{-1}_{approx}
grouping	2^4	2^8	589	624	36.8	39
sort			129	64	8.1	4
grouping		2^{12}	8,269	8,304	516.8	519
sort			129	64	8.1	4
grouping		2^{16}	131,634	131,184	8,227.1	8,199
sort			129	64	8.1	4
grouping		2^{20}	2,097,229	2,097,264	131,076.8	131,079
sort			129	64	8.1	4
grouping	2^8	2^8	2,319	2,304	9.1	9
sort			16,535	16,384	64.6	64
grouping		2^{12}	9,524	9,984	37.2	39
sort			16,535	16,384	64.6	64
grouping		2^{16}	132,354	132,864	517.0	519
sort			16,535	16,384	64.6	64
grouping		2^{20}	2,098,429	2,098,944	8,197.0	8,199
sort			16,535	16,384	64.6	64
grouping	2^{12}	2^{12}	37,378	36,864	9.1	9
sort			4,006,197	4,194,304	978.1	1,024
grouping		2^{16}	152,128	159,744	37.1	39
sort			4,006,197	4,194,304	978.1	1,024
grouping		2^{20}	2,117,639	2,125,824	517.0	519
sort			4,006,197	4,194,304	978.1	1,024
grouping	2^{16}	2^{16}	595,322	589,824	9.1	9
grouping		2^{20}	2,434,473	2,555,904	37.1	39
grouping	2^{20}	2^{20}	9,520,545	9,437,184	9.1	9

Table A.1: Comparison of grouping and sorting operator performance. $L_{measured}$ is the measured latency, while L_{approx} is the latency approximated using the latency formula. $T^{-1}_{measured}$ is the inverse throughput, i.e., the number of clock cycles it takes to process one record. T^{-1}_{approx} is the result of the throughput approximation formula.

```vhdl
constant c_timestamp_word_count_query_configuration : std_logic_vector := (
    -- 9: sink

    -- 8: stream delay (next_id=7)
    x"0007" &

    -- 7: stream join (in1_id=3, in2_id=8, next_id=9)
    x"0003" & x"0008" & x"0009" &

    -- 6: stream fork (next1_id=3, next2_id=1)
    x"0003" & x"0001" &

    -- 5: timestamp_sliding_window (field_end=0, field_start=8,
    --        parallel_windows=1, window_slide=1000000,
    --        window_size=1000000, nexts={2})
    x"0000" & x"0008" & x"00000001" & x"000F4240" & x"000F4240" &
            "0000000000000100" &

    -- 4: group_delimiter (field_end=0, field_start=8, next_id=6)
    x"0008" & x"0028" & x"0006" &

    -- 3: aggregation (fn=0(count), field_end=8, field_start=40, next_id=7)
    x"00" & x"0008" & x"0028" & x"0007" &

    -- 2: grouping (field_end=8, field_start=40, next_id=4)
    x"0008" & x"0028" & x"0004" &

    -- 1: projection (keep_bytes={5,4,3,2,1}, move_len=0, move_dest=0,
    --        move_src=0, next_id=8)
    "0000000000000000000000001111111111111111111111111111111100000000" &
            x"0000" & x"0000" & x"0000" & x"0008" &

    -- 0: source (next_id=5)
    x"0005"
);
```

Figure A.1: VHDL code resembling a configuration of the timestamp-based tumbling window word count query within a composable query.

Records	Execution time (ms)	Throughput (10^6 R/s)	Throughput (Mb/s)
10^0	0.055	0.018	9.3
10^1	0.055	0.181	93.1
10^2	0.062	1.613	825.8
10^3	0.116	8.621	4,413.8
10^4	0.616	16.234	8,311.7
10^5	5.663	17.658	9,041.1
10^6	53.330	18.751	9,600.6
10^7	529.838	18.874	9,663.3
$3.2 * 10^7$	1,741.840	18.883	9,668.1
$3.2 * 10^7$ (no burst)	25,908.713	1.267	649.9

Table A.2: Baseline execution measurements.

Records	Execution time (ms)	Throughput (10^6 R/s)	Throughput (Mb/s)
10^0	0.586	0.002	0.9
10^1	0.586	0.017	8.7
10^2	0.595	0.168	86.0
10^3	1.057	0.946	484.4
10^4	5.729	1.746	893.9
10^5	53.264	1.877	961.0
10^6	492.410	2.031	1,039.9
10^7	4,823.354	2.073	1,061.4
$3.2 * 10^7$	15,841.163	2.077	1,063.4

Table A.3: Execution measurements for the timestamp-based tumbling window word count query from Section 3.4.1.

Records	Execution time (ms)	Throughput (10^6 R/s)	Throughput (Mb/s)
10^0	0.055	0.018	9.3
10^1	0.055	0.181	93.1
10^2	0.062	1.613	825.8
10^3	0.117	8.547	4,376.1
10^4	0.620	16.129	8,258.0
10^5	5.657	17.677	9,050.6
10^6	53.331	18.751	9,600.1
10^7	529.458	18.887	9,670.3
$3.2 * 10^7$	1,740.897	18.893	9,673.2

Table A.4: Execution measurements for the fixed-size sliding sum aggregation query from Section 3.4.2.