Industry Paper: Hardware Accelerated Application Integration Processing

Daniel Ritter, Jonas Dann, Norman May
SAP SE
Dietmar-Hopp-Allee 16
Walldorf 69190
first-name.last-name@sap.com

Stefanie Rinderle-Ma
University of Vienna
Währingerstrasse 29
Vienna, Austria 1090
stefanie.rinderle-ma@univie.ac.at

ABSTRACT

The growing number of (cloud) applications and devices massively increases the communication rate and volume pushing integration systems to their (throughput) limits. While the usage of modern hardware like *Field Programmable Gate Arrays* (FPGAs) led to low latency when employed for query and event processing, application integration adds yet unexplored processing opportunities. In this industry paper, we explore how to program integration semantics (e. g., message routing and transformation) in form of *Enterprise Integration Patterns* (EIP) on top of an FPGA, thus complementing the existing research on FPGA data processing. We focus on message routing, re-define the EIP for stream processing and propose modular hardware implementations as templates that are synthesized to circuits. For our real-world "connected car" scenario (i. e., composed patterns), we discuss common and new optimizations especially relevant for hardware integration processes. Our experimental evaluation shows competitive throughput compared to modern general-purpose CPUs and discusses the results.

CCS CONCEPTS

• Applied computing → Enterprise application integration; **• Hardware → Reconfigurable logic and FPGAs**; **• Information systems →** *Stream management*;

ACM Reference format:
Daniel Ritter, Jonas Dann, Norman May and Stefanie Rinderle-Ma. 2017. Industry Paper: Hardware Accelerated Application Integration Processing. In *Proceedings of ACM International Conference on Distributed Event-Based Systems, Barcelona, Spain, June 19 - 23, 2017 (DEBS '17),* 12 pages.
https://doi.org/http://dx.doi.org/10.1145/3093742.3093911

1 INTRODUCTION

Through the increasing amount of (cloud) applications and devices, *Enterprise Application Integration* (EAI) [13] is facing immense data volumes that have to be processed with high throughput. Consider a "connected car" scenario where each car (e. g., 260.3 million in the US[1]) sends telemetry data and error code messages of one kB every few seconds for further processing in a data warehouse. As in [4], the near real time processing of complex events or patterns is considered a challenging requirement. For such scenarios, Field Programmable Gate Arrays (FPGAs) [25] promise lower latency, higher throughput and lower energy consumption than comparable solutions in software and on general purpose CPUs. With multi-chip processors delivered with FPGAs on the chip (e. g., by Intel), FPGAs might become far more widely used than today, and thus allows them to be included in cloud-scale deployments [5].

Considering only the throughput [22] shows that a single integration system instance would not suffice to process the load of messages generated in the connected car scenario using typical integration patterns. These enterprise integration patterns (EIP) [11] are more complex than message queuing for reliable queues and topics [6, 26] or event and stream processing for continuous queries and alerts [17, 27] for which FPGAs were employed before. At the same time the throughput demands are beyond the ones for query processing using FPGAs [16] which are bound by disk access.

Figure 1 shows how database query processing puts programmable hardware in form of FPGAs into the data path of the systems to evolve them toward heterogeneous many-core systems. We envision EAI processing logic on the FPGA, which we put on the network path between applications, devices and databases. The extensive body of research and industrial work on FPGA-based hardware event stream and data processing reports on competitive results (e. g., reconfigurable logic, low-latency) due to parallel streaming [9] through hardware characteristics like parallel stream evaluation and asynchronous circuits, and reduced power consumption compared to modern general-purpose CPUs.

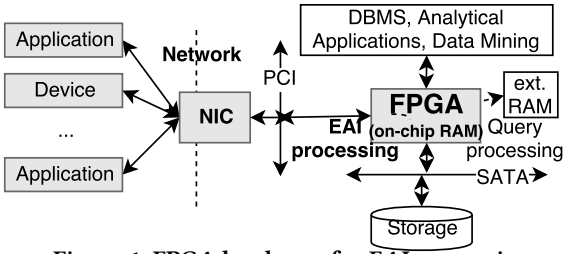


Figure 1: FPGA hardware for EAI processing.

However, hardware is not the "silver bullet" [16], and the efficient usage of FPGAs involves non-trivial aspects such as making the right design decisions for the computation model, dealing with

[1]Number of vehicles in the US, visited 05/2017: http://www.statista.com/statistics/183505/number-of-vehicles-in-the-united-states-since-1990/

DEBS '17, June 19 - 23, 2017, Barcelona, Spain

ACM ISBN 978-1-4503-5065-5…$15.00
https://doi.org/http://dx.doi.org/10.1145/3093742.3093911

low-frequency clocks, balancing the trade-off in usage between synchronous and asynchronous circuits, and resource limitation. In addition, it remains unclear, how the potential of FPGAs can be efficiently exploited for EAI. Important questions about the EAI building blocks and semantics, represented by the EIP, and their usage on FPGA hardware have to be answered. For instance, the EIP are defined for asynchronous, non-streaming cases, while FPGA hardware supports streaming very well. Hence, we have to answer the question "How does the application of EIP on FPGA influence their semantics and implementations?", and find an efficient representation on hardware. Furthermore, EAI is about variety in message protocols, which requires support for non-trivial message formats and operations (e. g., hierarchical formats like JSON and JSONPath predicates). The work on database and event processing discusses some data aspects relevant to application integration, which leads to the question: "Could FPGAs accelerate the message routing even more than transformations?".

In this work - taking the results and conceptual extensions into account - we study the feasibility, advantages and limitations of (composed) EIP on hardware. We define the streaming semantics of selected EIP, because our work fully focuses on streaming (i. e., no message off-loading into RAM), and analyze how they can be deployed on FPGAs. This is not trivial because of the illustrated trade-offs between computation model, throughput, resource limitations, parallelization and diverse integration semantics. We categorize the streaming semantics of the EIP into three template classes based on their interaction with user-defined conditions and expressions: Expression Template (ET), Predicate Template (PT) and No User Template (NUT). Consequently only these classes have to be synthesized to hardware. For the user conditions (i. e., predicate, expression), we define a state machine parsing and matching approach for hierarchical message protocols. In particular the paper contains the following contributions:

(1) An extensive literature review on the FPGA lessons learned and design choices in related domains.
(2) Definition of basic integration semantics (i. e., including message, pipeline) and EIP streaming semantics for most commonly used routing and transformation patterns in integration scenarios according to [22]
(3) Categorization of those patterns into three template classes and their implementations on hardware
(4) Hierarchical message format processing and user conditions on hardware (i. e., predicates and expressions)
(5) Evaluation of the approach using EIPBench [22] benchmarks and extensions for additional tests on FPGAs
(6) A study of non-trivial tradeoffs for message throughput and data sizes, parallelism and resource consumption, as well as optimizations for the connected car example

The work is set into context in Sect. 2, followed by a brief technical background in Sect. 3. We extend the EIP with streaming semantics and categorize them into templates for an efficient implementation on FPGAs in Sect. 4. Section 5 shows the design of format processing. We evaluate our work in Sect. 6, discuss optimizations and conclude in Sect. 7.

2 RELATED WORK

We are not aware of work on implementing EIPs on FPGAs. However, there is a rich body of work in related domains (i. e., query-, complex event and stream processing, message queuing), relevant for our work. Especially, the lessons learned on system design (e. g., parallelism, automata for format processing), the identified trade-offs between synchronous and asynchronous designs, and system integration aspects (e. g., FPGA in the system's data path) have influenced our work. We summarize the related contributions in Tab. 1 and subsequently discuss the approaches.

Table 1: Lessons learned from related work.

Design Decision	Reference	our approach
parallelism: systolic vs. MISD	[3],[16]	systolic for higher throughput
automaton: (non-) deterministic	[1],[6],[7],[10],[14], [20],[24],[28],[29]	deterministic format handling
synch vs asynch circuits	[16]	synchronous with flow control
configurable clocks	[16]	- (for asynch. designs)
back-pressure	[3]	for flow control
avoid long distance streams	[3], [17]	system part
avoid deep logic	[3]	system part
FIFO buffer	[6, 16–18]	BRAM for reliability

2.1 Query Processing

Several lessons learned on query processing are also relevant in the context of EIP. The design of the industrial solutions (e. g., "Netezza Performance Server"[2]), which consists of a number of "snippet processing units", each tightly coupled with network CPU and FPGA, and Kickfire's *MySQL Analytic Appliance*[3] with so-called "SQL Chip") do only give limited insight into their design decisions. However, both systems appear to use FPGAs primarily as customized hardware, with circuits that are geared toward very specific (data warehousing) workloads, but are immutable at runtime. In our approach, we aim at exploiting the configurability of FPGAs by compiling integration operations from an arbitrary workload into a tailormade hardware circuit.

The research on the *Glacier* query to hardware compiler provided valuable learnings with respect to design decisions [16, 18]. As in [3] several types of parallelism are discussed (cf. Tab. 1): e. g., systolic (i. e., pipeline-chain: good scalability even across chips) and MISD (i. e., tree: long signal path), for which we follow their lead to higher throughput systolic parallelism. In contrast to glacier, we do not use an asynchronous (i. e., lower latency, less flip-flop registers), but a modular synchronous stream design with re-usable logic. Hence, our design also does not make use of configurable clock frequencies, which is crucial in an asynchronous design. The frequencies only vary for integation adapters (e. g., TCP, UDP) and the integration processing steps. We also use finite state machines, which naturally translate into FPGAs (inspired by [7]), for message format conversions - in our case hierarchical data in form of JSON messages. The asynchronous design approaches in [16–18] requires FIFO buffers for synchronization. We use FIFOs to implement flow control in form of back-pressure [3].

2.2 Complex Event and Stream Processing

There is a lot of research on stream queries on hardware, hence we only summarize the directly related work. Most influential work

[2]Netezza Corp, visited 05/2017: http://www.netezza.com/
[3]Kickfire, visited 05/2017: http://www.kickfire.com/

was conducted in [17, 27], which analyzes the potential of FPGAs in the domain of data processing using sliding window operators for a median operator using a sorting network. The experimental analysis showed promising results compared to a small on-board PowerPC 405 processor. Compared to our synchronous approach, an asynchronous design fits well to the sorting network (cf. Tab. 1).

The work on event detection using regular expressions [29], efficient pattern matching [1] non-deterministic finite automaton design, complex event processing [10, 24], streaming system [20, 28] and XML/XPath evaluation [14, 15] influenced our work on the message format handling. The latter also sketches the idea of an internet protocol router on packet level, setting the addresses according to a routing table [15], similar to a recipient list [11]. Since no paket throughput or latency measurements were published, it remains a mere design sketch. All of these approaches use deterministic or non-deterministic state machines to represent user-defined conditions on the hardware. For the hierarchical format handling (e. g., JSON messages), we define a general, but resource-reduced way to represent the data (e. g., compared to approaches like [15]) to spare resources for the integration logic.

2.3 Publish/Subscribe and Queuing Systems

Message Brokers are used complementary to application integration systems. The most well-known commercial solution is the "Solace Message Router" [26], which implements a JMS-like topic and queue processing as well as high-availability and disaster recovery on FPGA hardware. The comparison to one of the most popular software brokers shows superior message throughput rates on the hardware[4].

While Publish/Subscribe systems mostly route data without looking into their content – giving them an edge over application integration systems – there is some work on content-based routing in the sense of [11] using FPGAs. [6], for example, proposes an architecture, which adds FPGAs to each message broker that implements XML/XPath message routing using complex memory based XML parser (using content-addressable memory, FIFO buffers) and matcher leveraging the dual port memory for concurrent read/write. This work is comparable to the content-based routing and JSON processing approaches presented in this work. However, our approach manage to process JSON with much lower resource consumption. We only optionally use FIFO buffers based on on-chip BRAM for back-pressure handling.

In general, back-pressure is used for flow control holding off senders to further transmit data until the resource is available. We use back-pressure in our message processing pipeline to avoid scrambled data and allow for TCP-based flow control via the integration adapters. This idea is based on [3], which defines a systolic message queuing system that uses back-pressure based on a wire protocol for synchronous communication (cf. Tab. 1). Furthermore, we should try to avoid long distance pipelines or streams and deep logic in processes [3] in the system part of our pipeline design (cf. Tab. 1). However, user-defined code might violate this rule.

[4]Solace vs Apache Kafka, visited 05/2017: http://www.solacesystems.com/techblog/deconstructing-kafka

2.4 Hardware-accelerated EAI Processing

The EIP authors admit that the current foundations are defined for an asynchronous processing style [21], and a definition for synchronous streaming is missing. In our work, we fill this void and enumerate some of the most relevant EIP as identified in [22] and specify semantically correct streaming semantics. Based in these semantics, we are able to define three template classes that are sufficient to represent all of the patterns and also to synthesize them to hardware. Overall, we are able to implement the complete integration system on hardware (cf. Fig. 1). In the experiments we test different aspects special to integration processing [22] that have not been tested on hardware before.

3 FPGAS FOR MESSAGING

Field-programmable gate arrays are reprogrammable hardware chips for digital logic. FPGAs are an array of logic gates that can be configured to construct arbitrary digital circuits. These circuits are specified using either circuit schematics or hardware description languages such as Verilog or VHDL. A logic design on a FPGA is also referred to as a soft IP-core (intellectual property core). Existing commercial libraries provide a wide range of pre-designed cores, including those of complete CPUs. More than one soft IP-core can be placed onto an FPGA chip. For evaluation and development purposes FPGAs are often shipped, soldered to a board with peripherals, like external RAM, network interfaces or persistent storage. The characteristics of the Arty Artix-7 FPGA used for this paper are listed in Tab. 2.

Table 2: Test Hardware Comparison.

Characteristics	Xilinx XC7A35T	Z600
lookup tables (LUT)	20,800	-
flip-flops (FF)	41,600	-
block RAM (BRAM) / on-board RAM	1,800 kB / 256 MB	- / 24 GB
clock rate (CR)	100 MHz	2.67GHz
cores	-	12

On-chip BRAM (divided in 50*36 and 100*18 kB units) is accessible in one and on-board RAM requires several cycles (depends on fabrics).

3.1 FPGA Architecture

The basic building blocks of FPGAs are Configurable Logic Blocks (CLBs). Each CLB has two slices that each contain Lookup Tables (LUTs) and Flip Flops (FFs). LUTs can be configured to arbitrary functions that map multiple bits (6 in the Xilinx 7-Series) to one (e. g., to implement AND and OR gates). The FFs are used to store signals, but can only hold one bit each. For larger on-chip storage, Block RAM (BRAM) stores multiple kilobyte. The CLBs and BRAM units are placed on the chip in a grid and connected over an interconnect fabric that spans the whole chip. Additional off-chip, DRAM storage is provided on-board (gigabyte to terabyte) and accessible through IP cores, however, requires more clock cycles than the one clock cycle BRAM access.

3.2 Hardware Setup

For a test on real hardware, we use the Arty board with a Xilinx FPGA. The board is connected to a workstation over the network interface. To transfer data from and to the FPGA we use the UDP protocol. The FPGA is not used as a co-processor, but rather as the

whole integration system, thus it could be directly connected to real applications (cf. Fig. 1). We compare the FPGA implementation with the open-source software integration system Apache Camel [12] running on an HP Z600 work station, equipped with two Intel X5650 processors clocked at 2.67GHz with 12 cores, 24GB of main memory, running a 64-bit Windows 7 SP1 and a JDK version 1.7.0 with 2GB heap space listed in Tab. 2.

4 FROM PATTERNS TO CIRCUITS

The EIP from [11] are the "de-facto" standard for asynchronous messaging in EAI. In this section we map the integration semantics in form of EIP to hardware concepts by re-defining them for synchronous streaming with flow control (similar to [3]) and classifying the patterns according to their characteristics to three templates that are then synthesized to the hardware.

4.1 Basic Integration Semantics

The basic integration semantics are described by a message, a message channel, at least one protocol adapter, and an ordered set of transformation or routing patterns that process the message.

Messages on Hardware. A message consists of a unique identifier, for identifying the message and enabling provenance along the different processing steps, a set of name/value pair header entries containing meta-data about the message and the message body, i. e., the actual data transfered.

For the processing on an FPGA, a message is defined as a stream of bytes, which gets meta-data assigned on entering the FPGA via the network interface. In particular, we assign a unique message identifier and the length of the message.

Message Channel on Hardware. The message channels decouple sending and receiving endpoints or processors and denote the communication between them. Thereby, the sending endpoint writes data to the channel, while the receiving endpoint reads the data for further processing. Our message channel definition on hardware

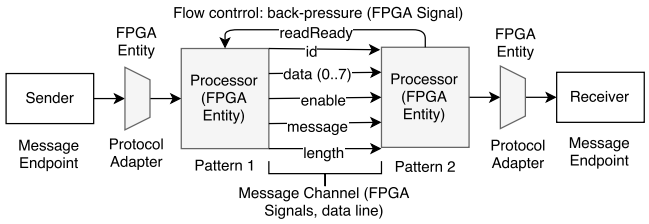


Figure 2: Basic integration aspects on hardware.

is depicted in Fig. 2. We use hardware signals and data lines to represent the control and data flow through a message channel. The channels contain a unique identifier as `id`, the message length as `length`, and the body as `data` of 8 bit chunks from the previously defined message over the data line (`data(0..7)`). To indicate that a message is sent over the channel, we added a message signal as `message`, which is set to one (i. e., `high`), when one message is sent - even if there is currently no valid data on the data line. The `message` signal is zero (i. e., `low`) only between messages (i. e., the channel is ready to receive another message). For the transport of the data to the subsequent processor we define an enable signal as `enable`, which is `high`, when valid data is on the data line and `low`, when there is no valid data on the data line. The `id` and `length` are separate lines, which are constant, when the message line is high.

The FPGA hardware is able to stream massively parallel using pipeline processing. However, for efficient processing, the hardware is limited to the resources on-chip (e. g., BRAM) and on-board (e. g., RAM). Our basic integration aspects are represented with on-chip resources to avoid latency and throughput penalties for the on-board resource access. However, we expect that in the future FPGAs can interact with less overhead with off-chip resources like DRAM or general-purpose CPUs. This will offer more freedom to use these off-chip resources, e.g. to buffer large messages.

Flow Control on Hardware Message Channels. The basic capability of an integration system to protect overload situations and data corruption is flow control. One technique used in connection-oriented wire protocols like TCP is *back-pressure*. Back-pressure allows the message processors (e. g., routing and transformation patterns) under high load to notify the sending operation or remote endpoint (e. g., via TCP back-pressure) about its situation. For instance, for TCP remote endpoints this could lead to the rejection of new connections.

On the FPGA, we define flow control similar to [3], which is exclusively used there for the synchronous communication between remote endpoints. For the back-pressure between message processors (i. e., no TCP support), we cannot reject messages atomically, because the stream might already be processed partially. Therefore, we decided for an approach with small FIFO queues in each processor that are used to buffer message data that cannot be immediately processed by the subsequent processor and thus ensure that no message data is lost. The receiving processor signals this by setting its `readReady` to low (cf. Fig. 2). The FIFO queues can be represented on hardware using flip-flops (FF), Block RAM (BRAM) or built-in FIFOs. Since FFs can only store one bit at a time and are very important for the logic of message processors, we chose BRAM. Although BRAM is a limited resources as well, it can be more easily extended by on-board DRAM to buffer larger messages. If the queue limit is exceeded, and the successor processor is not ready yet (i. e., `readReady` low), the current processor notifies its sender by setting its `readReady` to low.

4.2 Streaming Integration Patterns

Since the EIP are not defined for stream processing (cf. [21]), we define streaming semantics for the most relevant message routing and transformation patterns in practice (i. e., identified in [22]), and map them to circuits. For our example, we extend the EIP [11] by load balancer and join router.

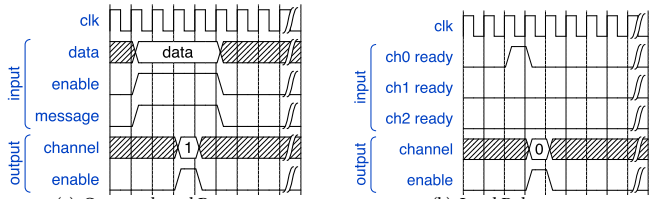
4.2.1 Routing Patterns. The routing patterns are used to decouple individual processing steps so that messages can be passed to different filters depending on a set of conditions [11]. We selected the most relevant patterns from [22]: content-based router, message filter and splitter, and add the aggregator, load balancer and join router used in the example scenario and in our evaluation.

Content-based Router. The content-based router and the message filter [11] are semantically similar. The filter is a special case of the router due to its channel cardinality of 1:1, while the router has 1:*n*. Both have a message cardinality of 1:1, are read-only (i. e.,

non-altering message access) and non-message generating (i. e., passing the original message).

Streaming Semantics. The router selects a message channel based on a condition that in worst case might have to fully read the message (i. e., requires buffering). In our approach the message corresponds to a data-based window similar to [8]. Alternatively, the message could be passed further into all leaving channels in parallel and filtered out later at a synchronization point. While the latter claims non-buffered streaming semantics the synchronization points cannot be set arbitrarily, which could lead to the same semantics as the message-window semantics in worst case. Hence, we use data-based windows as basis for our work.

On Circuit. Since, the leaving channel is selected based on a conditon evaluated on the stream, it specifies a mapping from a message to a channel. Figure 3(a) illustrates the semantics of our router design as waveform diagram, which shows high and low circuit settings for the different signals and data lines required for the pattern. The input denotes the message from the previous pattern. The output is the response from the user code. The clock cycles are denoted by `clk`. The `channel` is represented by an integer identifier. When a message enters the router (i. e., `message`, `data` high) and the data is valid (i. e., `enabled` high), then the condition (i. e., user code) is evaluated and the identifier of the selected channel is set together with the data valid signal `enabled`. According to the channel identifier the message will be routed.



(a) Content-based Router (b) Load Balancer

Figure 3: Router and Load Balancer patterns.

Load Balancer. The load balancer pattern – not in [11] – delegates a message to exactly one of several message channels using a load balancing strategy (e. g., uniform distribution). As the content-based router, it is read-only and non-message generating.

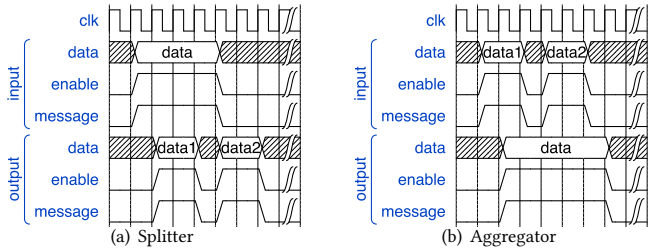
Streaming Semantics. For the purpose of this work, the load balancer is already suitable for streaming since it does not define any conditions on the message.

On Circuit. Similar to the content-based router, the load balancer maps from a message to a message channel. Only this time, the channel identifier is determined by a load balancing logic, configured through user code. Instead of a load distribution, we use the `readReady` signal used for back-pressue to determine which channel is currently available. The first channel available is selected as receiving channel. This semantics is shown in Fig. 3(b) as `ch0 ready`. Consequently the output channel is written and the data valid signal is set. The message will be routed accordingly.

Splitter. The splitter pattern [11] allows to process a message, if it contains multiple elements, each of which may have to be processed in a different way. Therefore the inbound message is split into a number of smaller messages according to a user-defined expression (i. e., message cardinality 1:n) with a channel cardinality of 1:1. While the splitter is message generating, it does not add new data to the n outbound messages.

Streaming Semantics. The splitter splits parts of a message into smaller parts similar to single elements of an iterable. There are splitter configurations with data structures of the form: head, iterable, tail. For each entry in iterable a new message is created by starting with the common head entries, one entry from iterable and the common tail entries. In these cases, the head has to be remembered and added before each element from the iterable. However, in case there is an element after the iterable, called tail, that has to be added, the streaming is limited. Thereby, the tail is unknown to the splitter until the end of the message, which means that the first new message would have to be buffered until the tail arrives. Similar to the router, a synchronization point could be used to add the tail part to each of the smaller messages. For the same reason as for the router, we use the buffered streaming option.

On Circuit. The splitter maps one message to multiple messages. Figure 4(a) illustrates the execution semantics for input and output. When the messages arrive and the data is valid, the user-defined split expression is executed, which leads to several messages. Thereby the splitter inserts a clock cycle, where the message signal is set to low and no data is sent inbetween the split messages.



(a) Splitter (b) Aggregator

Figure 4: Splitter and Aggregator patterns.

Aggregator. The aggregator pattern [11] combines a number of messages to one based on a time or number of messages based completion condition. Hence it has a message cardinality of n:1, a channel cardinality of 1:1, and is message generating, however, only combines data from the inbound messages to the one outbound message.

Streaming Semantics. By definition, an aggregator has to wait until the last message arrived. Only then the new message can be processed. Hence, the aggregator shows buffered streaming semantics on a streaming window, which is defined by the completion condition of the aggregator.

On Circuit. The aggregator maps from multiple messages to one message. Depending on the messages, the aggregator might only close the gaps between multiple messages or combine them in a new way (e. g., forwarding the common header of the messages and then the body of each one). In Fig. 4(b) data arriving in separate messages (i. e., `message` high) are combined to one message by setting the message signal to high as long as there are messages arriving that shall be combined to the outbound message.

Join Router. The join router is a new structural pattern – introduced in this work – that is needed to combine several control flows to one. This leads to a channel cardinality of n:1 and a message

cardinality of 1:1. This pattern is not in [11] and the data flow is combined using an aggregator, not the router.

Streaming Semantics. We define the router already suitable for streaming for our purpose, since it does not define any conditions or expressions on the message, but combines several streams to one.

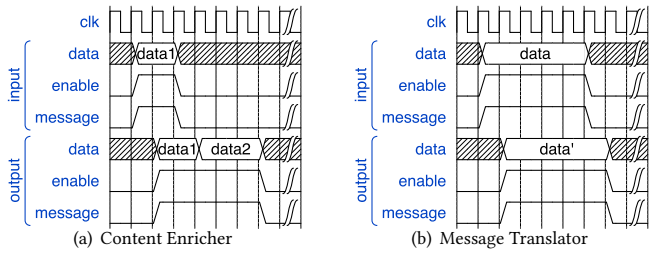
On Circuit. The join router maps from channel to channel, however, without any additional logic (e. g., as in the load balancer case). It simply checks, whether there are messages on the inbound channels and whether the outbound channel is free (i. e., `readReady` high).

4.2.2 Message Transformation Patterns. The transformation patterns are used to translate the message content to make it understandable for its receiving processor or endpoint [11]. We selected the most relevant patterns: content enricher and message translator, identified by a study on on-premise and cloud integration scenarios in [22]. All transformation patterns have a channel cardinality and a message cardinality of 1:1. They do not generate new messages, but modify the current one.

Content Enricher. The content enricher pattern [11] adds content to an existing message, if the message originator does not provide all the required data items. The enrichment can be done (a) statically, (b) from the message itself or (c) from an external source. In this work, we consider (a) and (b), however, the external data (c) could be provided on the on-board RAM.

Streaming Semantics. The current enricher semantics for (a) and (b) allow to fully stream this operation.

On Circuit. The enricher maps one message to another, while inserting data into the inbound message. Figure 5(a) shows the processing semantics for one message with `data1`, to which additional data is added by a user-defined expression as `data2`. Thereby, the message and the data valid signals are set to high.



(a) Content Enricher (b) Message Translator

Figure 5: Translator and Content Enricher patterns.

Message Translator. The message translator pattern [11] converts the structure of the inbound message into one understood by the receiver. This includes filtering content, which covers the content filter pattern [11].

Streaming Semantics. The current message translator definition covers streaming for simple cases (e. g., one to one assignments, data type operations). In addition, for many to one field translations, parts of the data have to be buffered for later lookup and assignment.

On Circuit. The translator maps one message to another one, while reorganizing the data in the inbound message using user-defined expressions. Figure 5(a) shows the behavior for the interaction with the user code, which returns the modified message content as `data'`. The message and the data valid signals are set to high.

4.3 Pattern Templates

Related approaches show that the hardware design is crucial for the performance of the resulting hardware scenarios [16]. To achieve good utilization of the FPGA hardware and high throughput we exploit commonalities between the patterns discussed in Sect. 4.2. Therefore we arrange them into three classes of behavior, which we call templates (similar to SQL-Query constructs like project and join in [18]). Unlike the original routing and transformation categories from [11], this classification is based on implementation criteria (i. e., mapping to FPGAs).

From Patterns to Hardware Templates. We build the categories for the classification along the observation based on the interaction with the user-defined conditions: predicates or expressions.

Expression Template (ET): The first template combines all patterns that conduct a "message to message" mapping. They mostly execute more complex expressions, which are provided by the user, while working directly with the data line. This applies to the splitter, aggregator, content enricher and message translator patterns.

Predicate Template (PT): The patterns that conduct a "message to channel" mapping, mostly execute simpler conditions like predicates. They set the `message` and `channel` signals. Candidates are the content-based router, the message filter, and the load balancer patterns.

No User Template (NUT): There is only one pattern in our selection that does not fit into the previous templates (and maybe not the only one). The join router conducts a "channel to channel" mapping and does not evaluate any user-defined conditions.

Putting it all together. With the basic integration semantics (incl. flow control) and the patterns categorized into templates, we give a conceptual view on how an integration pipeline and message processors can be synthesized to hardware. Figure 6 gives a conceptual overview for the three templates. The rectangles denote patterns or user code, the cylinders buffers and all straight directed edges denote message channels. The dashed edge returns `channel` and `enable` (cf. Fig. 3). The type converter Fig. 6(d) will be discussed in Sect. 5.

The ET patterns wire the data to the user code, where it is evaluated. The result is buffered in a FIFO queue to support buffered streaming patterns and deal with back-pressure.

The PT patterns require more system logic (cf. Fig. 6(b)). Hence, the input data is wired to the system code that executes the user code and also stores the messages in a FIFO queue for buffered streaming patterns and back-pressure handling. In addition, the user code does not return the modified message, but `channel` and `message` signals. Consequently, the data has to be forwarded from the buffer. The buffers are reset after the message was sent. For n outbound channels, the system code creates n buffers for wiring subsequent patterns.

The join router NUT, receives messages from n message channels at the same time, which are put into n FIFO buffers correspondingly. Figure 6(c) shows the NUT running a round robin fetch from the

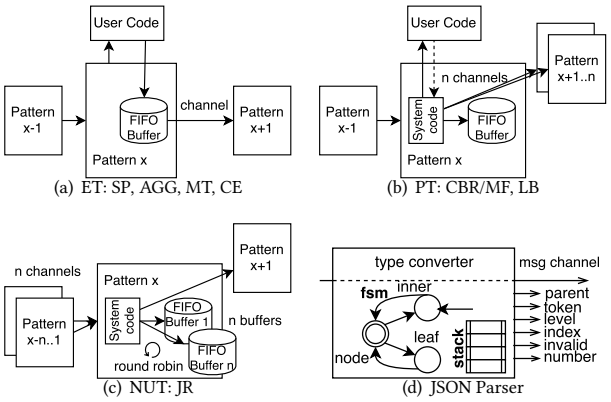


Figure 6: Conceptual view on pattern templates and hierarchical format processing.

FIFO queues. Then the messages are pushed further, and the buffers are emptied. The back-pressure technique is used to avoid several messages arriving at the same channel, while the buffer of this channel is not empty yet (cf. `writeReady` signal).

5 MESSAGE PROCESSING

The message processing is defined by predicates and expressions as user code. While we decided to transfer data as sequence of bytes, the data can have arbitrary message types (e. g., simple type like integer, or more complex like JSON). For instance, JSON messages can be evaluted by a JSONPath predicate. This in return can be implemented as an automaton [7, 29]. For predicates, we define one automaton for parsing the message (i. e., similar to a type converter [12]) and one for matching a set of conditions. In contrast, the data might be changed based on complex expressions, for which one type converter might be used either together with EIP-typed automata (e. g., a message translator state machine) or user-defined hardware code that constructs the output message.

5.1 Message Protocol Handling

In this paper, we focus on hierarchical message formats like JSON, thus implemented a streaming type converter that parses the data stream (Fig. 6(d)). To answer the general question in integration systems on "where to do the type conversions?", we placed the type parser in every user code that accesses the message. For instance, we placed the type converters between a pattern's system code and user code in ET and PT (cf. Fig. 6(a), Fig. 6(b)). The NUT has no user code, hence, no dependency on message types.

An alternative approach is having one type converter at the beginning of each flow, which would require less instances of the type converter (i. e., less LUTs, FFs). However, this converter requires that all message channels are of the same type, which reduces its flexibility in usage and consumes more resources due to bigger amounts of fully materialized data in the FIFO buffers and during processing. In our approach, we work with generic message channels that are not bound to one data type.

When parsing JSON messages, we assume an automaton with a JSON-specific alphabet (i. e., for tokens and nodes). For handling hierarchical messages, we define a deterministic automaton with a start state to an inner node (i. e., object, array) with transitions to a inner and leaf nodes, denoting simple typed values or complex types like object for arrays and name/value pairs for objects. Figure 6(d) depicts the outbound interface of the general type converter for hierarchical data sturctures. The `token` signal indicates the current token (e. g., string, quote, comma), which depends on the current state of the automaton. The `parent` signal gives the type of the parent node (i. e., object or array). It is set when a new inner node is encountered and the old value has to be pushed to a stack, which is popped when the new inner node ends. The `level` denotes a pointer in the hierarchical data structure and the `index` signal denotes the index of the node in the current level of the hierarchy. The level is increased and decreased as values are pushed and popped from the stack and the index is increased when a new node is encountered, and propagated to the stack together with `parent`. The `invalid` signal indicates a malformed structure. Only the `number` signal is specific to the JSON parser and passes a value for convenience, if the current token is a number. This design parses arbitrary hierarchical structures, while consuming less resources, e. g., compared to [15].

5.2 Predicates and Expressions

After the type conversion a predicate or expression can be executed. Although we mainly show the user code directly in VHDL for our evaluations, a general purpose JSONPath to VHDL translator - at least for predicates - is available. As illustration, List. 1 shows how a JSONPath predicate, matching `OTPRICE` lower than 100, 000, is generated to VHDL as a condition for a content-based router. While the code generated from the JSONPath serves as illustration, FPGA vendors provide alternative languages (e. g., Intels I++, former A++, to VHDL compiler) that could be used for formulating more complex expressions.

We assume a converted JSON message, which means that the signals `enable` (in the code shortened to `en`) and `data` are those of the message channel and the signals `token` (line 1) and `number` come from the type converter. The code in List. 1 is located inside a VHDL process, triggered by the clock that also drives the `data` and `enable` signals. The incoming data is compared to the string `otprice` (line 3) and if the whole string matched (line 6), the `field` signal is set high (line 8). In the clock cycle after the whole total price amount was read (i. e., the data is a comma that is the end of every line) the signal `number` is compared to 100, 000 (in List. 2, line 4). The default channel is 0 and if the number is smaller, the message is routed to channel 1.

Listing 1: Match field

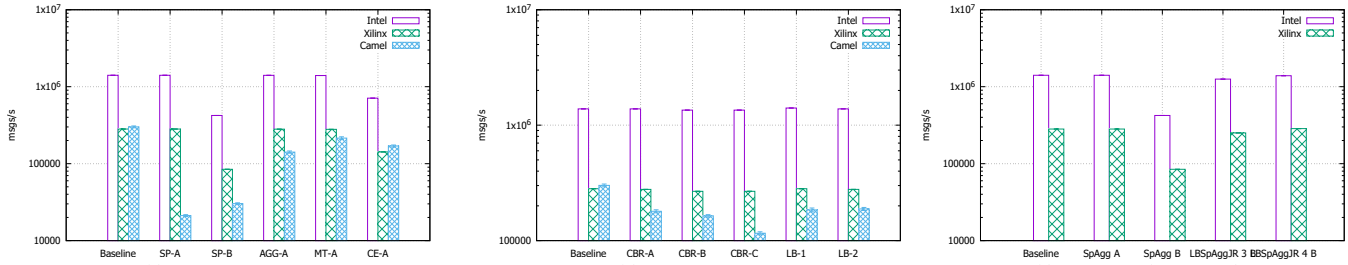
```
1 if en='1' and token=NAME
2 then
3   if data=otprice(index)
4   then
5     index<=index+1;
6   if index=otpriceLen-1
7   then
8     field<='1'; index<=0;
9 end if;end if;end if;
```

Listing 2: Eval. Cond.

```
1 if field='1' and en='1'
2 and data=COMMA then
3   channel<=0;
4   -- $[?(@.OTPRICE<100000)]
5   if number<100000 then
6     channel<=1;
7   end if; channelEn<='1';
8 end if;
```

6 EXPERIMENTS

In this section we evaluate the FPGA stream processing for application integration - represented by the three template variants (i. e.,



(a) ET: SP, AGG, MT (b) PT: CBR/MF, LB (c) Composed (FPGA only)

Figure 7: Template throughput for predicate and expression templates and composed patterns.

ET, PT and NUT) - assuming the FPGA can be used as network-attached integration system (cf. Fig. 1), e. g., as part of a company network or a cloud setup [5]. We selected the Arty educational board introduced in Sect. 3 for the hardware tests and compare some of the results with the open-source, software integration system *Apache Camel* [12] on CPU. For the CCT scenario benachmark, Intel provided us with the more "product-ready" Arria 10 SoC FPGA with 500 MHz clocks, 42,620 kB on-chip RAM, 1,006,720 registers and 251,680 adaptive logic modules, which we used in some of the other experiments as well. With this more powerful FPGA, we expect a linear increase of message throughput by the factor five higher clock speed. Camel runs on a HP Z600 work station, specified in Tab. 2. Besides verifying the feasibility and correctness of our approach, the main goal of the experiments is to perform throughput measurements, to study instance parallelization and resource consumption. Therefore we use the EIPBench pattern benchmark [22], which specifies benchmark configurations derived from "real-world" integration scenarios.

EIPBench is a "data-aware" micro-benchmark designed to measure the throughput of messaging patterns. Therefore, it specifies scale factors at pattern (e. g., number of condition and branching definitions) and at process level (e. g., number of concurrent users and message size). The message data sets are generated from the well-known TPC-H order to customer processing, but we only generate messages based on orders and customers (optionally embedded in order messages for message size scaling). To keep this paper self-contained, Tab. 3 summarizes the benchmark configurations from [22] for the benchmark definitons that are relevant for our evaluation and maps them to our benchmark identifiers (e. g., the EIPBench SP-B ↦ SP-A). We used the existing EIPBench definitions, however, added new benchmarks required for our analysis (cf. Tab. 3 without EIPBench representation). The hardware throughput for all benchmarks is measured with a simulator provided by Xilinx that uses post implementation simulation and element timing data of the FPGA as in [17]. First we study the message throughput on a single data stream with the same message size, and later we consider parallelism and message size, before we showcase our motivating scenario.

6.1 Pattern Throughput in Perspective

In this section we study the message throughput of our FPGA-based EIP that show better results than the software implementation.

Table 3: FPGA pattern benchmarks.

Benchmark	EIPBench[22]	Description
CBR-A	CBR-A	simple cond.: OTOTALPRICE < 100.000
CBR-B	CBR-B	multiple conds.: OTOTALPRICE < 100.000, OORDERPRIORITY = "3-MEDIUM", OORDERDATE < 1970, OORDERSTATUS = "P"
CBR-C	CBR-C	conds. on same fields as CBR-B, but multiple branches with different values
LB-x	LB-A	distributes messages over x routes.
SP-A	SP-B	split fields (iterable) into msgs.
SP-B	SP-C	split order fields (iterable) into separate msgs. while always adding head and tail
AGG-A	AGG-B	aggregate fields into msg. (SP-B reverse)
AGG-B	-	aggregate order entries into msg. (SP-C reverse)
JR-x	-	join router which joins x routes.
MT-A	MT-B	map names and filter entries according to a mapping program
CE-A	-	copy each entry, concatenate with a constant

Content-based router (CBR), load balancer (LB), splitter (SP), aggregator (AGG), join router (JR), message translator (MT), content enricher (CE).

Therefore we use the configurations from EIPBench for all considered patterns in this work and subsequently identify them by their abbreviations (cf. Tab. 3).

We measure the empty pipeline as baseline (i. e., w/o message processors) for all three FPGA templates and for the Camel using EIPBench order messages. The results are collected in Fig. 7, which shows that some of the FPGA patterns perform close to the baseline (i. e., near optimal).

Message and Content Generation. Although the splitter and aggregator are classified as routing patterns according to [11], they reside in ET template with the message translator and the content enricher. Figure 7(a) shows that the splitter SP-A performs close to the baseline, since emitting the same amount of data that it consumes. In the second, SP-B case, the splitter has to wait for the end of the message to be able to create the messages correctly and then emits the head with one entry from the iterable and the tail multiple times. The results for this data generating pattern are better than for Camel, however, the increasing amount of data reduces the throughput.

While the aggregators AGG-A (reverse SP-A) and AGG-B (reverse SP-B; similar to AGG-A, thus not shown) as well as the message translator MT-A on the FPGA perform close to the baseline, the content-generating CE-A content enricher case shows a similar effect as for SP-B. That is due to the CE-A enricher case essentially

duplicates the amount of data processed. While the Camel enricher is not limited by physical on-chip memory, the hardware enricher throughput reduces to half of the possible capacity (cf. baseline). The hardware throughput would be higher on bigger FPGAs. The high throughput of the aggregators are measured on the order data set, not the much smaller, split order messages, which are produced by SP-A (in average 11.73 B) and SP-B (in average 197.67 B). When testing these cases AGG-A performs at 8, 396, 946 msgs/s and AGG-B at 518, 134 msgs/s. This indicates that our approach saturates well up to the physical capacity limits.

Conclusions: (1) The message throughput of transformations (i. e., MT-A) and routings (i. e., SP, AGG) is much higher for all of the benchmarks, but the CE-A case. (2) Content generating patterns lead to degrading throughput on the FPGA due to the increasing messages sizes and saturation up to the resouce limits (e. g., BRAM). (3) The throughput scales linearly with more hardware resources.

Multiple routing conditions and branchings. Let us start with the question from EIPBench about the "impact of multiple conditions and route branchings" for the content-based router. The software implementation shows a decrease in throughput, especially when increasing the number of conditions and branchings in routing cases CBR-B and CBR-C (cf. Fig. 7(b)). On the hardware, all router implementations score close to the baseline due to the parallel processing capabilities of our approach and the underlying hardware support. Hence, as long as the conditions can be executed in parallel, neither multiple conditions nor branchings significantly reduce the throughput.

The same results were observed for the load balancer, which is based on the same PT hardware implementation. Independent of the number of branches, the load balancer shows results close to the baseline. These observations indicate a major benefit of hardware over current software designs for routing patterns (e. g., due to thread handling).

The third template, which is implemented by the join router pattern, again scores close to the baseline (not shown). It is independent of the number of branches it accepts, too.

Conclusions: (4) The hardware throughput is invariant to multiple, parallelizable conditions and route branchings. (5) The load balancer and join router perform near baseline.

Pattern composition. The previous results indicate that some variants of the splitter, aggregator load balancer, and join router can be combined to a composite message processing pattern [11] without much throughput penalty (i. e., perform close to the baseline). However, data generating patterns should be avoided. For example the scatter-gather [11] uses a multicast pattern [12] (not discussed), which copies the messages, thus increasing the amount of data. However, fork and join patterns that introduce no performance penalty like the load balancer and the join router are usually used to support optimizations such as "rewrite operator to parallel operator" or "rewrite sequence to parallel" [2]. Especially patterns with less message throughput (e. g., splitter, content enricher) might benefit from a parallel instantiation. To shed some light into this hypothesis we conducted the following experiments on the FPGA only: composite message processing for the simpler SP-A and AGG-A case (i. e., `SpAgg A`) and for the more complex B case (i. e., `SpAgg B`), as well as the "rewrite sequence to parallel" optimization with three (i. e., `LBSpAggJR 3`) and four (i. e., `LBSpAggJR 4`) parallel SP-AGG sub-sequences for case B.

Figure 7(c) denotes the results for these experiments, showing a near baseline throughput for `SpAgg A` case. The `SpAgg B` is dominated by the splitter, which reduces the throughput to the individual SP-B result. When adding a load balancer that distributes the messages to three instances of a splitter, aggregator pair, while afterwards joining their output messages using a join router (cf. `LBSpAggJR 3B`), the throughput can be increased significantly. With four splitter, aggregator pairs, the throughput is near baseline again.

Conclusions: (6) The patterns that duplicate data like the multicast reduce the throughput. (7) The "sequence to parallel" optimization works well on sub-processes that only temporarily work with more data.

Table 4: Resource Occupation.

Building Block	LUTs	%	FFs	%	BRAM	%
Translation	82	0.39%	187	0.45%	1	1%
Routing(1↦2)	246	1.18%	523	1.26%	4	4%
Join Router(2↦1)	193	0.93%	361	0.87%	2	2%
JSON Parser	752	3.62%	897	2.16%	0	0%
UDP Receiver	649	3.12%	437	1.05%	1	1%
UDP Sender	457	2.20%	234	0.56%	0	0%

This is the maximum space occupation of each building block with optimization turned off. This can be much less, when only a few features of the building block are used. BRAM is in 18 kB units.

Table 5: Resource Occupation CBR-A|SP-B.

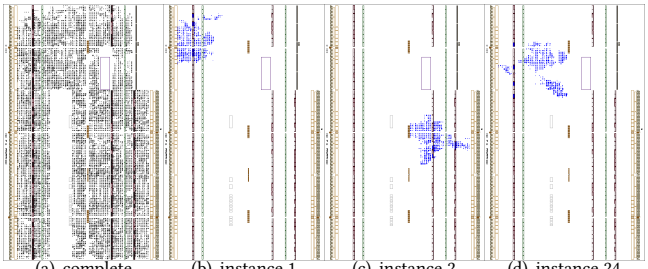
Building Block	LUTs	FFs	BRAM
user code (w/o parser)	101|1907	67|2977	0|0
JSON parser	504|396	193|159	0|0
messageRouter	246|-	523|-	4|-
messageTranslator	-|82	-|187	-|1
total	851|2385	783|3323	4|1
percentages	4.09%|11.47%	1.8%|7.99%	4%|1%

6.2 Parallelism: Space Management

The inherent support for parallelism is an advantage of FPGAs. When instantiating multiple integration scenarios in FPGA hardware, multiple message streams can be processed truly in parallel. The number of deployable scenario instances is determined both by the size of the FPGA, i. e., its resource capacity (incl. LUTs, FFs and BRAM), and by the capacity of the FPGA interconnect fabric.

Table 4 shows the resource occupation of the system code building blocks we explained earlier. One important limiting factor of these building blocks is the BRAM. Each FIFO queue in the building block uses one 18 kB BRAM block. For example a maximum of 25 routers can be placed on the Artix-7 chip we used for our hardware tests. Code that has a lot of state, like the JSON Parser, has a high occupation on LUTs and FFs. The resource occupation numbers in the table are obtained via the resource occupation analysis tool, of the Vivado IDE, that can be run on a synthesized and implemented design.

We placed one fully configured instance of the ET and PT templates on the Artix-7 chip, the SP-B, and CBR-A. Table 5 shows the resource usage differentiating the JSON parser, the user and template code. We also give the usage in percent of the total number of available resources. Note that there is a significant difference in size



(a) complete (b) instance 1 (c) instance 2 (d) instance 24

Figure 8: Resource usage on the FPGA chip (floorplan) by the template two space (i. e., CBR-A) and the remaining system components.

between the space required by the user code including the JSON parser (101+504, 1907+396 LUTs, respectively) and the space required by the system template code (246, 396 LUTs). This overhead indicates that the space consumption and the pattern performance hugely depends on the specific user code. Another interesting effect is the implicit optimization during synthesis to the FPGA (e. g., the reduction of the JSON parser to the features that are used).

The usage of parallelism brings forth another design trade-off characteristic of FPGAs. Due to their space occupation, the CBR-A can be instantiated 24× and the SP-B 8× on the Xilinx chip. To accommodate these instances, the VHDL compiler has to trade latency for space by possibly placing unrelated logic together into the same slice, resulting in longer signal paths and thus longer delays. This effect can also be seen in Fig. 8, where we illustrate the space occupied by three of the 24 CBR-A configurations (cf. instance 1, 2, 24). Occupied space regions are not contiguous, which increases signal path lengths. This effect has also been identified for predominantly asynchronous desings [3, 16], while our experiments did not show any negative impact on the message throughput for our mostly synchronous designs. In summary, with more on-chip resources (e. g., FFs, BRAM) a higher degree of scenario instance parallelism, and thus more overall throughput could be reached.

Conclusions: (8) The message protocol handling and the complexity of user code impact the space consumption. (9) The parallel processing through multiple instances is only limited by the FPGA's resources. (10) The on-chip signal path length does not have an impact on message throughput.

6.3 Parallelism: Performance

One of the important questions to answer for pattern implementations is "What is the impact of concurrent users?" [22]. To answer this question we used the SP-B and CBR-A configurations mentioned above to run up to 24 independent data streams in parallel. We call them processing units to allow comparison with the results of the software system from [22], where multiple threads were measured. Figure 9 shows the message throughput per second for an increasing amount of parallel processing units compared to the corresponding software implementation.

An important observation is that running additional process instances has no impact on the other instances, which let the processes be executed concurrently. Thereby the signal path length does not decrease the measured message throughput. Consequently, the throughput scales linearly with the number of process instances.

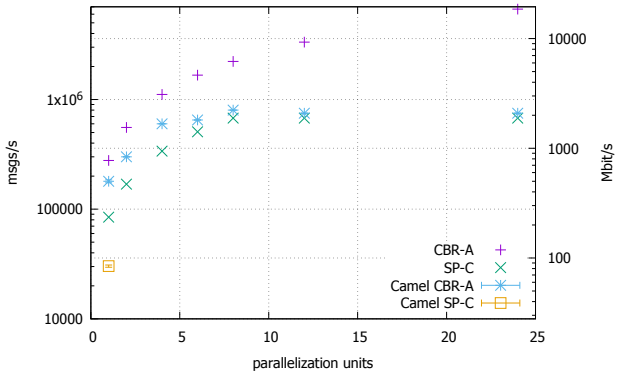


Figure 9: Concurrent user measurements.

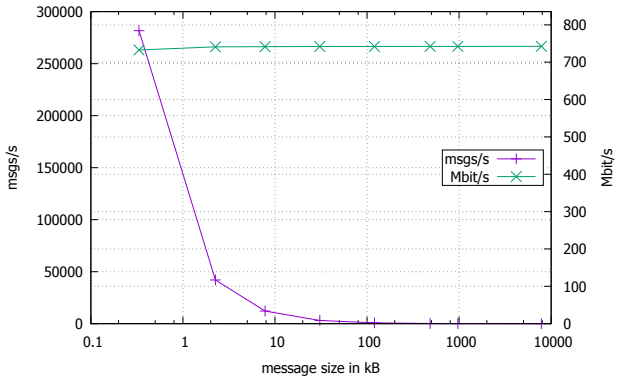


Figure 10: Message size baseline measurements.

The results for the patterns based on the two hardware templates show a saturation after their space limits are reached (i. e., for 8, 24 units, respectively). The multi-threaded software implementations, executed in a single JVM-process, cannot provide the same level of parallelism as an FPGA. This could be achieved with more JVM-processes on more CPUs, however, at a considerable expense (e. g., management of JVM or even VM instances, power consumption).

Conclusion: (11) The throughput scales linearly with the number of instances until resource saturation.

6.4 Message Size

The EIPBench specifies a scale factor for data size benchmarks to target the question on "What is the impact of message sizes?" [22]. Therefore we used the TPC-H order-based messages approximately up to 8 MB per messsage from EIPBench and evaluated them for all defined templates. Figure 10 depicts only one result, because all template baselines performed identical.

The immediate observation is that for increasing message sizes the memory bandwidth is saturated. In contrast to designs that use higher level memory (e. g., DRAM) and in-memory object materialization [17], which scales linearly with the input data size, our approach uses byte-streams using the local BRAM only for flow control (e. g., back-pressure). This design decision makes the approach directly limited by the practical upper boundary of resources on the board, which allows a throughput of approximately 800 MBit/s. For illustration, we added a secondary axis for MBit/s in Fig. 10, which shows an almost stable data volume saturation. We

think that for most of the IoT scenarios the many smaller messages should fit into the on-chip memory.

As an evolution of our design, similar to [17], we could aim to load the messages in RAM temporarily and only pass the message pointers and a message structure, e. g., containing a map of names in a JSON to pointers to their data, from pattern to pattern (cf. *Claim Check* [11]). This would allow for bigger messages, if the patterns only perform operations on small parts of the message (e. g., object lookup). For those cases pointers to larger messages could be transported, while keeping the processing fast. In case of many write and/or read operations, data has to be passed back and forth to the on-board RAM (compared to on-chip BRAM), which would decrease message throughput.

Conclusions: (12) The throughput is physically limited by the capacity of the hardware. (13) The throughput can be increased by using secondary memory, however, trading for more selective data operations.

6.5 Patterns to Scenarios: Connected Cars

Let us get back to the motivating connected car example. The data sent from the vehicles separates into approximately 304 B error code JSON messages with fields like `"Diagnostic_Trouble_Codes":"MIL is OFF0 codes"` and approximately 762 B telemetry data with fields like `"Vehicle_Speed":"0km/h"` and `"Engine_Load":"18,8%"` from the car's OBD device. The error codes are enriched with master data of the owner by lookup of the `obd2_Vehicle_Identification-_Number_(VIN)` and translated into the format understood by the receiver. For the telemetry data processing, the latter two steps are performed as well, while an additional message filter is added to consider driving cars only. The differentiation between error code and telemetry data is done by an initial content-based router and the two control sequences are combined by a join router, before enriching and translating.

Figure 11 depicts the message throughput for the sketched simplified implementation of the sample scenario divided into the paths taken through the scenario: error code, telemetry and filtered telemetry. While the measured throughput for the unoptimized processing (i. e., `normal`) is as expected, considering the single pattern results, some control- and data flow optimizations from related domains (e. g., data integration [2]) can be considered. Thereby, we exclusively focus on techniques that are especially beneficial for our hardware-based approach and the particular scenario. For instance, on FPGAs the flows are executed instantly and in parallel. Hence, optimizations like "Reschedule start of parallel flows" [2] (i. e., "start the slowest flow first") and "Merge parallel flows" [2] (i. e., "avoid forking costs"; no performance improvement, but space reduction possible), are not applicable or desirable.

The first control-flow optimization that looked promising was "sub-process parallelization" (i. e., `sub-parallel`) [2], which targets to exploit the FPGAs parallel processing, of the sequence: content enricher and message translator. This worked well for the error codes, however, not for the telemetry, due to the size of the data.

Since the data and not the control-flow seems to be the limiting factor in this scenario, message-flow optimizations are more promising. Merging the neighbor patterns (i. e., `neighbor merge`) requires less resources due to the removal of one channel. However,

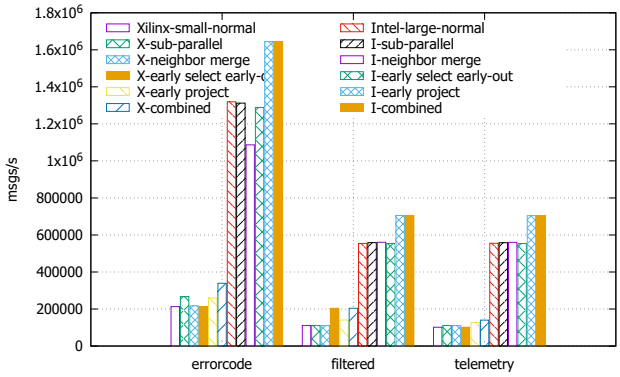


Figure 11: Connected cars scenario performance.

the freed space is not enough for another instance and the performance pentality of a channel is low, thus no significant throughput increase is measured.

The throughput of the filtering can be increased in all cases, when stopping the evaluation immediately, when the condition matches (i. e., `early select early-out`). This optimization worked well because significantly less data has to be processed.

The optimizations that work well for query processing are early-selection and early-projection (i. e., similar to [19]). The selection optimization (not shown) has no positive effect (i. e., throughput increase less than 30 msgs/s), because the message filter is not able to cancel a message transmission when the condition does not match. The early-projection places a content filter pattern (not discussed) to the beginning of the process that filters empty fields. The results in Fig. 11 for `early project` show an increase in throughput, since less data is moved through the hardware pipeline.

Lastly, we combined several of the promising optimisations like sub-parallel, early-out and early project into one scenario and measured the performance (i. e., `combined`). For the error codes and the telemetry the sub-parallel and early project optimizations increase the throughput, because the early project significantly reduced the amount of data and the sub-parallel parallelized the slowest part of the flow. For the filtered messages the early-out optimization caps the throughput. Seven parallel scenario instances with combined optimizations fit onto the FPGA.

Power Consumption. For this setup the FPGA can handle 153,061.22 msgs/watt, while the CPU on the Z600 processes 11,052.32 msgs/watt (i. e., measured CPU consumption only). The measurement follows the technique in [17], which measures the power consumption of CPU and FPGA. The power consumption P of a logic circuit depends linearly on the frequency at which it operates: $P \propto U^2 \times f$, with voltage U, frequency f. For the consumption of the FPGA, a power analyzer provided by Xilinx reports an estimated consumption of 1.0 W, and for the CPU the consumption lies between the Extended HALT Power and the Thermal Design Power around 95 Watt. Hence, the scenario can be handled by an FPGA with much less energy consumption compared to the CPU. If all US cars sent one message every second, this was a saving of approximately 92.78% of energy, which becomes a critical factor for most data centers[5].

[5]NRDC, Anthensis. Data Center Efficiency Assessment, visited 03/2017: https://www.nrdc.org/sites/default/files/data-center-efficiency-assessment-IP.pdf

Product-ready Hardware. Despite the promising results on Xilinx, we studied the CCT scenario on the Intel Arria 10 SoC FPGA. Figure 11 shows the message throughput for one scenario instance (cf. `Intel-large-normal`) as for Xilinx. The results are as expected due to the factor five higher clocking (i. e., 100 MHz Xilinx vs 500 MHz Intel), thus our design scales linearly and could cover CCT for the car traffic in most countries with one FPGA.

Conclusions: (14) Especially data flow optimizations that reduce the message sizes increase the throughput. (15) Besides the optimizations suitable for CCT, a more systematic study on hardware process optimizations and their combination is required that collects all control and data flow approaches, analyzes their applicability to hardware and their impact on the message throughput and space reduction.

6.6 Discussion

From these observations we conclude that FPGAs allow for an effective message processing based on the EIP streaming semantics. Despite the slow clock rate of our Xilinx FPGA (100 MHz), it achieves even higher throughput than powerful general-purpose CPUs, because FPGAs can implement fully concurrent data steaming pipelines. At the same time it consumes less power than a general-purpose CPU.

Additionally, we would like to discuss further non-functional topics including security aspects, exception handling, and monitoring of the hardware processing as they are required in advanced scenarios. While some security aspects can be handled on the transport protocol level, characteristics like message privacy either require additional on-chip IP cores or an on-board CPU (e. g., decryption). The exception handling in integration scenarios can be implemented with a "stop process" and "message retry" on exception processing [23]. As part of our future work, we want to investigate how the additional logic for the cancellation of a message in the hardware pipeline and a timed resend can be added. Eventually, monitoring capabilities on message and channel level are required for a more productive setting. Therefore, the statistics could be written to the on-board RAM and asynchronously fetched and processed by an on-board CPU.

7 SUMMARY AND OUTLOOK

In this paper we assess the potential of FPGAs as message processors for data-intensive operations in the context of application integration. Therefore we define compatible EIP message streaming semantics, along which we categorized the patterns into three templates that can be efficiently synthesized to hardware. In addition, we specify a lightweight component for hierarchical message format processing.

Our experiments illustrate how FPGAs help to improve message routing and transformation throughput on hardware compared to a comparable software setup (e. g., due to the invariance to number of branches and conditions). Our analysis also revealed some limitations for further research (e. g., data-generating patterns, message sizes). Furthermore, pattern instance parallelism can be used to scale the throughput to cover whole real-world integration scenarios on one chip by synthesizing multiple scenario instances.

We analyzed and discussed the applicability of existing optimization techinques from the application integration domain to an integration process synthesized on hardware. But we leave a more systematic analysis as future work.

Acknowledgments: We thank Intel for providing us with the Intel FPGA for the CCT scenario benchmark.

REFERENCES

[1] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *ACM SIGMOD*, pages 147–160, 2008.
[2] M. Böhm, U. Wloka, D. Habich, and W. Lehner. Model-driven generation and optimization of complex integration processes. In *ICEIS (1)*, pages 131–136, 2008.
[3] E. Caspi. *Design Automation for Streaming Systems.* PhD thesis, UC Berkley, Berkeley, CA, USA, 2005.
[4] S. Chaudhuri, U. Dayal, and V. Narasayya. An overview of business intelligence technology. *Communications of the ACM*, 54(8):88–98, 2011.
[5] F. Chen, Y. Shan, Y. Zhang, Y. Wang, H. Franke, X. Chang, and K. Wang. Enabling FPGAs in the cloud. In *ACM Conf. on Computing Frontiers*, pages 3:1–3:10, 2014.
[6] F. El-Hassan and D. Ionescu. A hardware architecture of an XML/XPath broker for content-based publish/subscribe systems. In *IEEE ReConFig*, pages 138–143, 2010.
[7] R. W. Floyd and J. D. Ullman. The compilation of regular expressions into integrated circuits. In *Annual Symposium on Foundations of Computer Science*, pages 260–269, 1980.
[8] M. Grossniklaus, D. Maier, J. Miller, S. Moorthy, and K. Tufte. Frames: data-driven windows. In *ACM DEBS*, pages 13–24, 2016.
[9] Z. Guo, W. Najjar, F. Vahid, and K. Vissers. A quantitative analysis of the speedup factors of FPGAs over processors. In *Proc. ACM/SIGDA Int. Symposium on Field Programmable Gate Arrays*, pages 162–170, 2004.
[10] D. Gyllstrom, E. Wu, H. Chae, Y. Diao, P. Stahlberg, and G. Anderson. SASE: complex event processing over streams (demo). In *CIDR*, pages 407–411, 2007.
[11] G. Hohpe and B. Woolf. *Enterprise integration patterns: Designing, building, and deploying messaging solutions.* Addison-Wesley, 2004.
[12] C. Ibsen and J. Anstey. *Camel in Action.* Manning, 2010.
[13] D. S. Linthicum. *Enterprise Application Integration.* Addison-Wesley, 2000.
[14] A. Mitra, M. R. Vieira, P. Bakalov, V. J. Tsotras, and W. A. Najjar. Boosting XML filtering through a scalable FPGA-based architecture. In *CIDR*, 2009.
[15] J. Moscola, J. W. Lockwood, and Y. H. Cho. Reconfigurable content-based router using hardware-accelerated language parser. *ACM Trans. Design Autom. Electr. Syst.*, 13(2):28:1–28:25, 2008.
[16] R. Müller and J. Teubner. FPGAs: a new point in the database design space. In *EDBT*, pages 721–723, 2010.
[17] R. Müller, J. Teubner, and G. Alonso. Data processing on FPGAs. *PVLDB*, 2(1):910–921, 2009.
[18] R. Müller, J. Teubner, and G. Alonso. Streams on wires - A query compiler for FPGAs. *PVLDB*, 2(1):229–240, 2009.
[19] I. S. Mumick, S. J. Finkelstein, H. Pirahesh, and R. Ramakrishnan. Magic is relevant. In *ACM SIGMOD*, pages 247–258, 1990.
[20] M. Najafi, M. Sadoghi, and H. A. Jacobsen. Configurable hardware-based streaming architecture using online programmable-blocks. In *ICDE*, pages 819–830, 2015.
[21] D. Ritter, N. May, and S. Rinderle-Ma. Patterns for emerging application integration scenarios: A survey. *Information Systems*, 67:36–57, 2017.
[22] D. Ritter, N. May, K. Sachs, and S. Rinderle-Ma. Benchmarking integration pattern implementations. In *ACM DEBS*, pages 125–136, 2016.
[23] D. Ritter and J. Sosulski. Exception handling in message-based integration systems and modeling using bpmn. *International Journal of Cooperative Information Systems*, 25(02):1650004, 2016.
[24] M. Sadoghi, H. Jacobsen, M. Labrecque, W. Shum, and H. Singh. Efficient event processing through reconfigurable hardware for algorithmic trading. *PVLDB*, 3(2):1525–1528, 2010.
[25] S. Singh and D. J. Greaves. Kiwi: Synthesis of FPGA circuits from parallel programs. In *16th Int. Symposium on Field-Programmable Custom Computing Machines*, pages 3–12, 2008.
[26] Solace Solutions. Solace message router. https://solace.com/, 2016.
[27] J. Teubner and L. Woods. *Data Processing on FPGAs.* Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2013.
[28] P. S. Vaidya, J. J. Lee, F. Bowen, Y. Du, C. H. Nadungodage, and Y. Xia. Symbiote: A reconfigurable logic assisted data stream management system (rladsms). In *ACM SIGMOD*, pages 1147–1150. ACM, 2010.
[29] L. Woods, J. Teubner, and G. Alonso. Complex event detection at wire speed with FPGAs. *PVLDB*, 3(1):660–669, 2010.