An FPGA Avro Parser Generator for Accelerated Data Stream Processing

Tobias Hahn[1] Daniel Schüll[2] Stefan Wildermann[3] Jürgen Teich[4]

Abstract: Big Data applications frequently involve processing data streams encoded in semi-structured data formats such as JSON, Protobuf, or Avro. A major challenge in accelerating data stream processing on FPGAs is that the parsing of such data formats is usually highly complex. This is especially true for JSON parsing on FPGAs, which lies in the focus of related work. The parsing of the binary Avro format, on the other hand, is perfectly suited for being processed on FPGAs and can thus serve as an enabler for data stream processing on FPGAs. In this realm, we present a methodology for parsing, projection, and selection of Avro objects, which enforces an output format suitable for further processing on the FPGA. Moreover, we provide a generator to automatically create accelerators based on this methodology. The obtained accelerators can achieve significant speedups compared to CPU-based parsers, and at the same time require only very few FPGA resources.

Keywords: Avro, parsing, FPGA, semi-structured data, accelerator

Acknowledgement: This work has been supported by the German Science Foundation (Deutsche Forschungsgemeinschaft, DFG) as part of the Priority Programme SPP 2037.

1 Introduction

Many Big Data applications in areas such as the Internet of Things and Industry 4.0 are not only confronted with large volumes of data generated at a high frequency but also place high demands on the latency for analyzing this data. In this area, stream processing is becoming increasingly important, which means that data is continuously processed and analyzed as soon as it is generated or received. To meet the growing demands of stream processing applications in terms of high throughput and low latency, FPGA accelerators have been proposed as a solution in the past [MTA09; TM11]. Since stream applications typically run for long periods, the relatively long synthesis times required to generate application-specific accelerators are tolerable as they enable perfectly tailored and thus very resource- and energy-efficient data processing.

For being able to build such FPGA accelerators, different approaches to compile queries to FPGA primitives have been presented in the past [MTA09; MTA10; Sa12]. However,

[1] Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Erlangen, Germany, tobias.hahn@fau.de
[2] Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Erlangen, Germany, daniel.schuell@fau.de
[3] Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Erlangen, Germany, stefan.wildermann@fau.de
[4] Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Erlangen, Germany, juergen.teich@fau.de

doi:10.18420/BTW2023-46

these approaches ignore the fact that the data arriving at the FPGA is usually formatted in ways that are difficult to process directly on the FPGA or even by machines in general. As a matter of fact, parsing such data may take most of the time when being processed on a CPU. For example, in case of JSON, it has been shown that parsing may account for around 90% of CPU time for some stream processing applications [Li17].

Offloading parsing to an FPGA would offer two major advantages. First, would relieve the CPU from this time-consuming task so that it can be better utilized by other tasks or workloads. Second, this would be particularly advantageous when FPGAs can directly access the data stream, e.g., when attaching them to a network interface or in the form of FPGA-based smart NICs, as additional data movements could be avoided. However, this requires concepts to parse the serialized data format as a byte stream.

In this realm, two approaches to parsing JSON data on FPGAs have been presented recently [Da22; Pe21]. However, the presented approaches only consider the acceleration of the parsing process, thus supporting only traditional software-based data processing. Consequently, they parse the received data into data structures that are tailored to be further processed on a CPU. In contrast, the approach presented in this paper aims at enabling complete data processing on FPGAs.

We specifically target parsing, selection, and projection of Avro objects which are widely used in Apache-based computing infrastructures. The Avro format [Ap21] has a much higher information density than semi-structured formats such as JSON. It is better tailored to FPGA-based processing than to CPU-based processing, as techniques to increase the performance of modern CPUs, like branch prediction, multi-level caches, and SIMD instructions, are not of any benefit when parsing Avro data. In this paper, we present general techniques for parsing data streams consisting of Avro objects on FPGAs. In addition, we introduce a methodology to automatically generate hardware accelerators for parsing, selection, and projection on FPGAs based on user-defined Avro schemas and queries. We present a system architecture where accelerators generated with the methodology can be loaded to parse data streams of Avro objects at line rate, for example, arriving at a network interface. Thanks to a fixed data layout, the outgoing data stream can even be forwarded to other FPGA accelerators and used to process further application steps. Moreover, due to low resource consumption, sufficient resources are often remaining available on the FPGA to build further accelerators for subsequent processing of the parsed, filtered, and projected data stream.

The paper is organized as follows: First, in Sect. 1.1, we will give a brief overview of common semi-structured data formats and justify our choice of Avro. In Sect. 2 we will introduce our parser generator and describe how an accelerator can be created given a schema and a path expression. In Sect. 3, the obtained accelerators are evaluated using the Yahoo [Ch16] and RiotBench [SCS17] benchmarks. Finally, the paper closes in Sect. 4 with a conclusion and an outlook for future work.

1.1 Data Formats for Stream Processing

Despite the huge overheads that parsing semi-structured data formats imply, there are many good reasons why these formats are nonetheless used universally today. These can be broken down to being human readable, having a relatively small data footprint, the possibility to enforce a fixed schema, and the possibility of extending the format while maintaining forward and backward compatibility *(schema evolution)*.

Semi-structured data formats can be roughly divided into *storage formats* and *exchange formats*. Storage formats arrange objects in such a way that they can be quickly searched for individual attributes, e.g., by using column-oriented formats. Examples are Apache Parquet [Ap22b], Apache ORC [Ap13], and Google Dremel [Me10]. The exchange formats on the other hand are easy to be serialized which makes it possible to write and process objects continuously as a data stream. Examples are JSON [Br17], Apache Avro [Ap21], and Google Protocol Buffers [Go22]. In the following overview, we focus on exchange formats, as we aim to accelerate data stream applications.

JSON, CBOR & Protobuf Tab. 1 gives an overview of the most common exchange formats. For each format, it rates the (a) readability by human and by machine and (b) the data footprint. Here, also the footprint for encoding the same record is displayed. List. 1 shows this record formatted in JSON. Finally, the table rates (c) schema evolution and (d) whether the format uses a schema. The most commonly used exchange format is the text-formatted JSON [Br17], which particularly stands out due to its high human readability. However, this in turn has a severe negative effect on its data footprint. In addition, JSON can also be used to achieve a good backward and forward compatibility by simply adding new attribute fields in newer versions. CBOR [BH20] is also a schema-less format where fields are binary encoded, resulting in a smaller data footprint, while still allowing attributes to be added or omitted as desired. Protobuf, on the other hand, relies on binary encoding and a schema that can be extended without corrupting older parser versions. This is achieved by assigning an index to all attributes in the schema so that the old parser versions can simply ignore indexes they do not recognize.

Tab. 1: Overview of the strengths, weaknesses, and general properties of data formats discussed.

	readability		footprint (car ex. size)		schema evolution	schema used
	human	machine				
JSON	++	- -	- -	(96B)	++	no
CBOR	- -	+	-	(50B)	++	no
Protobuf	-	++	+	(18B)	++	yes
Avro	-	++	++	(12B)	+	yes

```
{
  "id": 42,
  "name": "Golf",
  "engine": {
    "serialNr": 1234,
    "horsepower": 85.5
  }
}
```

List. 1: Motivational JSON record.

```
{
 "name": "car",
 "type": "record",
 "fields": [
   {"name":"id",    "type":"int"},
   {"name":"name",  "type":"string"},
   {"name":"engine","type": {
     "name": "engine",
     "type": "record",
     "fields": [
      {"name":"serialNr",  "type":"int"},
      {"name":"horsepower","type":"float"}]
}}]}
```

List. 2: Avro schema for a car object.

Avro Avro [Ap21] is a binary schema-based data format. Unlike Protobuf, Avro does not use indexes to identify fields and consequently requires even fewer bytes for encoding. The type and order of the fields are defined solely by the schema used. Accordingly, the Avro object encoding itself does not support schema evolution, as decoding always requires the corresponding schema. Instead, Avro is usually used in combination with wrapper formats, as presented in Sect. 1.1, to solve schema evolution at a higher protocol layer.

In the following, the Avro specification [Ap21] will be presented in more detail. Avro offers a range of elementary and complex data types. As elementary data types, booleans, signed integers (32-bit), signed longs (64-bit), floats (32-bit), doubles (64-bit), strings, and byte sequences of fixed and variable length are available. Avro includes records, enums, arrays, maps of key-value pairs, and union types as complex types. The Avro schema to be used is specified in JSON. List. 2 shows the Avro schema that complies with the JSON record from List. 1.

In terms of parsing speed, Avro outperforms all other formats. JSON is the most time-consuming to parse due to its text-based format. Unlike Protobuf and Avro, both JSON and CBOR cannot be parsed using finite state machines due to the arbitrarily deep nesting of records, making parsing again more complex. Since no indexes or attribute names are required for reading, parsing Avro has the advantage over Protobuf that only the sequence of fields defined in the schema has to be processed. However, Avro parsing does neither require complicated control flow nor flexible memory accesses and thus is not the field for which modern general-purpose processors with their sophisticated branch prediction and multi-level cache hierarchy have been optimized for. Nonetheless, a required simple finite state machine can be easily mapped to FPGAs with very low resource requirements. In this paper, we show in this paper how the generation of such logic circuits can be performed completely automatically solely based on the specified schema and a query defining projection and selection in a path expression.

Avro Container- & Wire-Format The Avro format can be used both for storing data on a hard drive and for transferring data over a network. In most cases, two wrapper formats are used, which are tailored to the respective case. For the storage of Avro objects, there is the Avro Object Container Format, which stores the used schema directly beside the data. A particular advantage of this format is the partitioning of the objects into blocks, which enables an efficient separation for parallel processing.

The wire format, on the other hand, is optimized for the transmission of data over a network. The Avro object stream is organized in a series of buffers. Each buffer begins with a four-byte length field that specifies the buffer length in bytes. The respective number of subsequent bytes contain Avro objects. A message may contain multiple buffers. The end of a message is indicated by a buffer of length zero, i.e., only a length field with value 0, and no buffer data is transmitted. In contrast to the container format, the schema is not transmitted here. Instead, the receiver must already be familiar with it, e.g., Apache Kafka uses a registry to resolve schemas of incoming data.

1.2 Related Work

Extensive literature already exists for processing XML data on FPGAs [EI10; Mi09; TWN12; WA11]. XML parsing on FPGAs has always been accompanied by path expressions for projections to reduce the amount and complexity of the outstream data. As Koch et al. [KSS08] have shown, projecting XML data can be solved most efficiently by transforming the problem into a string matching problem. However, this makes it difficult to transfer findings from XML parsing to parsers for binary encoded formats like Avro.

Recently, research attention has shifted to the JSON format which is predominant today. However, research on JSON parsing on FPGAs is still scarce, and furthermore, there is no solution for integrating FPGA parsers with existing accelerators for subsequent query processing on the FPGA. Peltenburg et al. [Pe21] propose to speed up JSON parsing by converting the input data to the columnar in-memory format Apache Arrow [Ap22a]. The FPGA performs the parsing of the JSON data, converts the data to the Arrow format, and then transmits it to the host memory. The authors implement their parser with a one-to-one relation between fields in the schema and parser blocks, which hence expects the same schema at all times. However, there is no fixed schema for JSON, so valid data in terms of the JSON specification can cause the parser to enter an invalid state. PipeJSON [Da22] follows a more flexible approach where the JSON data is converted into a tape structure, similar to CPU-based parsers. This tape structure consists of a variable-length array of 64-bit values, which contains the decoded values as well as offsets for navigating through the array structure (e.g., to the end of a record). While such flexible data structures can be processed efficiently on the CPU, they are unsuitable for further processing on the FPGA. Hahn et al. [Ha22; HWT22] present acceleration techniques for raw filtering of JSON data on FPGAs. Raw filtering is a selection technique on the raw byte stream of serialized JSON data. As such, it does not require to parse JSON records completely, but only to identify specific patterns according to the filter expression in the byte stream. However, since only irrelevant JSON records are filtered out and the data is not completely parsed, it does not directly enable any further processing of a given data stream on the FPGA.

2 Proposed Parsing Architecture

In this section, we present a technique for automatically generating hardware parser accelerators for a given schema and query (specified by JSONPath [Gö07]). The schema defines all elements that appear in each Avro object. JSONPath defines the selection criteria and attributes to be projected per Avro object. The parser generator goes through three phases. In the first phase, an object parser is created that can parse incoming Avro objects (see Fig. 1 left). In the second phase, this object parser is then extended to a Parse, Project & Select (PPS) module by augmenting logic for projection and selection (see Fig. 1 right). During the third phase, this PPS module is then wrapped into an accelerator which can later be deployed on the FPGA.

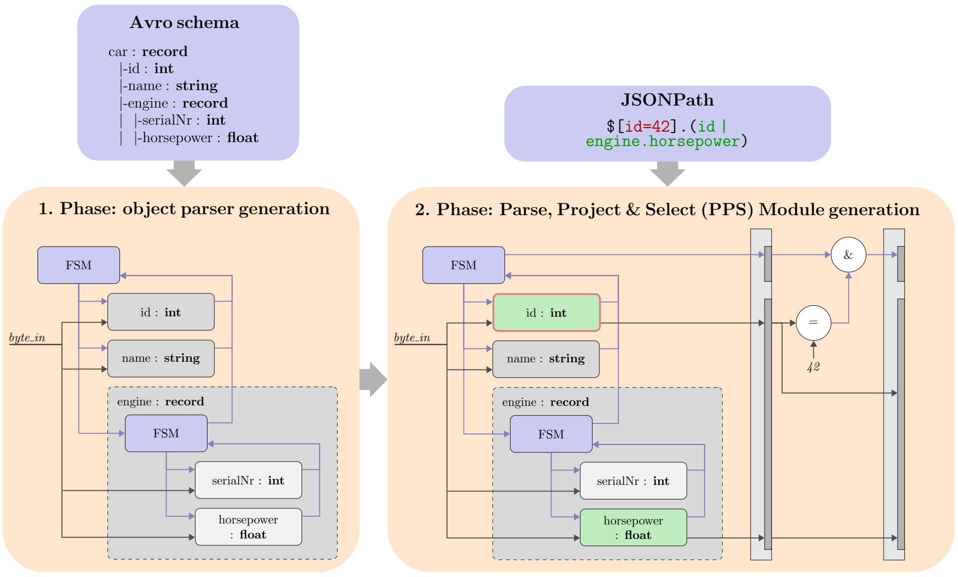


Fig. 1: Overview of the first and second phase of the parser generation process.

In the *first phase*, the schema is interpreted and a corresponding object parser is generated. For this purpose, the schema is traversed recursively. A parser block module is instantiated for each field of the schema. A finite state machine coordinates which field in the schema, respectively which parser block module is responsible for parsing the incoming byte at each clock cycle.

Example: Fig. 1 (left) schematically illustrates the parser structure generated for the schema given in List. 2. It contains blocks for `id`, `name`, and `engine`, as well as an FSM that coordinates the sequential parsing of the respective blocks. The latter is a complex (recursive) type, which again consists of internal parser blocks as well as an FSM to hierarchically coordinate the parsing procedure. Sect. 2.1 explains the details of parser block generation.

In the *second phase* (see Sect. 2.2), the PPS module is generated based on the object parser from the first phase and the specified JSONPath query, which defines the attributes to be projected and the selection criteria. First, all parser blocks are determined that provide the attributes required for selection or projection and their output signals are connected to a register stage (*stage 1*). The register stage also contains a valid bit which indicates whether the object was read completely and that the data in the register represents a valid Avro object. Subsequently, a further pipeline stage (*stage 2*) is generated containing the logic for evaluating the selection expressions by comparisons on the values stored in the stage 1 register `and` combining the result with the valid bit of the stage 1 register.

Example: Fig. 1 (right) shows an example query with a selection on field `id` and projection of attributes `id` and `horsepower` together with the generated parser accelerator. Only the signals of the parser blocks responsible for parsing the respective attributes get connected to the register stages. The selection logic for `id == 42` is added between the register stages.

In the *third phase* (see Sect. 2.3), an accelerator is generated from the PPS module created in the second phase, which can finally be deployed on the FPGA. One of the challenges of parsing Avro data is that some of the elementary data types are encoded with variable lengths. This makes it very difficult to parallelize generated hardware components to process multiple bytes in one cycle since in order to interpret a byte, it must first be clear which field in the schema it corresponds to. Therefore, we choose to process only one byte per cycle with the introduced object parsers and PPS modules. During the third phase, however, we again introduce parallelism by inserting multiple parallel PPS modules in the accelerator. For this purpose, we exploit the properties of the wire format to split Avro objects among parallel channels, while working with a higher word width.

2.1 Phase 1 – Object parser generation

Avro schemas are composed of different elementary and complex types. In the following, we present parser blocks for each Avro type, excluding the null and array type. Parser blocks can likewise be divided into *elementary parser blocks* (`fixed`, `float`, `boolean`, `int`, `enum` & `string`) and *complex parser blocks* (`record`, `map` & `union`). Elementary parser blocks are the basic blocks that interpret the input bytes to parse the desired fields. Complex parser blocks, on the other hand, are composed of one or more parser blocks (both elementary & complex) and do not interpret any input data but merely coordinate when which of the contained blocks is *active* and when its output is *valid*.

All parser blocks follow the same interface as illustrated in Fig. 2. This consists of an input `in_byte`, which contains the current input byte of the Avro object and is directly connected across all parser blocks. The port *in_valid* controls which parser block is active. The activated block accordingly interprets the obtained data on the port `in_byte`. If a parser block finished reading an encoded field, the port *out_valid* is set to one. The signal *out_value* remains still unconnected during the first phase and is subsequently connected to the register stages during the second phase in case it is required for selection or projection. Next, the details for generating parser blocks for covering all supported Avro types.

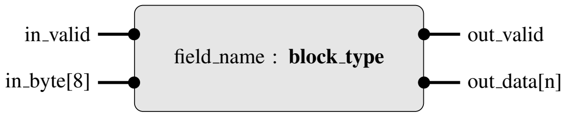


Fig. 2: Interface of a parser block.

Fixed parser block (`fixed`) The `fixed` parser block reads a fixed amount of n bytes, which is statically defined by the given schema. The block is controlled based on a counter which is initialized with $n - 1$ and decremented in each clock cycle the signal *in_valid* is one. Each input byte is written into a shift register of size $n - 1$ bytes. When the counter reaches zero, the complete field is available and a one is emitted on *out_valid*. The signal *out_value* with n bytes is then composed of the concatenation of the signal *in_bytes* and the $(n - 1)$ bytes in the shift register.

Float/double and Boolean parser block (`float` and `boolean`) In the Avro format, floating point numbers are directly encoded in the IEEE 754 format. Accordingly, a `float` parser block is just a special case of a `fixed` parser block of constant size n, which is 4 bytes for floats (single precision) and 8 bytes for doubles (double precision). No further binary format conversion is necessary by this parser block. The same applies to the `boolean` parser block, which is always encoded using one byte and therefore implemented as `fixed` parser block with a constant size of 1.

Int/long and enum type parser block (`int` and `enum`) Avro integers and longs are encoded via the zigzag format, which allows for a small data footprint. The zigzag format is a variable-length quantity code in which both small positive and negative numbers can be encoded to fewer bytes. This is achieved by reserving the eighth bit of each byte to indicate whether there are more bytes to follow, as can be seen in the example integer in Fig. 3. The remaining parts carry payload bits. The task of the `int` parser block is to take the zigzag formatted integer byte by byte and to decode it into the two's complement.

For the sake of clarity, however, let's first consider a zigzag-formatted number z without byte boundaries or continuation bits. The decoding of this zigzag encoded number z into its corresponding two's complement b of constant size n depends on whether a positive or negative number is encountered. Accordingly, the first step is to extract the sign of the number to be decoded, which is located in bit 0 of the least significant byte ($z[0]$). Subsequently, the decoding is done via

$$b = \begin{cases} z >> 1 & \text{if } z[0] = 0 \text{ //positive} \\ \sim (z >> 1) & \text{else //negative} \end{cases}$$

where $>>$ is a non-arithmetic right shift (i.e., 0 padding) and $\sim$ is a bitwise inversion.

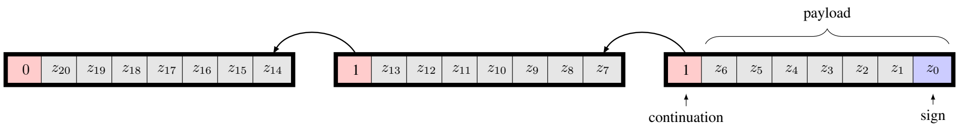


Fig. 3: Example of a 3 byte zigzag integer.

As the decoding is carried out byte by byte, the sign is extracted at the start of the first step $k = 0$. If a 1 is observed, the six payload bits z_6 to z_1 of the first byte are inverted. The result whether inverted or not is then written into the lowest 6 bits of the destination register ($b_0 \ldots b_5$). Since the obtained number can be smaller than the bit width of b, all higher bits ($b_6 \ldots b_{n-1}$) must be initially set to 1 in case of a negative sign, resulting in a sign extension. Each following byte $k > 0$ is treated as follows: 1.) Invert the 7 payload bits in case of a negative sign. 2.) Write the result into the next 7 bits of the destination register ($b_{6+(k-1)*7} \ldots b_{6+k*7-1}$). The b register is then connected to *out_data* and, as soon as a continuation bit is 0, the signal *out_valid* is set to 1.

The size of b, respectively of the signal *out_data* can be configured freely for the `int` parser block. Although 4 bytes are always used for integers and 8 bytes for longs, integer parser blocks are also used internally for parsing enums and metainformation fields of other types (see string, maps, union parser blocks below), where smaller sizes may be sufficient. Thus the parser blocks can be constructed as small as possible. An enum block is accordingly implemented as an int block with a b register size of n equal to $\log_2$(# enum elements) bits.

String parser block (`string` and `count_byte`) The string parser block consists of an `int` parser block and a `count_byte` block which are processed one after the other as shown in the flowchart in Fig. 4. As soon as the integer length field *len* is parsed, the received value is used to initialize the `count_byte` block. This block is basically a counter that remains active for *len* subsequent valid bytes, with the current input byte written to a shift register (similar to the `fixed` parser block). The length of the shift register, and thus the maximum readable string length, is set to 32 bytes by default but can be altered in the second phase by specifying a maximum string length in the JSONPath.

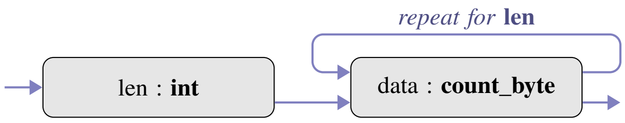


Fig. 4: FSM for the string parser block.

Record parser block (`record`) A record type contains multiple fields each being of a specific Avro type. The record parser block, therefore, contains one parser block for each field. Once *in_valid* is set to one, a controller consecutively activates the contained parser blocks for evaluating the input bytes, as shown in Fig. 5. The signal *out_valid* is set to one on completion of the last block.

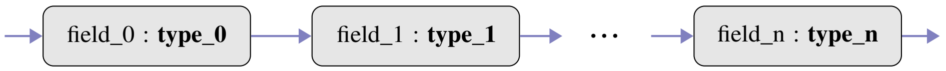


Fig. 5: FSM for the record<type_0, type_1, . . . , type_n> parser block.

Map parser blocks (`map`, `key_value` and `string_matcher`) The map type is encoded by an int field *obj_cnt*, followed by as many key-value pairs as specified in the int field. Once the *obj_cnt* field is parsed, it is used to initiate the loop counter to control the `key_value` parse block, as illustrated in Fig. 6. The `key_value` parse block sequentially parses first a key, which is encoded as a string, and then the respective value, which type is defined in the schema. As long as the loop counter is greater than 0, the `key_value` parser block is kept active, which repeats its internal parsing. The loop counter is decremented each time the signal *out_valid* of the `key_value` parser block is one, i.e., after each parsed key-value pair. The `key_value` parser block is a special case of a record parser block with only two fields, where the first field is a `string_matcher` block. This is a special block that can test whether the parsed string matches a given string from a dictionary. The rationale for this is that a query may use values from a subset of keys for projection or selection. This means that only key-value pairs with keys from this set have to be registered for the next pipeline stage. Therefore, the second phase of parser generation will populate the string matcher dictionary with each key string in this subset and augment the corresponding matching logic. The string matcher will generate one signal per key in this set to indicate when the respective key-value pair is currently parsed (see Sect. 2.2 for more details). According to the Avro specification, the specified length of the map can also be negative. In this case, the absolute value of the length does not indicate the number of objects, but the length of the payload data in bytes. However, this behavior is not supported in our parser block but could be added in the future by modifying the loop counter.

Union parser block (`union`) The Avro union type is a complex type. It is possible to specify a list of several different types in the schema for a union field. The data contained in that field of an Avro object can then be of one of these types. This is achieved in Avro

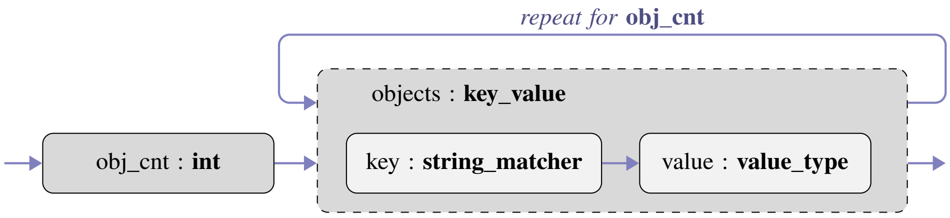


Fig. 6: FSM for the map<key_type, value_type> parser block.

by encoding an `int` index before the payload to indicate which type should be used for encoding the field. The index can then be used to select the desired type from the given list of types during interpretation. Accordingly, the `union` parser block is composed of an `int` block followed by all types specified in the schema, respectively their associated parser blocks, as seen in Fig. 7. Once the parsing of the *union_index* int block is complete, only the input of the respective `type_⟨union_idx⟩` block is activated. The signals *out_valid* of the parallel blocks can simply be combined via an or-reduction, as only one of the blocks can become active anyway.

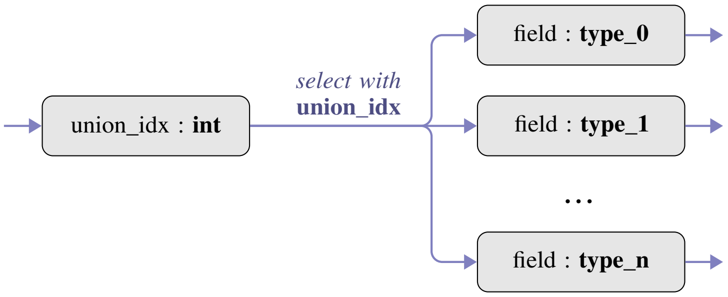
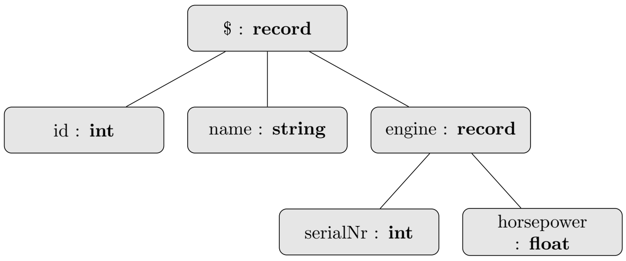


Fig. 7: FSM for the union<type_0, type_1, . . . , type_n> parser block.

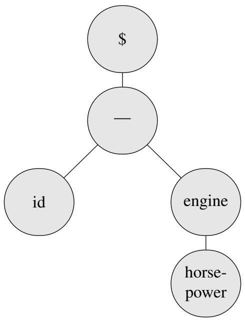
Object parser generation The parser generation consists of allocating parser blocks according to the given schema and generating the control logic of the FSMs (one for each complex parser block) to coordinate the parsing process. Each Avro schema can be represented by a *object parser tree* $G_O\{V_O, E_O\}$ with vertices V_O and edges E_O. The leaves are elementary types. Complex types have multiple children. The child nodes are ordered in the order they appear in the given schema. In the object parser generation phase, the respective Avro parser tree is first generated from the given schema. Then, this parser tree is traversed depth-first in the given order. At each node, a parser block of the respective type is generated. As an example, Fig. 8a specifies the Avro parser tree of the schema in List. 2. Fig. 1 illustrates the parser structure generated for this example. The FSMs are initialized to schedule the parser blocks in the specified order.

2.2 Phase 2 – Selection and projection logic generation

The second step deals with creating the Parse, Project & Select (PPS) module based on the object parser by adding the hardware for selection and projection, given by a JSONPath expression. The JSONPath expression specifies all attributes that are relevant for projection and selection. A dollar sign at the beginning represents the root (or start) of an Avro object. Elements are separated by dots. If several attributes are required, they can be concatenated with a `|`. In addition, the JSONPath expression contains the selection criterion consisting of comparisons of attributes that can be combined via Boolean expressions. The JSONPath expression spans a tree, which we denote by *path tree* $G_P(V_P, E_P)$ in the following, where the root represents the start of the Avro object and the leaves the attributes of interest. Each path in this tree also has a corresponding path in the object parser tree.



(a) Object parser tree G_O for the schema in List. 2.



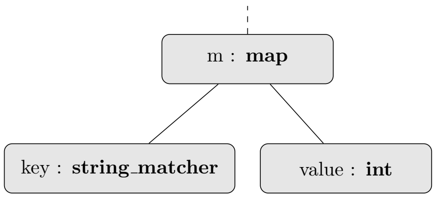
(b) JSONPath tree G_P for $.(*id* | *engine.horsepower*).

Fig. 8: Tree structures for the path evaluation.

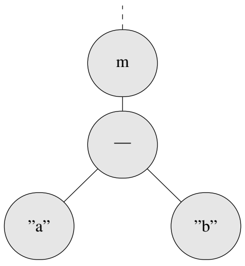
Extract attributes We first consider only the projection of fields and present the specifics of selection further below. The extraction of the attributes relevant to a given query works by recursively traversing every path in the JSON path tree and at the same time determining the respective path in the parser tree. The algorithm works by starting from the roots of both trees, $v_o \in V_O$ and $v_p \in V_P$. Then, for each child $v'_p \in V_P$ of the current path tree node v_p, the corresponding child $v'_o \in V_O$ of the current object parser tree node v_o is determined. The same procedure is repeated for the obtained pair (v'_p, v'_o) of child nodes. Once, the path tree node v'_p is a leaf in the path tree, the respective object parser tree node v'_o represents one attribute that is required by the query. Therefore, the signals *valid_out* and *data_out* of the parser block that was previously generated for this object parser tree node v'_o are then connected with the first stage register.

Example: Take as an example the parser tree G_O shown in Fig. 8a and the path tree G_P of the path expression `$.`(*id* | *engine.horsepower*) shown in Fig. 8b. For extracting all attributes needed for selection and projection, we first consider the root nodes of both trees, that is, the `record` parser block of the parser tree and the $ node of the path tree. Starting from the $ node, we next consider the children in the path tree. In Fig. 8b, this is the | node, which in turn indicates that the path splits into two subpaths, each considering the respective child node in the path tree and its associated parser block. Thus, in the first subpath, the *id* : `int` parser block and the *id* node in the path tree are selected. Given that the *id* node is a leaf node, the currently selected parser block (*id* : `int`) is marked for extraction so that a later selection or projection can be performed. The same procedure is repeated in the right subpath. However, in this case, the observed node in the path tree (*engine*) is not a leaf node, which is why its child (*horsepower*) must be again taken into consideration. As the currently selected parser block is of `record` type, the corresponding parser block to the *horsepower* node can be selected (*horsepower* : `float`). At this point, the path node is a leaf node as well, which means that the *horsepower* block is also marked for extraction. After all paths have been traversed, the signals *out_data* of all parser blocks marked for extraction are connected to the stage 1 registers, which are written with a one on the signal *out_valid* of the selected block.

Attribute extraction works straightforward for elementary and record parser blocks. However, `map` and `union` types have peculiarities as discussed in the following. While with a record the children in the path tree correspond directly to the children in the parser block, the path children of the map type correspond to its queried key strings (cf. Fig. 9). Accordingly, no parser blocks are selected, but the `string_matcher` match dictionary is set up with the searched key string. The resulting signal *key_match* is then used as a write condition for writing *out_data* to the projection register in stage 1. Since several entries can be extracted from a map, there is also the possibility to split the path into subpaths using | nodes, as can be seen in Fig. 9b. In this case, a separate entry is created in the `string_matcher` dictionary for each of the visited children (here *"a"* and *"b"*), each with its independent signal *key_match*. The signal *out_data* of the value parser block is then connected to a separate stage 1 register for each child and only written if the respective *key_match* is present.



(a) Object parser tree G_O for the map type.



(b) JSONPath tree G_P for path *.m.("a"|"b")*.

Fig. 9: Tree structures for the path evaluation of the `map` type.

Whereas with the `maps` type, the same parser block can be projected multiple times, the `union` type contains multiple parallel parser blocks, one for each data type the field may potentially take. During the projection and selection phase, it is therefore necessary to decide which of the types and, with that, which of the parser blocks is relevant for the query and should therefore be connected with the stage 1 register. The selection of the target type could be done at runtime or statically at design time. Since we want to enforce a fixed data layout for further data processing on the FPGA, we have opted for static types. However, since the JSONPath syntax does not provide for type casts or similar, we extended the syntax accordingly. For this purpose, the expected type is specified after the attribute name separated by two colons (e.g., for a `union` variable *sensor* which is to be mapped to the `int` type, the syntax is *$.sensor*`::union(int)`). Assuming it is not statically known which union type is to be expected, all possible types must be projected as separate fields, which of course creates overheads, but still allows to preserve a fixed layout.

Apply selection logic Filter expressions start with a question mark and iterate over all array entries which can be referenced via the @ symbol. To be able to perform selections on the record scope, we have again extended the JSONPath notation lightly. Thereby, entire records can be filtered by specifying the filter expression directly at the root of the record (e.g., $[`?(id=0)`]). If the expression evaluates to false, the entire record is discarded in the parser and not passed on for further processing. The resulting path tree G_P is depicted in Fig. 10.

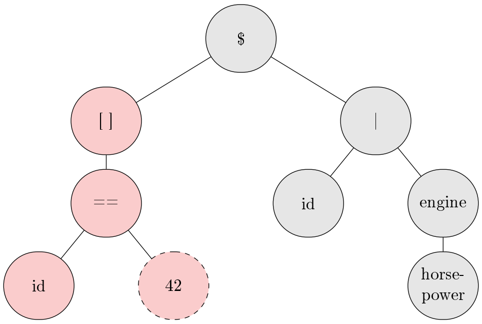


Fig. 10: Path tree G_P with selection for JSONPath $[`?(id=42)`].(*id* | *engine.horsepower*).

The filter expression is always placed at the left child of the root node, so before traversing the path, it is checked first whether there is a filter expression at this position. If this is the case, the filter expression is evaluated first. When visiting the comparison node, the corresponding compare logic is instantiated based on the data types of its children. Here, we support direct comparisons of two strings, two booleans, or two enums and <, >, ≤, ≥, == comparisons of two integers or two floats. When referencing attributes, as is the case with the `id` block in Fig. 10, the corresponding parser block is connected to a stage 1 register as described in the previous paragraph. The register is then used as input for the comparison logic, as shown in Fig. 1 and the result is connected to the valid flag of the stage 2 register. The right subtree represents the projection paths and therefore is evaluated as described before. The stage 1 registers of the attributes to be projected must then be connected to the stage 2 registers.

2.3 Phase 3 – Accelerator generation

In the third phase, additional logic is generated to obtain an accelerator module that can be deployed on an FPGA. For communication with other accelerators or system interfaces, the generated PPS module is connected to AXI interfaces. Furthermore, if the Avro objects are embedded in a wrapper format, such as the container format or the wire format, its decoding must be carried out. Furthermore, since the PPS module generated in the previous phase can only process 1 byte per clock cycle, it is necessary to instantiate multiple PPS modules to run parallel in order to achieve a high throughput.

Usually, we allocate 8 parallel PPS modules, each being fed by one channel. The wire format introduced in Sect. 1.1 contains multiple subsequent buffers. Each buffer stores one or multiple Avro objects. The buffers are assigned to the channels in a round-robin fashion. As we instantiate 8 parallel channels, this scheme can saturate a 64-bit interface at a throughput of 1 byte/cycle and channel.

However, this approach also results in limitations concerning the input data. If the objects are required to be reassembled after processing in the same order they entered, each buffer may only contain one Avro. Splitting individual Avro objects to the channels and assembling from the parallel PPS modules has then to happen in the same round-robin fashion. For larger objects, this is no problem, since the overhead of the buffer length field on the overall data footprint can be neglected. Should this be a problem nevertheless, it could be solved by modifying the Wire format: By splitting the 4 bytes of the length field into 3 bytes for the buffer length and 1 byte for the number of records contained, the output stream could later be reassembled based on the given number of records in each buffer and channel.

2.4 Automatic hardware generation

Automatic generation of hardware is typically done by emitting VHDL or Verilog code, using template engines, or even worse, using a large number of print statements in the generator. This results in extremely poor readability and maintainability of the generator code. We therefore decided to generate all logic circuits using Python-based HDL called Amaranth[5]. In Amaranth, hardware is described at the register transfer level as in VHDL or Verilog, while allowing for modern language features of Python such as object orientation. This is especially advantageous for the implementation of the parser blocks, since, for example, all complex parser blocks can inherit from a class `sequential_parser`, which already contains basic FSM logic for sequential activation of the instantiated parser blocks. In addition, the interface for parser blocks introduced in Sect. 2.1 can be inherited by all blocks via an abstract class, so instantiating parser blocks in other blocks (e.g., in the `record` parser) can be easily implemented using attributes of the abstract class type. Moreover, the object structure created in this way is perfectly suited for traversing the parser tree

[5] Amaranth HDL: https://github.com/amaranth-lang/amaranth

(see Sect. 2.2). The described hardware can then be finally output in Verilog, allowing the generated modules to be used platform-independently. The Python-based HDL not only allows us to create highly nested modules, but it is also possible to leverage well-established Python modules for parsing the Avro schema and JSONPath expressions.

3 Evaluation

We selected two stream processing benchmarks to evaluate our approach and the generated circuits. These are the Yahoo Streaming Benchmark [Ch16], which monitors advertising campaigns, and the RIoTBench [SCS17], which includes various applications for the Internet of Things. In both cases, we only consider the parsing stage, as well as subsequent projections and selections for our evaluation. As both benchmarks originally expect JSON-formatted data as input, an equivalent Avro schema had to be defined first. However, to define these schemas, we first discuss the input data of both benchmarks. Following this, a path expression is to be chosen based on the selection & projection applied in each benchmark.

Yahoo Streaming Benchmark The input JSON data of the Yahoo Streaming benchmark consists of 7 string attributes. These are first three UUIDs (*user_id*, *page_id* & *ad_id*) to identify the advertisement event, two type fields (*ad_type* & *event_type*), a timestamp (*event_time*) and the IP address of the user (*ip_address*). We decided to encode the UUIDs using the fixed type (16 Byte), the type fields (*ad_type* & *event_type*) using one enum in each case, the timestamp using a long and the IP address using a string. During selection, it is tested whether the *event_type* enum type is set to the *"view"* enum element[6] Subsequently, the attributes *ad_id* and *event_time* are projected in the benchmark, which results in the following path expression: $[`?event_type = 'view'`].(*ad_id* | *event_time*).

RIoTBench Similarly to the Yahoo Benchmark, the RIoTBench originally works with JSON formatted data. The structure of the JSON data is based on the SenML format, which is used as an exchange format for sensor measurements. The JSON records received in the benchmark are encoded as a JSON record with initially two fields. This is first a *timestamp* of the measurements and second an *array* which contains all measured values. The array is in turn always comprised of 8 records. Each measurement records contain three fields, a *value* field with the actual measured value, a field for the *name* of the measured value, and a field for the *physical unit*. While the name and unit fields are encoded as a string, the value field can be encoded either as an integer or as a float depending on the physical unit of the sensor measurement.

We adapted the benchmark for Avro. The Avro schema first contains a `long` timestamp and a field of type `map` (*values*), which contains the sensor measurements. We chose to use a map

[6] In the original benchmark, this is a string comparison as the *event_type* attribute is formatted as a string.

for the measurements instead of an array, as this makes it easier to extract its entries. A `map` entry, respectively a sensor measurement, consists of the sensor name as `key` and `union` type for its value. The `union` type, in turn, defines one type for each physical unit. Accordingly, the correct value type (`int` or `float`) can be defined for each physical unit. In the SmartCity query of the RIoTbench, five variables are initially projected after parsing. All five variables are then used for a selection expression. The resulting path expression can be seen in List. 3.

```
$[?values.temperature::union(senml_fahrenheit) >= -12.5
    & values.temperature::union(senml_fahrenheit) <= 43.1
    & values.humidity::union(senml_percentage) >= 10.7
    & values.humidity::union(senml_percentage) <= 95.2
    & values.light::union(senml_percentage) >= 1345
    & values.light::union(senml_percentage) <= 26282
    & values.dust::union(senml_percentage) >= 186.61
    & values.dust::union(senml_percentage) <= 5188.21
    & values.airquality_raw::union(senml_percentage) >= 17
    & values.airquality_raw::union(senml_percentage) <= 363]
        .(values.temperature::union(senml_fahrenheit))
        |(values.humidity::union(senml_percentage))
        |(values.light::union(senml_percentage))
        |(values.dust::union(senml_percentage))
        |(values.airquality_raw::union(senml_percentage))
```

List. 3: Path expression for the RIoTBench SmartCity query.

System architecture The generated parser accelerators have been evaluated on a Xilinx ZCU106 Zynq SoC. Fig. 11 depicts the architecture of our system, based on [Be19], consisting of a tightly coupled ARM CPU and programmable logic (PL). The PL contains several dynamically Reconfigurable Regions (RRs), which are connected to each other and to various interfaces via a crossbar. In each of the RRs resides a DMA engine, managed by the on-chip ARM CPU.

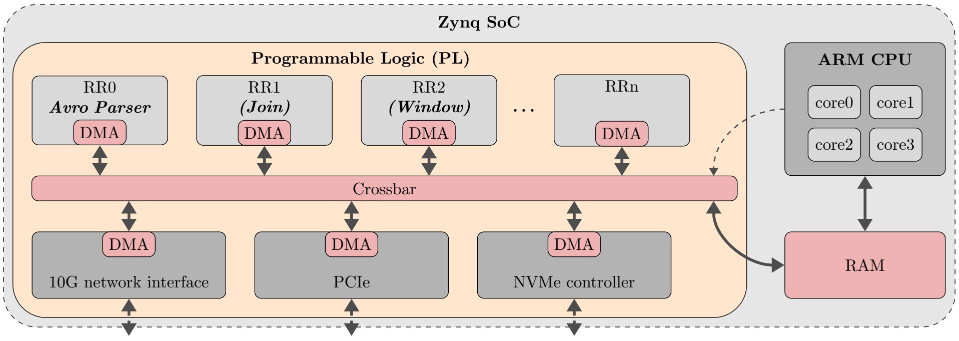


Fig. 11: FPGA-based system architecture for evaluation.

In our experiments, 3.6 MB of Avro data from the Yahoo benchmark, as well as 1.5 MB from the RiotBench, were preloaded into RAM and transferred to the parser accelerator using DMA. The results containing the parsed Avro objects were again written back to RAM via DMA. For the experiments, the entire system was clocked at 200 MHz, thus with a word width of 64 bits, a throughput of 1.6 GB/s should ideally be achievable. In practice, however, we only achieved a throughput of 1.45 GB/s, which was due to the CPU not being able to schedule new DMA descriptors fast enough. In the future, this problem could be solved by using more cores to schedule the DMAs, or even by adding a dedicated hardware component for scheduling the DMAs. For us, however, the achieved throughput is sufficient since it suffices to process the incoming data from a 10 GBit/s network interface at line rate. Our system architecture can thus be used to process and accelerate further stages or even an entire data stream processing application on the FPGA.

Example: Consider the Yahoo Benchmark again. First, a data stream of Avro objects is received at the network interface. This stream is then forwarded as described to the first Reconfigurable Region (RR) which contains our generated Avro parser accelerator, directly performing the first three steps of the Yahoo benchmark (parse, project & select). Then the stream of parsed, projected, and filtered Avro objects is passed to the next RR, which performs a join against a document store in the Yahoo benchmark. The tuples of the document store required for joining are also transferred via DMA from the NVMe controller to the corresponding RR. The joined tuples are then passed again to another RR to aggregate via a window function in the last step. The accelerators for processing joins and windows can be generated as shown in past research [MTA09; MTA10; TM11]. The output stream of the window is again to be stored in the document store and must be accordingly transferred back to the NVMe controller.

Benchmark results Besides the above experiments, we determined the maximum achievable clock frequency as well as the resource consumption of the generated accelerator engine for both benchmarks. The results are depicted in Tab. 2. The number of LUTs required for the more complex RIoTBench schema is slightly higher than for the Yahoo benchmark, but remains low overall for both accelerators, thus allowing resources to be used for further query processing, as described above. For the same reason, the maximum achieved clock frequency is also higher for the Yahoo benchmark. If the two generated accelerators are operated at their maximum clock rate, they can theoretically achieve throughputs of 3.4 GB/s (Yahoo) and 2.8 GB/s (RIoTBench). Unlike JSON and CBOR, in Avro it is likely that decoded objects have a larger data footprint at the output than at the input, making the output interface potentially the bottleneck. However, since we perform additional selections and projections in both evaluated benchmarks, the amount of data at the output is typically greatly reduced compared to the input, so the input interface remains practically the bottleneck, meaning that the given throughput numbers still correspond to the parsing speed.

Finally, the two parsing benchmarks were run with the C++ Apache Avro parser on an Intel(R) Core(TM) i7-3770 CPU to obtain an x86 CPU baseline. With one thread, a throughput of 390 MB/s was achieved for the Yahoo benchmark and a throughput of 96 MB/s for the more complex schema of the RIoTBench. A speedup of up to 4 can be achieved by parallelizing on multiple threads for exploiting the 8 hyperthreads provided by the CPU. Here, the limited scaling results from additional overhead due to the memory management for distribution of the data as well as the synchronization of the threads. Only the parsing itself was measured in the CPU benchmarks. If selection and projections are additionally performed and the results of the threads are merged, the throughput would even be worse. In contrast, the proposed FPGA design could ideally achieve a speedup of 8.8 for the Yahoo benchmark or 29.4 for the more complex RIoTBench given that the problems of the DMA scheduling are resolved, while only requiring a minimal share of its resources.

Tab. 2: Resource consumption, clock frequency, and throughput results for two benchmarks.

	benchmark	Yahoo	RIoTBench
resources (% of FPGA)	LUTs	4,900 (2.1%)	6,691 (2.9%)
	FFs	7,288 (1.6%)	7,173 (1.6%)
	maximal clock frequency	430 MHz	359 MHz
throughput (speedup)	CPU single thread	390 MB/s (1)	96 MB/s (1)
	CPU multi thread	1,405 MB/s (3.6)	380 MB/s (4.0)
	FPGA theoretical	3,440 MB/s (8.8)	2,827 MB/s (29.4)
	FPGA experimental	1,450 MB/s (3.7)	1,450 MB/s (15.1)

4 Conclusion & Future Work

Avro's simple encoding can be interpreted using basic finite-state machines, making the parsing process perfectly suited for acceleration in hardware using FPGAs. The accelerators generated by the presented generator can achieve significant speedups compared to CPU-based parsers, although only a minimal share of the FPGA resources is required. Moreover, path expressions can be used to parse the received objects into a fixed data layout and reduce the amount of output data to avoid unnecessary data movement. The enforced data layout is then perfectly tailored to accelerate further steps of the given application on the available FPGA resources.

Optimization potential for our approach arises from the fact that parser blocks must be instantiated multiple times when the same Avro type occurs multiple times in the schema. Furthermore, the generated parser is only able to parse the schema it was generated with, making schema evolution only possible by creating and instantiating multiple parser accelerators. In the future, we want to solve these two problems by coordinating the parser blocks via an instruction set. The schema would then be translated into a program that would control the sequence of parser blocks. Thus, different schemas and, accordingly, schema evolution could be achieved simply by executing different programs.

References

[Ap13] Apache Software Foundation: Apache ORC Specification v1, Oct. 13, 2013, URL: `https://orc.apache.org/specification/`.

[Ap21] Apache Software Foundation: Apache Avro 1.11.0 Specification, Oct. 29, 2021, URL: `https://avro.apache.org/docs/1.11.0/spec.pdf`.

[Ap22a] Apache Software Foundation: Apache Arrow Columnar Format Specification v1, Dec. 1, 2022, URL: `https://arrow.apache.org/docs/format/Columnar.html`, visited on: 12/01/2022.

[Ap22b] Apache Software Foundation: Apache Parquet Specification, Mar. 24, 2022, URL: `https://parquet.apache.org/docs/`.

[Be19] Becher, A.; Herrmann, A.; Wildermann, S.; Teich, J.: ReProVide: Towards Utilizing Heterogeneous Partially Reconfigurable Architectures for Near-Memory Data Processing. In. Gesellschaft für Informatik, Bonn, 2019, URL: `https://doi.org/10.18420/btw2019-ws-04`.

[BH20] Bormann, C.; Hoffman, P.: Concise Binary Object Representation (CBOR), STD 94, RFC Editor, Dec. 2020, URL: `https://tools.ietf.org/pdf/rfc8949`.

[Br17] Bray, T.: The JavaScript Object Notation (JSON) Data Interchange Format, RFC 8259, Dec. 2017, URL: `https://www.rfc-editor.org/info/rfc8259`.

[Ch16] Chintapalli, S.; Dagit, D.; Evans, B.; Farivar, R.; Graves, T.; Holderbaugh, M.; Liu, Z.; Nusbaum, K.; Patil, K.; Peng, B. J.; Poulosky, P.: Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming. In: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). Pp. 1789–1792, 2016, URL: `https://doi.org/10.1109/IPDPSW.2016.138`.

[Da22] Dann, J.; Wagner, R.; Ritter, D.; Faerber, C.; Froening, H.: PipeJSON: Parsing JSON at Line Speed on FPGAs. In. DaMoN'22, 2022, URL: `https://doi.org/10.1145/3533737.3535094`.

[El10] El-Hassan, F.; Ionescu, D.: A hardware architecture of an XML/XPath broker for content-based publish/subscribe systems. In: 2010 International Conference on Reconfigurable Computing and FPGAs. IEEE, pp. 138–143, 2010.

[Gö07] Gössner, S.: JSONPath - XPath for JSON, Feb. 20, 2007, URL: `https://goessner.net/articles/JsonPath/`.

[Go22] Google Inc.: Protocol Buffers Version 3 Language Specification, Nov. 1, 2022, URL: `https://developers.google.com/protocol-buffers/`.

[Ha22] Hahn, T.; Becher, A.; Wildermann, S.; Teich, J.: Raw Filtering of JSON data on FPGAs. In: DATE'22, Antwerpen. Mar. 14–23, 2022, URL: `https://doi.org/10.23919/DATE54114.2022.9774696`.

[HWT22] Hahn, T.; Wildermann, S.; Teich, J.: Auto-Tuning of Raw Filters for FPGAs. In: FPL'22, Belfast, United Kingdom. Aug. 29–Sept. 2, 2022.

[KSS08] Koch, C.; Scherzinger, S.; Schmidt, M.: XML Prefiltering as a String Matching Problem. In: ICDE'08. Pp. 626–635, 2008, URL: `https://doi.org/10.1109/ICDE.2008.4497471`.

[Li17] Li, Y.; Katsipoulakis, N. R.; Chandramouli, B.; Goldstein, J.; Kossmann, D.: Mison: A Fast JSON Parser for Data Analytics./, 2017, URL: `https://doi.org/10.14778/3115404.3115416`.

[Me10] Melnik, S.; Gubarev, A.; Long, J. J.; Romer, G.; Shivakumar, S.; Tolton, M.; Vassilakis, T.: Dremel: Interactive Analysis of Web-Scale Datasets. In: VLDB'10. 2010, URL: `http://www.vldb2010.org/accept.htm`.

[Mi09] Mitra, A.; Vieira, M.; Bakalov, P.; Najjar, W.; Tsotras, V.: Boosting XML Filtering with a Scalable FPGA-based Architecture./, 2009, URL: `https://arxiv.org/abs/0909.1781`.

[MTA09] Mueller, R.; Teubner, J.; Alonso, G.: Streams on Wires: A Query Compiler for FPGAs./, Aug. 2009, URL: `https://doi.org/10.14778/1687627.1687654`.

[MTA10] Mueller, R.; Teubner, J.; Alonso, G.: Glacier: a query-to-hardware compiler. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. Pp. 1159–1162, 2010, URL: `https://doi.org/10.1145/1807167.1807307`.

[Pe21] Peltenburg, J.; Hadnagy, Á.; Brobbel, M.; Morrow, R.; Al-Ars, Z.: Tens of gigabytes per second JSON-to-Arrow conversion with FPGA accelerators. In: ICFPT'21. 2021, URL: `https://doi.org/10.1109/ICFPT52863.2021.9609833`.

[Sa12] Sadoghi, M.; Javed, R.; Tarafdar, N.; Singh, H.; Palaniappan, R.; Jacobsen, H.-A.: Multi-query Stream Processing on FPGAs. In: 2012 IEEE 28th International Conference on Data Engineering. Pp. 1229–1232, 2012, URL: `https://doi.org/10.1109/ICDE.2012.39`.

[SCS17] Shukla, A.; Chaturvedi, S.; Simmhan, Y.: RIoTBench: An IoT benchmark for distributed stream processing systems. Concurrency and Computation: Practice and Experience/, Oct. 2017, URL: `https://doi.org/10.1002%2Fcpe.4257`.

[TM11] Teubner, J.; Mueller, R.: How Soccer Players Would Do Stream Joins. In. SIGMOD '11, Athens, Greece, 2011, URL: `https://doi.org/10.1145/1989323.1989389`.

[TWN12] Teubner, J.; Woods, L.; Nie, C.: Skeleton Automata for FPGAs: Reconfiguring without Reconstructing. In. SIGMOD '12, 2012, URL: `https://doi.org/10.1145/2213836.2213863`.

[WA11] Woods, L.; Alonso, G.: Fast data analytics with FPGAs. In: 2011 IEEE 27th International Conference on Data Engineering Workshops. IEEE, pp. 296–299, 2011, URL: `https://doi.org/10.1109/ICDEW.2011.5767669`.