Automated Composition and Execution of Hardware-accelerated Operator Graphs

Stefan Werner, Dennis Heinrich, Jannik Piper, Sven Groppe
Universität zu Lübeck
Institute of Information Systems
23562 Lübeck, Germany
Email: {lastname}@ifis.uni-luebeck.de

Rico Backasch
Technische Universität Dresden
Faculty of Computer Science
01187 Dresden, Germany
Email: rico.backasch@tu-dresden.de

Christopher Blochwitz, Thilo Pionteck
Universität zu Lübeck
Institute of Computer Engineering
23562 Lübeck, Germany
Email: {lastname}@iti.uni-luebeck.de

***Abstract*—In this paper, we present the fully automated composition and execution of Semantic Web queries within a hardware/software system which uses a Field Programmable Gate Array (FPGA) as an accelerator. The presented approach allows to write a query in the database's front-end, transparently executes all steps to retrieve a configuration suitable for the FPGA which represents the given query, and obtains the result by evaluating the query on the FPGA. In order to obtain a runtime-reconfigurable framework we define a static and a dynamic partition. The dynamic partition itself is composed by using a predefined query template and a pool of operators. The operators of the given query are written automatically into the template and connected accordingly. After reconfiguration of the FPGA the host system supplies the initial data to the FPGA which computes the final result and sends it back to the host system to be displayed to the user or application. The evaluation shows that not all queries may take benefit from a dedicated hardware-accelerator but shows promising speedup for complex queries.**

I. Introduction

It is well-known that the amount of data in our information society underlies an inherent, steady growth. On one side that progress is technically motivated, e.g., due to massive usage of any kind of sensors and technological improvements to gather more precise measurements which in turn both directly result in heavy data increase. On the other side, there are several economically (social networks, advertisement) and politically (intelligence agencies) motivated reasons. Whereas persistently storing of these massive data is hassle-free, the processing and analysis of this huge amount of data within an acceptable time frame becomes more and more difficult. In order to cope with these problems, in the last decades intensive work was done to optimize database software and data structures. Furthermore, technological advances enabled shrinking feature size to increase clock frequency and thus the overall performance. However, nowadays these approaches are getting close to their limits (*power wall*) and in the last years the trend evolved to multi/many-core systems in order to increase performance. Additionally, these systems are not assembled with uniform cores, but rather are composed by heterogeneous and specialized cores which compute a specific task efficiently. The main issue of this approach is that these specialized cores cannot be used in application domains showing a huge variety in processing. Widely available Field Programmable Gate Arrays (FPGA) with the capability of (partial) runtime-reconfiguration

978-1-4673-7942-7/15/$31.00 ©2015 IEEE

[1] are able to close the gap between the flexibility of general-purpose CPUs and the performance of specialized hardware-accelerators. With respect to the information flood we develop a hybrid hardware/software system which allows the user to write a query to retrieve specific information from Semantic Web datasets by automatically adapting the query structure to a runtime-reconfigurable FPGA and execute the query evaluation on it.
The remainder of this work is structured as followed: Section II outlines related work in the area of hardware-accelerators in database systems and depicts our new contributions. Section III gives an architectural overview by introducing the system's general work flow between host and FPGA and introduces the used data structures. The automated composition of the configuration suitable for the FPGA is described in Section IV. Section V evaluates the feasibility of the presented approach and Section VI concludes with a summary and future work.

II. Related Work

Mueller et al. [2], [3] propose the component library and compositional compiler *Glacier*. It takes a continuous query for data streams and generates a corresponding VHDL file. After the time-consuming synthesis, mapping and place & route the query can be programmed on an FPGA in order to accelerate the evaluation on data streams. Consequently, this approach is only suitable for a known query set. Additionally, the library does not cover join operators.
Teubner et al. [4] present a window-based stream join, called Handshake join. The approach allows items of two data streams flowing by each other in opposite directions to find join partners with each item they encounter. All items in a predefined window are considered in parallel to compute an intermediate result. Due to the window-based architecture and since the window size is limited by the chip size it cannot be guaranteed that the result contains all possible join partner of the two datasets.
IBM's Netazza FAST engines [5] uses FPGAs to reduce the amount of data to be transferred from (persistent) memory to the CPU by early execution of projection and restriction. Furthermore, uncompressing data at wire speed increases the read throughput and thus reduces the drawback of hard disks. As the system does not support a modular composition of complete queries, Dennl et al. [6], [7] present concepts for on-the-fly hardware acceleration of SQL queries in the re-

© 2015 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

lational database MySQL. The authors focus on restriction and aggregation operators and thus cannot execute complete queries on the FPGA. However, in order to evaluate more complex queries (including joins) additional views are created to represent partial results and the proposed hardware/software system achieves promising speed-up gains.
Becher et al. [8] extend this approach to an embedded low-energy system-on-chip platform and added more complex operators (e.g., Merge Join and sorting of small datasets). For a simple query including one join they achieve a comparable performance but higher energy efficiency than a standard x86-based system. Additionally, they provide a theoretical model for their operators to estimate the performance.
Casper et al. [9] explore approaches to accelerate in-memory database operations with focus on throughput and utilization of memory bandwidth during sorting. The presented system performs an equi-join after sorting of two tables.

In the context of Semantic Web databases we develop a hybrid hardware/software system which transparently transforms and executes SPARQL queries on a run-time reconfigurable FPGA. In previous works we have presented our approaches to implement and execute the join operator [10], [11] and filter expressions [12] on an FPGA. Whereas these operators have been evaluated in isolation, this work presents the fully automated and transparent composition and execution of complete operator graphs on an FPGA. As a result the user is able to simply write an arbitrary SPARQL query in the GUI of LUPOSDATE [13] which is automatically transformed into a reconfiguration suitable for the FPGA and evaluated on it. Finally, the result is displayed by the host system to the user. To the best of our knowledge this is the first system which fully-automatically executes queries with an arbitrary number of operators[1] on Semantic Web data on an FPGA.

III. Architectural Overview

In the following we introduce some fundamentals of Semantic Web databases. Furthermore, we present the overall architecture of the hybrid hardware/software system and give an insight into how the synchronization between host and FPGA is done during query evaluation.

A. Data Representation and Queries

The Resource Description Framework (RDF) [14] is used as the basic data format in the Semantic Web to describe statements about web resources. The data is represented as RDF triples (S,P,O). The components are called subject (S), predicate (P) and object (O). A (possibly distributed) set of RDF triples is used as the data basis for SPARQL [15] queries. Listing 1 shows an example for a SPARQL query to retrieve all document titles and their year of publication in the SP²B dataset [16]. The SELECT clause defines a projection list of variables to appear in the final result (i.e., the bindings of the variables *?title* and *?yr*). The WHERE clause contains two triple patterns. Matching triples lead to bindings of the variables. As the variable *?doc* appears in both triple patterns both intermediate results will be joined (in this case with a Merge Join). The resulting operator graph is shown in Figure 1.

[1] only restricted by FPGA resources

Listing 1: SPARQL query on the SP²B dataset

```
1 PREFIX dc: <http://purl.org/dc/elements/1.1/>
2 PREFIX dcterms: <http://purl.org/dc/terms/>
3
4 SELECT ?title ?yr
5 WHERE {
6   ?doc dc:title ?title .
7   ?doc dcterms:issued ?yr
8 }
```

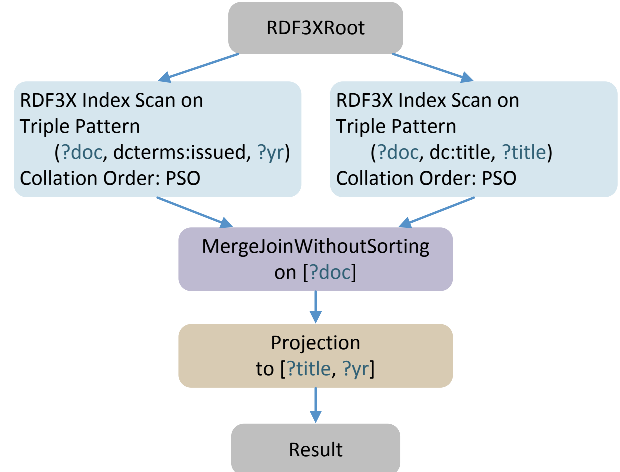
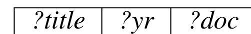


Fig. 1: Operator graph of the example query

Depending on the constant values in a triple pattern LUPOSDATE chooses a collation order for each index scan operator. As in both triple patterns the predicate is a constant value the collation order PSO is chosen. This means the data is primarily sorted by the predicate, secondarily by subject and tertiary by object. In fact, LUPOSDATE uses six indices, each for one of the six collation orders (SPO, SOP, etc.), and thus is able to retrieve sorted triples efficiently for a given triple pattern. Intermediate results (bound values for the variables) are stored in *bindings arrays*. Each variable has a dedicated position in this array where its bound value is stored (see Figure 2). Furthermore, the LUPODATE system uses a dictionary to map RDF terms into integer IDs [17] and thus each binding in the bindings array is an integer ID which refers to the actual string representation. Due to lower space consumption of the evaluation indices and a significantly smaller memory footprint of intermediate results during query execution, this feature is intensively used by LUPOSDATE. With respect to the FPGA design this is quite handy as handling arrays of integer values is rather easier than dealing with strings of unfixed length.

B. Hybrid Work Flow

Figure 3 shows the general program flow. After the user submitted a query the LUPOSDATE system transforms the



?title	*?yr*	*?doc*

Fig. 2: Bindings array of the example query

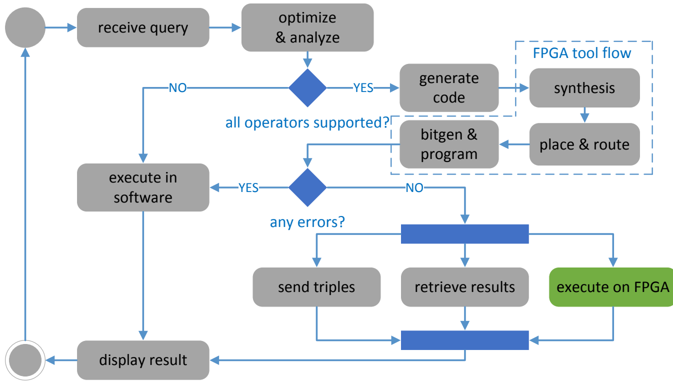


Fig. 3: Flow chart of the hybrid system

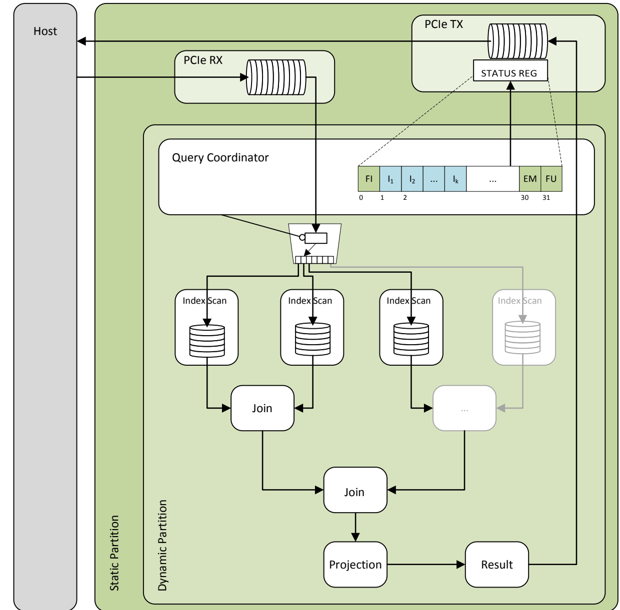


Fig. 4: FPGA design

query into an operator graph. On the operator graph several logical and physical optimizations are applied [17]. The optimized operator graph is now analyzed for unsupported operators[2]. In case such an operator was found the query is evaluated completely by the software engine on the host system. Besides implementing more operators on the FPGA, in the future we plan to break down this strict separation as well. This means that the FPGA processes as much as possible of the operator graph and the host system covers the remaining operators. However, if all operators are supported then our new extension traverses the operator graph and generates a VHDL file which represents the given query. A detailed description of this process can be found in Section IV. On the VHDL description of the circuit the full FPGA tool chain is applied (synthesis, mapping, place & route). The resulting bitstream is programmed on the FPGA and the query is ready to be executed. Again, in case of an error (e.g., translation failed caused by lack of FPGA resources) the query is evaluated in software. This always preserves a running system. If no error occurred the query is executed on the FPGA. In fact, during execution the host and FPGA work in parallel. The host's task simply covers providing the input triples and retrieval of results from the FPGA.

C. Architecture

The general architecture of the design on the FPGA is shown in Figure 4. It consists of a static and a dynamic partition. The static partition contains the modules which are needed by each query independent of the actual query structure. Mainly those modules are the receive (RX) and transmit (TX) engines of the PCIe interface. The Query Coordinator, located in the dynamic partition, is directly connected to these engines. PCIe RX is used to transfer RDF [14] triples as well as commands to indicate which triples belong to which data source. Incoming triples are simply inserted into a FIFO which can be accessed from the dynamic partition. The Query Coordinator controls a demultiplexer in order to lead incoming triples into the right index scan operator. Similarly, the TX engine contains a FIFO which is used for writing results from the dynamic partition. Furthermore, the Query Coordinator is able to write a status register to indicate the global status of the query execution to the host system (see Table I). The host

[2] e.g., sorting operators are not implemented yet

TABLE I: Status Register

Bit	Name	Description
0	FI	'1' when query processing finished, else '0'
1..k	$I_1 \dots I_k$	'1' when index scan I_i ($i \in 1 \dots k$) has space for more triples, else '0' (k currently set to 16)
(k+1) .. 29	unused	can be used to enable more index scans
30	EM	'1' when no results are available, else '0'
31	FU	'1' when result FIFO is FULL, else '0'

system reads the FIFO or status register. Depending on the flags of the status register the host system executes different tasks. If FI is set to '1' then the host system retrieves the remaining results from the FIFO until the EM flag is set to '0' and the result is shown to the user. At the beginning of the query execution the host system sends a specific amount of triples for all index scan operators. After this initial loading the host system determines the triples to be sent by controlling the request flags of all index scan operators $I_1, ..., I_k$. If one index scan is saturated its corresponding flag is set to '0'. The host system continuously checks which index scan flags are set to '1' and thus sends more triples for these index scans only. Hence, overflow and data loss are avoided. If a saturated index scan depletes then its corresponding flag is set back to '1' and the host will send triples in the next round.

IV. Automated Composition

The configuration of the dynamic partition is generated automatically for each query. We developed a VHDL template and a pool of operators to compose the VHDL file which represents the query. In turn the template consists of a static and dynamic part as well (shown in Table II). In the following a detailed description for each part of the template is given.

TABLE II: Parts of the VHDL template for the dynamic partition

Static signals & constants definition
Dynamic signals & constants definition
Dynamic signal assignment
Dynamic mapping of index scans
Static instantiations of entities
Dynamic instantiations of entities
Static *glue* logic

Listing 2: Connection record

```
1 type op_connection is record
2   read_data  : std_logic;
3   data       : std_logic_vector(DW-1 downto 0);
4   valid      : std_logic;
5   finished   : std_logic;
6 end record op_connection;
```

A. Template - Static Parts

Each query contains a result operator which is connected using the signal *results* of the record type `op_connection` (see Listing 2). It is used as an interface between the dynamically generated operator graph and the static code which takes the query results and serializes them into the PCIe TX engine. The record type `op_connection` plays an important role in the dynamic part as well and will be explained in detail in Section IV-B. Furthermore, the constant *TRL* (Triple Receive Limit) is defined. It indicates how many triples each index scan can temporarily hold. If an index scan exceeds this number then it is considered as saturated and thus (temporarily) no more triples for this index scan are requested. The host is allowed to send up to *TRL* triples for one index scan k in one burst without checking its corresponding flag I_k in the status register. Typically this value is set to half of the index scans internal FIFO size. This is motivated by the fact that the host sends triples for one specific index in a burst of *TRL* triples. If I_k is set to '0' then the internal FIFO is more than half full and thus cannot store another burst of *TRL* triples without potential data loss. If I_k is set to '1' then the internal FIFO is less than half full and thus can store at least one burst of *TRL* triples. As the internal FIFOs of the index scans are empty at the beginning of query processing the host can send up to 2**TRL* triples to each index scan.
Additionally, the static instantiation of entities covers a reset generator, a cycle counter (for debugging purposes and to evaluate the raw FPGA performance) and the Query Coordinator (QC). The QC is the interface between the PCIe RX engine in the static partition and an adjustable[3] number of index scans. For incoming data the QC distinguishes two types: (i) a message to select an index scan or (ii) triple components. If the host has sent a message to select one index scan then this implies that the next up to *TRL* triples belong to the previously selected index scan and are inserted into its FIFO. If *TRL* triples have been received then the QC expects the next message to select the index scan. Additionally, if a predefined *invalid* triple component (0xFFFFFFFF) is received then this indicates that all triple of this index scan are sent by the host.

[3] we support up to 16 index scan operators

B. Template - Dynamic Parts

The template is analyzed for the dynamic parts (see Table II) by LUPOSDATE at startup. Basically the dynamic parts are marked with predefined markups as comments. LUPOSDATE searches for those markups and replaces them with VHDL code as follows.

1) Operator Instantiation and Interconnects: The operator template can be seen in Figure 5. It defines the input and output signals which need to be implemented by each operator. Operators can have up to two preceding operators. If an operator needs only one preceding operator (e.g., filter) then only the left input is used. The second input is simply not used by the operator and not connected to any other operator by the automated composition algorithm. The following synthesis implicitly detects those unused signals and removes them accordingly. Furthermore, each operator has exactly one succeeding operator. The signals are grouped in such a way that the output of each operator can be used as an input for any other operator. Each group consists of (i) a vector *data* which corresponds to the bindings array, (ii) a *valid* flag which indicates the validity of *data*, (iii) a *finished* flag which indicates the end of *data*, and (iv) a backward flag *read* which notifies the proceeding operator that *data* was read. Reading the *data* implicitly invalidates the *data* until new data arrives. It can be seen that one group corresponds to the previously defined record type `op_connection` (Listing 2). The data width (DW) of the signal *data* depends on the number of variables in the bindings array and is set for each query.

Thus, while traversing the optimized operator graph for each visited operator, an ID X is assigned and the signals *opXinput1*, *opXinput2* and *opXoutput1* are defined (see lines 1 to 3 in Listing 3). Additionally an entity of a specific operator with type *OperatorType* is instantiated (line 5). In the port map (lines 11 to 27 in Listing 3) the internal operator signals are connected with the previously defined `op_connection` signals. The generic map (lines 6 to 10 in Listing 3) is used to parameterize this particular operator. All operators have the generics for the data and value width in common which correspond to the width of the bindings array and variables. Furthermore, each operator can have additional individual generics. These extensions will be described in Section IV-C. The actual wiring of two consecutive operators is outlined in Listing 4. As the output of operator X is used as (left) input of the operator Y this implies that X is the predecessor of Y (see lines 2 to 4). The *read_data* flag indicates operator X that

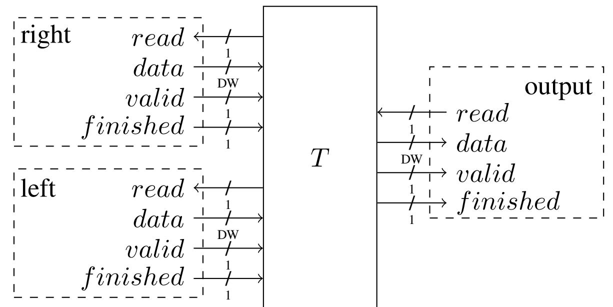


Fig. 5: Common operator interface

Listing 3: Operator instatiation

```
signal opXinput1   : op_connection;
signal opXinput2   : op_connection;
signal opXoutput1  : op_connection;
[...]
operatorX : entity work.OperatorType(arch)
generic map(
  DATA_WIDTH  => BINDINGS_ARRAY_WIDTH,
  VALUE_WIDTH => BINDINGS_WIDTH,
  --[... more operator specific generics ...]
)
port map(
  [...]
  left_read      => opXinput1.read_data,
  left_data      => opXinput1.data,
  left_valid     => opXinput1.valid,
  left_finished  => opXinput1.finished,

  right_read     => opXinput2.read_data,
  right_data     => opXinput2.data,
  right_valid    => opXinput2.valid,
  right_finished => opXinput2.finished,

  result_read    => opXoutput1.read_data,
  result_data    => opXoutput1.data,
  result_valid   => opXoutput1.valid,
  finished_out   => opXoutput1.finished
);
```

Listing 4: Linking of two operators X and Y

```
opXoutput1.read_data <= opYinput1.read_data;
opYinput1.data       <= opXoutput1.data;
opYinput1.valid      <= opXoutput1.valid;
opYinput1.finished   <= opXoutput1.finished;
```

operator Y has read the provided data and thus can provide the next data (see line 1).

2) Input Mapping: As described before (Section IV-A) the QC receives index scan select commands and triple data. With the knowledge of the ID of the index scan, the QC simply triggers a demultiplexer to switch between the index scans. This is shown for two index scan operators in Listing 5, lines 1 to 18. The signals *index_scan_left_data* and *index_scan_left_valid* are outputs of the QC and redirect a triple component respectively indicate the validity of this triple component to the index scan. Lines 19 and 20 of Listing 5 show how the synchronization between index scans and QC is done in order to avoid an overflow and thus data loss. The QC has an input vector *index_scan_left_read* with up to k bits, one for each index scan. Each *read_data* flag of the index scans is assigned at its corresponding position in *index_scan_left_read*. The QC can directly reuse this information to set the flags I_1 to I_k in the previously described status register (Table I) which in turn is evaluated by the host system to determine which index scans are depleted and thus need more triples. Depending on the actual query and the number of index scans this code is dynamically generated.

C. Parametrization of Operators

As mentioned before each operator type can have individual generics. In the following the scheme to parametrize operators is outlined.

Listing 5: Mapping of incoming triples to index scan operators

```
triple_store_map : process(index_scan_select_id,
    index_scan_left_data, index_scan_left_valid)
begin
  op1input1.data  <= (others => '0');
  op1input1.valid <= '0';
  op2input1.data  <= (others => '0');
  op2input1.valid <= '0';
  case index_scan_select_id is
    when 1 =>
      op1input1.data(index_scan_left_data'range)
        <= index_scan_left_data;
      op1input1.valid <= index_scan_left_valid;
    when 2 =>
      op2input1.data(index_scan_left_data'range)
        <= index_scan_left_data;
      op2input1.valid <= index_scan_left_valid;
    when others => null;
  end case;
end process triple_store_map;
index_scan_left_read(1)  <= op1input1.read_data;
index_scan_left_read(2)  <= op2input1.read_data;
```

1) RDF3XIndexScan: The RDF3XIndexScan is the link between the QC and the inner operators. Typically the RDF3XIndexScan provides data triples s, p, o but a bindings array can have less or more than three variables and also not all of the three triple components might be necessary to evaluate the query. Thus, the main objective of this operator is to receive triples from the QC and map their required components to a position in the bindings array. Therefore the RDF3XIndexScan has three additional generic one-hot-coded bit vectors (`SUBJECT_POSITION`, `PREDICATE_POSITION`, `OBJECT_POSITION`). Each vector consists of as much bits as there are variables in the bindings array. During synthesis these vectors are evaluated following a simple scheme: If bit x is set in the bit vector of one triple component then this triple component is connected to position x in the bindings array. Unbound variables are initialized with an invalid value (0xFFFFFFFF).

2) Join: This operator joins the intermediate results of two preceding operators depending on one or more common join attributes. Similar to the RDF3XIndexScan, the position of the join attribute is determined by the one-hot-coded bit vector `JOIN_VECTOR`. As the structure of the bindings array is globally the same in the whole operator graph, only one set bit is necessary. However, it is possible that a join on more than one common variable is executed. Although this is not yet supported by the proposed system it is possible to simply add additional bit vectors for secondary, tertiary, etc. orders. In fact, there are several different algorithms for join computation such as Merge or Hash Join [10] but all are equipped with the same generics.

3) Filter: In previous work [12] we presented two approaches to implement the filter operator for Semantic Web databases. Taking the optimizer of LUPOSDATE into account we are able to break down complex filter expressions into multiple simple filter operators of the scheme *VALUE COND VALUE*, with *COND* as the condition (e.g., equality) and *VALUE* either a constant or variable. Specifically, this means that conjunctions of filter expressions result in a chain of simple filter operators each checking only one relational condition.

In case of disjunction the operator duplication takes place and thus multiple disjunctive conditions are evaluated by simple filter operators in concurrent branches of the operator graph. In turn the intermediate results of both (or more) branches simply need to be unified in a lower level of the operator graph. As a result each filter operator is equipped with the following generics. The generic `FILTER_OP_TYPE` describes the relational operation to be evaluated by the filter. Due to the mapping from strings to integer IDs our approach supports only *equal* and *unequal* comparators at the moment. However, if the dictionary (ID→string) would be available on the FPGA also other conditions such as *greater/smaller than* are possible. Furthermore, we have to distinguish between expressions comparing a variable with a constant value and comparing two variables of the bindings array. Therefore `FILTER_LEFT_IS_CONST` is set if the left value is a constant. If so then the constant value is passed through the generic `FILTER_LEFT_CONST_VALUE` by setting the actual value to be compared. Contrary if the left value is not a constant then the one-hot-coded bit vector `FILTER_LEFT_VAR_POS` is considered. Like in previously described operators a set bit in this vector corresponds to the position of the variable in the bindings array. By simply replacing the term `LEFT` with `RIGHT` in the generics that scheme is applied for the right value of the filter expression as well.

4) Projection: The Projection carries out the SELECT clause of the SPARQL query. Therefore it is equipped with the bit vector `PROJECTION_VECTOR`. It has as much bits as the bindings array has variables. If bit x is set to '1' in the bit vector then the corresponding variable at position x in the bindings array remains in the result. Otherwise the corresponding variable is projected out.

5) Other Operators: Several other operators such as Limit, Offset, (Merge-)Union, etc. have been implemented. In general the parametrization follows the same pattern and is omitted due to space restrictions.

V. Evaluation

A. Test Setup

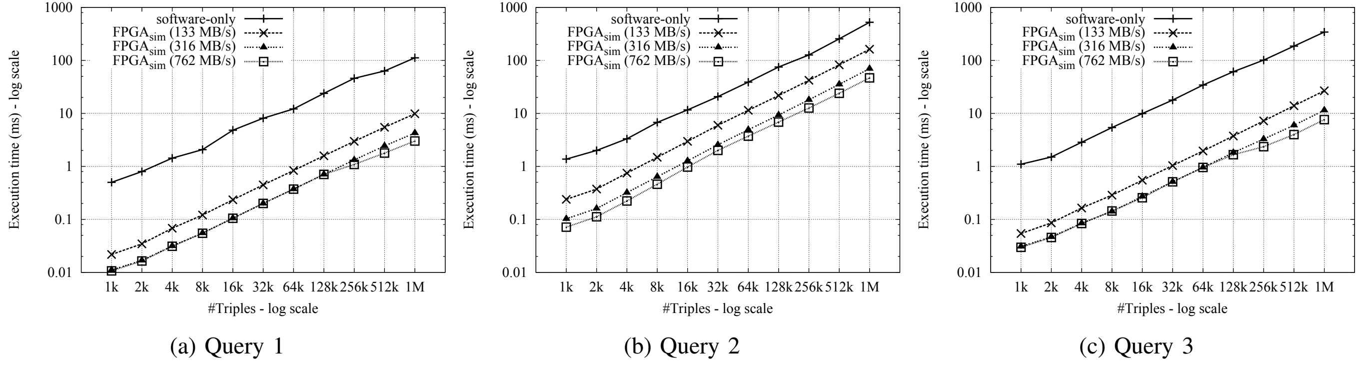
The test dataset is generated with the pre-compiled SP^{2}B data generator [18]. For evaluation purpose we executed different queries on different datasets with sizes from 1,000 up to 1 million triples. Due to missing operators (e.g., sorting and distinct) we choose three SPARQL queries (inspired by the SP^{2}B queries) to show the feasibility of our approach. Query 1 consists of one join and a simple filter expression. Query 2 consists of two joins which can be executed independently. Both intermediate results are unified. Query 3 consists of three joins. Two of them can be executed independently as well, while the third join combines the intermediate results of the previous two operators. The last join operates in a pipelined fashion concurrently to the other two joins. The complete test queries can be found in the appendix.

The host system is a Dell Precision T3610 (Intel® Xeon® E5-1607 v2 3.0 GHz, 40 GB DDR3-RAM) which is equipped with a Xilinx Virtex®-6 FPGA (XC6VHX380T) [19] board via PCIe 2.0 with 8 lanes. Although the presented approach is completely implemented and integrated into LUPOSDATE at this point we cannot provide real-world execution times of the whole system due to instabilities and performance lacks in our PCIe implementation. However, the following estimated throughput requirements are promising that the system with fully equipped PCIe interface is able to achieve performance enhancements. The query circuit operates at a clock frequency of 200 MHz. As described before the *Query Coordinator* consumes incoming data immediately and distributes the triple components to the corresponding index scan operators. Together with the 32-bit data width of one triple component respectively one variable this results in a throughput of 762 MByte/s. With respect to protocol overhead of PCIe it still achieves a much higher throughput than required in our design.

B. Execution Times and Speedup

In order to get an idea of possible performance enhancements we execute the operator graphs of the three given queries in a simulation environment. The evaluated operator graph is automatically generated by the described system and the used input data is directly retrieved from the LUPOSDATE system as it would send it in a real-world case. Thus, the PCIe is emulated with the previously described throughput of 762 MByte/s for host-to-FPGA and FPGA-to-host communication. However, in order to address a possibly limited bandwidth between host and FPGA we evaluate the performance with a bounded throughput of 316 MByte/s and 133 MByte/s as well. All FIFOs are empty at the beginning of each run. The start timestamp is taken at simulation start and corresponds to the first writing of data into the host-to-FPGA FIFO on the host's side. The end timestamp is taken when the finished flag FU is set and the FPGA-to-host FIFO is empty (indicated by rising EM) and thus all results are consumed.

Figure 6 shows the execution times of the test queries for different dataset sizes. Each data point of the software system represents the average of 1,000 single executions with warm caches. In both, in the software and FPGA implementation, all queries scale linearly to the dataset size. Obviously, decreasing the throughput between host and FPGA results in a significant increase of execution time especially for bigger datasets. Interestingly, with a relatively low bandwidth of 133 MByte/s the FPGA-based query execution still outperforms the software approach. However, taking the synchronization between host and FPGA into account this low throughput might not be enough and has to be evaluated in the future. In addition, the achieved performance gain by the FPGA differs significantly depending on the query. As previously described we expect that not all queries can be successfully accelerated. Figure 7 depicts this expectation by means of the achieved speedup. The speedup values are obtained taking the execution time of the software solution divided by the execution time of the FPGA solution (with host-to-FPGA bandwidth of 762 MByte/s). Whereas queries 1 and 3 achieve speedup factors between 33 and 49, the speedup of query 2 starts at 19 and settles down at a factor of 10. In fact the speedup degradation is caused by the union operator. Although this operator is very simple it tends to consume one intermediate result of one preceding operator and stalls the other preceding operator. Although 10 is still a significant speedup this gain might be lost in the real-world case.



(a) Query 1 (b) Query 2 (c) Query 3

Fig. 6: Execution times of the test queries (in milliseconds - logarithmic scale)

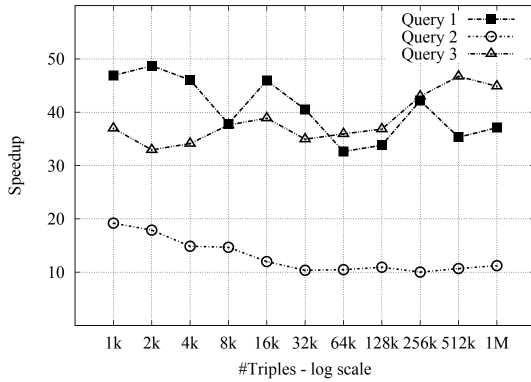


Fig. 7: Speedup (Exec. time software / Exec. time FPGA)

C. Open Issues

Besides the lacking performance of our PCIe implementation another main issue needs to be addressed: generating the partial bitfile takes between 20 and 30 minutes in our system and thus is far from being used in a real-world case. A straightforward approach is the generation of bitfiles for highly frequent queries and reuse them during runtime. In fact, the detection and reuse of known queries is already prototypically implemented in the proposed system. However, in a highly dynamic environment this approach is not applicable as any arbitrary query is possible. In the following we discuss two approaches to overcome this problem in the future.

1) Semi-static Operator Graph: In a semi-static operator graph the overall structure of one or more query trees can be implemented and programmed on the FPGA. Each node of this tree is implemented as a *super operator* which means that it contains the implementation of all possible operators including a bypass operator. In such a system the optimizer simply needs to generate a stream of meta data to set the final configuration and thus the behavior of the operator graph. The meta data stream is broadcasted through the operator graph and determines the type for each operator including operator specific meta data such as position of the join variable or filter condition. As each node is set to one operator type at a time a big amount of resources (used for other operator types) is utilized but actually not used. Consequently, less chip area is available, e.g., for other or larger operator graphs.

2) Multiple Dynamic Partitions: As previously described we use partial reconfiguration to exchange the actual operator graph while static modules such as the PCIe interface are not replaced with each query. One step further, we can divide the space into multiple dynamic partitions (tiles). Each tile is able to take one (but arbitrary) operator. Thus each tile should provide the same resources. An efficient way for identification of homogenous reconfiguration areas is presented by Backasch et al. [20]. Before system runtime the partial bitfile for each operator and each possible tile needs to be generated. In order to adjust operator specific properties (such as position of join attribute) it is necessary to store these information in registers within the operator and modify their content by manipulating the bitfile according to the query. Besides this major challenge, a semi-static network between the tiles needs to be established. This can be done by a predefined routing structure to neighboring tiles and adding an additional signal to the operator's interface to choose a neighbor as its succeeding operator in the operator graph.

VI. Summary

In this paper, we presented a fully fledged and integrated query framework which enables the user to write a query in LUPOSDATE's front-end which is transparently evaluated on an FPGA. Therefore we use partial runtime-reconfiguration to exchange the query to be executed. The dynamic partition is automatically assembled by using a query template and a pool of operators. With respect to the query the contained operators are connected with each other and parametrized by operator specific generics. As all operators implement a common interface the presented framework can be easily extended by new operator implementations. In the evaluation we discussed the impact of query structure and bandwidth between host and FPGA on the execution times and speedup. Furthermore, we outlined two approaches to cope with the reconfiguration overhead which will be extensively investigated in future work. At the moment, the host systems holds all the initial data to be queried. As our FPGA is equipped with a SATA interface it might be reasonable to store triple data at hard drives attached on the FPGA. Consequently, the communication overhead between host and FPGA will be significantly reduced because only the result of a query needs to be sent to the host

system. Additionally, we intend to take advantage of FPGAs in other computationally intensive database tasks such as index generation as well.

ACKNOWLEDGEMENTS

This work is funded by the German Research Foundation (DFG) project GR 3435/9-1 and is supported by the Federal Ministry for Economic Affairs and Energy on the basis of decision by the German Bundestag (KF3125601).

Supported by:

on the basis of a decision
by the German Bundestag

REFERENCES

[1] Kyprianos Papadimitriou, Apostolos Dollas, and Scott Hauck. Performance of partial reconfiguration in fpga systems: A survey and a cost model. *ACM Trans. Reconfigurable Technol. Syst.*, 4(4):36:1–36:24, December 2011.

[2] Rene Mueller, Jens Teubner, and Gustavo Alonso. Streams on Wires: A Query Compiler for FPGAs. *Proc. VLDB Endow.*, 2:229–240, August 2009.

[3] Rene Mueller, Jens Teubner, and Gustavo Alonso. Glacier: A Query-to-Hardware Compiler. In *Proceedings of the 2010 International Conference on Management of Data*, SIGMOD '10, pages 1159–1162, New York, NY, USA, 2010. ACM.

[4] Jens Teubner and Rene Mueller. How Soccer Players Would do Stream Joins. In *Proceedings of the 2011 International Conference on Management of Data*, SIGMOD '11, pages 625–636, New York, NY, USA, 2011. ACM.

[5] IBM Corp. The Netezza Data Appliance Architecture: A Platform for High Performance Data Warehousing and Analytics, 2011.

[6] Christopher Dennl, Daniel Ziener, and Jürgen Teich. On-the-fly Composition of FPGA-Based SQL Query Accelerators Using a Partially Reconfigurable Module Library. *20th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 20:45–52, 2012.

[7] Christopher Dennl, Daniel Ziener, and Jurgen Teich. Acceleration of SQL Restrictions and Aggregations Through FPGA-Based Dynamic Partial Reconfiguration. In *Proceedings of the 2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM '13, pages 25–28, Washington, DC, USA, 2013. IEEE Computer Society.

[8] Andreas Becher, Florian Bauer, Daniel Ziener, and Jürgen Teich. Energy-Aware SQL Query Acceleration through FPGA-Based Dynamic Partial Reconfiguration. In *Proceedings of the 24th International Conference on Field Programmable Logic and Applications (FPL 2014)*, pages 662 – 669. IEEE, 2014.

[9] Jared Casper and Kunle Olukotun. Hardware acceleration of database operations. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, FPGA '14, pages 151–160, New York, NY, USA, 2014. ACM.

[10] Stefan Werner, Sven Groppe, Volker Linnemann, and Thilo Pionteck. Hardware-accelerated Join Processing in Large Semantic Web Databases with FPGAs. In *Proceedings of the 2013 International Conference on High Performance Computing & Simulation (HPCS 2013)*, pages 131 – 138, Helsinki, Finland, July 1 - 5 2013. IEEE.

[11] Stefan Werner, Dennis Heinrich, Marc Stelzner, Volker Linnemann, Thilo Pionteck, and Sven Groppe. Accelerated join evaluation in Semantic Web databases by using FPGAs. *Concurrency and Computation: Practice and Experience*, May 18 2015.

[12] Stefan Werner, Dennis Heinrich, Marc Stelzner, Sven Groppe, Rico Backasch, and Thilo Pionteck. Parallel and Pipelined Filter Operator for Hardware-Accelerated Operator Graphs in Semantic Web Databases. In *Proceedings of the 14th IEEE International Conference on Computer and Information Technology (CIT2014)*, pages 539–546, Xi'an, China, September 11 - 13 2014. IEEE.

[13] S. Groppe. LUPOSDATE Open Source. [Online] https://github.com/luposdate, 2013.

[14] World Wide Web Consortium (W3C). RDF/XML Syntax Specification (Revised), 2004. W3C Recommendation.

[15] World Wide Web Consortium (W3C). SPARQL Query Language for RDF, 2008. W3C Recommendation.

[16] Michael Schmidt, Thomas Hornung, Georg Lausen, and Christoph Pinkel. SP2Bench: A SPARQL Performance Benchmark. In *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*, pages 222–233, 2009.

[17] Sven Groppe. *Data Management and Query Processing in Semantic Web Databases*. Springer Verlag, Heidelberg, 2011.

[18] Michael Schmidt, Thomas Hornung, Georg Lausen, and Christoph Pinkel. SP2Bench. [Online] http://dbis.informatik.uni-freiburg.de/index.php?project=SP2B/download.php, 2009.

[19] Xilinx. Virtex-6 Family Overview. [Online] http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf, January 2012.

[20] Rico Backasch, Gerald Hempel, Sven Groppe, Stefan Werner, and Thilo Pionteck. Identifying Homogenous Reconfigurable Regions in Heterogeneous FPGAs for Module Relocation. In *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Cancun, Mexico, December 8 - 10 2014.

APPENDIX

Query 1

```
1  #Get all articles with property swrc:pages.
2  PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-
       syntax-ns#>
3  PREFIX bench: <http://localhost/vocabulary/bench/>
4  PREFIX swrc:  <http://swrc.ontoware.org/ontology#>
5
6  SELECT ?article
7  WHERE { ?article rdf:type bench:Article .
8          ?article ?property ?value .
9          FILTER (?property=swrc:pages)}
```

Query 2

```
1  #Get incoming and outcoming properties of persons.
2  PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-
       syntax-ns#>
3  PREFIX foaf: <http://xmlns.com/foaf/0.1/>
4
5  SELECT ?predicate
6  WHERE {
7    { ?person rdf:type foaf:Person .
8      ?subject ?predicate ?person
9    } UNION {
10     ?person rdf:type foaf:Person .
11     ?person ?predicate ?object
12   }
13 }
```

Query 3

```
1  #Get all articles with title, number of pages and
       creator.
2  PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-
       syntax-ns#>
3  PREFIX swrc: <http://swrc.ontoware.org/ontology#>
4  PREFIX bench: <http://localhost/vocabulary/bench/>
5  PREFIX dc: <http://purl.org/dc/elements/1.1/>
6
7  SELECT ?article ?title ?pages ?creator
8  WHERE { ?article rdf:type bench:Article .
9          ?article dc:title ?title .
10         ?article swrc:pages ?pages .
11         ?article dc:creator ?creator }
```