# Parallel and Pipelined Filter Operator for Hardware-Accelerated Operator Graphs in Semantic Web Databases

Stefan Werner, Dennis Heinrich, Marc Stelzner, 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

Thilo Pionteck
Technische Universität Hamburg-Harburg
Institute of Computer Technology
21071 Hamburg, Germany
Email: thilo.pionteck@tuhh.de

***Abstract*—In this paper, we investigate the use of FPGAs to enhance the performance of filter expressions in Semantic Web databases. The filter operator is a central part of query evaluation. Its main objective is to reduce the amount of data as early as possible in order to reduce the calculation costs for succeeding and more complex operators such as join operators. Due to the proximity to the data source it is essential for the overall query performance that the filter operator is able to evaluate single data items as fast as possible. In this work, the advantages of using FPGAs in query evaluation are outlined and an overview about the provided degree of parallelism is given. We propose two different approaches to implement the filter operator for the Semantic Web database LUPOSDATE are presented. The Fully-Parallel Filter evaluates all conditions by dividing the input into several sub-items which are evaluated by dedicated sub-filters in parallel. The second approach creates a pipeline of sub-filters to evaluate the filter expression *step-by-step*. If an item reaches the end of this pipeline then it complies the whole filter expression. The final evaluation shows that both approaches of the hardware-implemented filter operator running at 200 MHz defeat the comparable software solution written in C running at 2.66 GHz. Processing 100M items per second, the hardware-accelerated filter provides a more than 5 times higher throughput than the general-purpose CPU. In contrast to the software solution, the total throughput is independent of the match rate and the structure of the filter expression, and is a valuable contribution to the hardware-accelerated query evaluation.**

***Keywords*—Database Systems, Query Processing, Sematic Web, Field Programmable Gate Arrays***

## I. Motivation

Databases form the backbone of the today's World Wide Web. While the data sets increase rapidly the time requirements to query these data become more strict. Several approaches over the last decades optimized the database software in order to reduce the query execution time but are also limited by the hardware architecture. On the other side, shrinking feature size to increase the clock frequency finally hit the power wall. As a consequence, the costs to make processors faster grow exponentially, while costs to increase the number of processing units increase linearly. Thus, nowadays multi-/many-core systems are omnipresent. Besides this trend, specialized hardware-accelerators have been developed to accomplish one specific task and usually can not be applied in application domains showing huge variety in processing.

In [1] we proposed the conception of our hybrid software-/hardware system to provide hardware-accelerated query evaluation in Semantic Web databases. As one crucial element of the query execution we evaluated several join algorithms on an FPGA. Due to the complexity and its memory consumption it is likely to execute joins later and reduce the amount of data in earlier processing steps. Typically, the query optimizer tries to push filter operations as close as possible to the data source in order to reduce the intermediate results and thus succeeding operators have to cope with less amount of data. Usually, those are more complex operations like joins which have higher memory requirements and need more additional storage to fulfill their task. In the query execution tree the succeeding operators can not start processing until the first intermediate results arrived. As the filter is executed in an early stage of query execution, it is crucial that the filter reaches a low latency and high throughput.

In Section II we outline related work about hardware-accelerated database operations. Section III describes the fundamentals about FPGAs and introduces the Semantic Web database to be accelerated and the used data structure. The implementation of the filter operator on the FPGA is described in Section IV and evaluated in Section V. The summary of the presented work and an outlook for future work are given in Section VI.

## II. Related Work

FPGAs are used to enhance computation efficiency in several scientific areas. However, as this work focuses on the filter operator in query evaluation, in the following we summarize existing work using hardware acceleration in the context of databases. First attempts were made by Leilich et al. [2] respectively DeWitt [3] in the late 1970's but were limited by the technological capabilities and high manufacturing costs. With the increasing availability of FPGAs the idea of hardware-accelerated databases got revived and resulted in innovations the last years. The Netezza Data Appliance Architecture [4] consists of several distributed computing nodes (*S-Blades*). Each node is equipped with multi-core CPUs, gigabytes of random access memory and FPGAs which are mainly used for compression and filtering of data coming from physical hard drives in order to accelerate relational databases. The FPGA is used as a co-processor to support the CPU and neither considers a modular composition of database operators nor the complete query execution in the FPGA. Mueller et al. ([5], [6]) present the *Glacier* component library and compiler in order to build a streaming engine for continuous queries. The compiler generates synthesizable VHDL code for a given query. Afterwards, a digital circuit is synthesized from the given code. This process is time consuming (minutes to hours) and thus only applicable for a known query set. Dennl et al. ([7], [8]) apply Netezza's concept of using FPGAs as a precomputational step between data source and data sink in order to accelerate SQL queries in relational database systems. They show that query execution benefits especially when arithmetic operations are involved. Our final intended hybrid hardware/software system follows a building block concept in order to compose pre-synthesized operators to a complex operator graph answering the given query. Hence, each operator needs to implement a common interface to enable the flexible composition. In the following, we describe two approaches to implement the filter operator in the context of query evaluation in Semantic Web databases.

## III. Architectural Overview

First we will give a basic introduction to FPGAs as it is the base for the final intended system. Additionally, in order to understand the design decisions we made to implement the hardware-accelerated filter operator, it is necessary to understand how the underlying database system works. Thus, in this section we will give a short overview to the Semantic Web database LUPOSDATE [9] and explain how the data is represented and evaluated.

### A. Field Programmable Gate Array

An FPGA (Field Programmable Gate Array) is an integrated circuit which can be configured to execute almost any logical function. In case of SRAM-based FPGAs the configuration is done by writing memory cells. The content of these memory cells determines the function of each particular basic element and their interconnection. The configuration data is *synthesized* from code written in a hardware description language like VHDL. In case of the Xilinx Virtex-6 FPGA [10] the chip area is divided into a grid of configurable logic blocks (CLB), each containing 2 slices, which in turn consist of 4 look-up tables (LUT) and 8 flip-flops. The LUTs have 6 inputs and 1 output and thus can implement any arbitrarily boolean function with 6 inputs and 1 output as well[1]. Combining those LUTs lead to more complex functions. Furthermore, dynamic partial reconfiguration [11], [12] enables to modify parts of the configuration while system runtime and to change the functionality of the whole design. Even though the clock rate is relatively low FPGAs can outperform solutions on general-purpose CPU. Due to their architecture and distributed memory (Block RAM) FPGAs achieve their performance from inherent parallelism enabled through pipelining and a high quantity of concurrently working units. In summary, reconfigurable hardware like FPGAs provide design flexibility like software but can be tailor-made oriented towards the actual problem to solve. Thus, it is an emerging technology to accelerate scientific applications in several areas like signal processing, cryptography and network based systems.

[1]Or as two 5-input LUTs with separate outputs but common logic inputs.

### B. Data Representation and Filter Expressions in LUPOSDATE

List. 1 shows a typical SPARQL query [13] which returns articles having more than 5 pages and published in February, March, May or April. A SPARQL query contains a SELECT and a WHERE clause. The SELECT clause defines a projection list of the variables to appear in the final query result (i.e., the bindings of the variables ?article, ?pages and ?month). The WHERE clause contains two triple patterns and a filter expression. The former constrain the input RDF [14] data. The filter expression consists of boolean formulas which are checked for each intermediate result during query execution. If the boolean formula becomes true the bindings of the variables remain in the intermediate result, otherwise they are discarded. Corresponding to the described query, Fig. 1 shows the optimized query execution plan constructed by the Semantic Web database LUPOSDATE [9]. As mentioned before the logical optimization tries to push filter expressions (purple rectangles) as close as possible to the data source (orange rectangles). The early evaluation of the filter expression reduces the size of intermediate results and consequently decreases the calculation costs for succeeding operators (in this case a Merge Join, green rectangle).

```
PREFIX swrc: <http://swrc.ontoware.org/ontology#>
SELECT ?article ?pages ?month
WHERE {
  ?article swrc:month ?month.
  ?article swrc:pages ?pages.
  FILTER(?pages>5 && ?month>1 && ?month<6)
}
```

Listing 1: Example SPARQL query with filter expression.

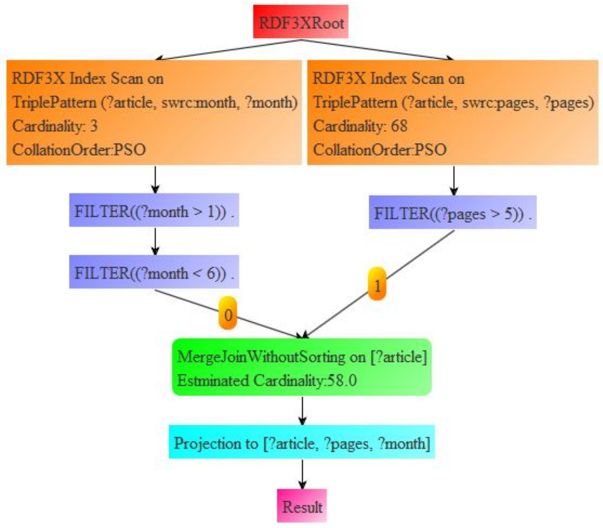

RDF3XRoot
RDF3X Index Scan on
TriplePattern (?article, swrc:month, ?month)
Cardinality: 3
CollationOrder:PSO
RDF3X Index Scan on
TriplePattern (?article, swrc:pages, ?pages)
Cardinality: 68
CollationOrder:PSO
FILTER((?month > 1)) .
FILTER((?pages > 5)) .
FILTER((?month < 6)) .
0
1
MergeJoinWithoutSorting on [?article]
Estminated Cardinality:58.0
Projection to [?article, ?pages, ?month]
Result

Fig. 1: Optimized operator graph to evaluate the query presented in List. 1 (generated by LUPOSDATE [9]).

The LUPOSDATE systems maps the occurrence of variables to a data structure called *bindings array*. In this array each variable has a dedicated position where the actual bounded value of this variable in an intermediate query result is stored. Furthermore, like RDF3X [15] and Hexastore [16] the SPARQL engine of LUPOSDATE uses dictionary indices to map RDF terms into integer ids [17]. This results in a lower space consumption of the evaluation indices and reduces the memory footprint of intermediate results. Contrary, the materialization (mapping back from ids to strings) is expensive especially in case of large final query results. However, typically the advantages surpass the disadvantages and with respect to the implementation of the filter operator in the FPGA this data structure can be easily used and mapped to signals and hardware primitives in the FPGA.

## C. Query Operators on FPGAs

As our approach follows a building block concept each operator works as a black box. Thus, an operator does not know which actual functions are processed by its preceding and succeeding operators. The communication between two operators bases on the operator template and the defined interfaces outlined in Fig. 2. It is motivated by *Volcano* [18], a well-known scheme for a query execution engine. The scheme enables a high flexibility in orchestrating operators and a pervasive degree of parallelism. Each operator has up to 2 preceding operators. Each is connected by 4 signals, namely *data*, *valid*, *finished* and *read*. The signal *data* provides the bindings arrays (with a generic data width *DW*) to be processed, whereas the *valid* flag indicates the validity of this data. If an operator has read the current data it uses the backward channel *read* to indicate the read to the preceding operator. The preceding operator might provide more data by raising the *valid* flag again. If no more data will be present this can be notified by raising the *finished* flag. Each operator implements those signals as an output as well and, consequently, its output can be used as an input of another operator. The final intended hybrid hardware-/software system will use dynamic partial reconfiguration to compose several pre-synthesized operators (each implements the described interface) to a complex query graph at system runtime.

## D. Dimensions of Parallelism in Query Execution

Due to the previously described operator template we can achieve a higher flexibility composing several operators to a query execution plan. Each single operator does not know any information about the kind of the preceding respectively succeeding operators. Furthermore, it enables multiple levels of parallelism and thus it provides perfect conditions to accelerate the query evaluation in FPGAs. Fig. 3 outlines the idea of mapping multiple queries into an FPGA to enable following levels of parallelism.

*1) Horizontal:* Horizontal parallelism is provided in different levels of granularity. First, multiple queries can be executed in parallel (*inter-query parallelism*). *Intra-query parallelism* is provided by the concurrent processing of intermediate results in two or more subtrees of a complex query tree. Regarding two operators on the same level of a (sub-)tree we refer as *horizontal inter-operator parallelism*. If the data can be partitioned and has less dependencies, then using several sub-units (built-in one operator) can enable *horizontal intra-operator parallelism*. The global operator verifies the partial results to get a global result.

*2) Vertical:* The signals *read* and *valid* control the data flow and ensure that the data items are transferred from one operator to another. As an operator provides the resulting bindings array as long as the succeeding operator notifies the read by raising the *read* signal, no data will be lost due to the overload of a lower operator. While one operator is processing its recently consumed bindings array, the preceding operator can already process newly arriving bindings arrays (*inter-operator parallelism*). The natural data flow from the *top* to the *bottom* of the whole operator tree establishes a processing pipeline (*vertical intra-query parallelism*). Depending on the granularity of an operator also a micro pipeline inside the operator can be established (*vertical intra-operator parallelism*).

## IV. Filter Operator on FPGA

In the following two approaches to implement the filter operator in an FPGA are presented. The previously described operator template supplies the input of two preceding operators to be processed by the current operator. The input bindings arrays are directly mapped to the *data* input signals as well as the resulting bindings array to the *data* output signals. However, the filter operator is a unary operator and consumes the actual data of only one preceding operator. Internally, it

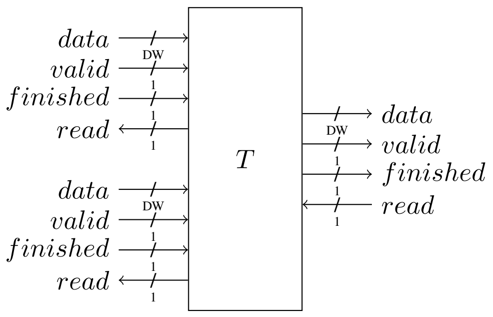


Fig. 2: Operator Template to combine several operators to a operator graph.

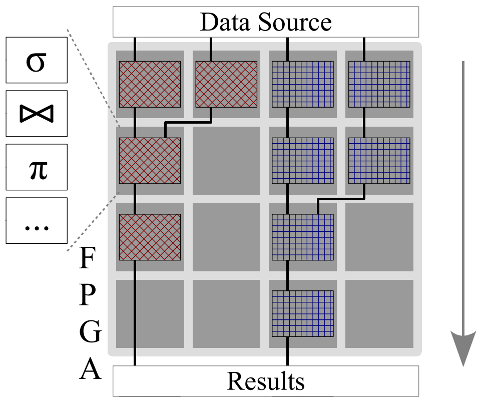
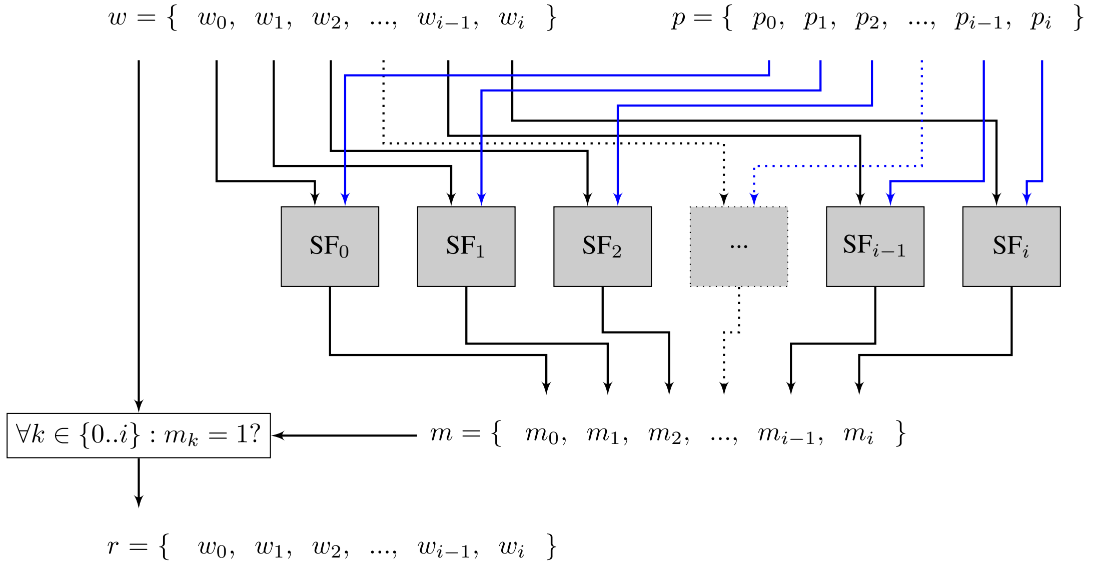


Fig. 3: Mapping of two operator graphs into the FPGA. Each tile contains one operator (Schematic).



$w = \{ w_0, w_1, w_2, ..., w_{i-1}, w_i \}$
$p = \{ p_0, p_1, p_2, ..., p_{i-1}, p_i \}$
$SF_0$
$SF_1$
$SF_2$
...
$SF_{i-1}$
$SF_i$
$\forall k \in \{0..i\} : m_k = 1?$
$m = \{ m_0, m_1, m_2, ..., m_{i-1}, m_i \}$
$r = \{ w_0, w_1, w_2, ..., w_{i-1}, w_i \}$

Fig. 4: Schematic of the Fully-Parallel Filter.

can just ignore the other input. This is feasible because there is actually no other preceding operator connected. In order to realize a higher flexibility we use the second input interface for configuration data like the pattern or the comparison operator.

## A. *Fully-Parallel Filter*

The Fully-Parallel Filter (Fig. 4) divides the bindings array $w$ and the pattern $p$ into its bindings $w_i$ and $p_i$ (horizontal intra-operator parallelism). The number $i$ of bindings depends on the defined total width of the bindings array and the value width of a binding. Each pair of $w_i$ and $p_i$ is connected to a sub-filter SF$_i$. Each SF$_i$ executes the actual comparison and decides if the binding $w_i$ fulfills the matching condition according to its corresponding pattern $p_i$. As this decision is binary, each sub-filter SF$_i$ indicates a match by *'1'*, respectively a *'0'* in case of a mismatch. Finally, the matches of all sub-filters are mapped to a match vector $m$ with $m_i$ is the result of SF$_i$. In order to verify a global match the global filter operator compares the match vector with the match mask *"1..1"*. Obviously, choosing the match vector in this manner allows only conjugated conditions and is chosen for simplicity in this case. We will describe later why this approach does not affect the generality and allows any kind of condition with the same throughput. With increasing the total width of the bindings array the number of sub-filters increases as well and the incoming bindings need to be distributed to their according sub-filters. This might cause that the distance in the FPGA between the port which receives the whole input bindings array and some of the sub-filters becomes very long. This may cause long signal paths which lead to longer clock cycles and thus limits the achievable clock frequency of the whole design. An example execution of the Fully-Parallel Filter is shown in Fig. 5. The bindings array contains the bindings of 3 variables and thus 3 sub-filters are used. Each sub-filter consumes a specific variable and the corresponding value of the pattern (5a). The first incoming bindings array satisfies the whole condition (Fig. 5b). Thus, all sub-filters set their corresponding bit in the match mask to *'1'* and the bindings array is forwarded to the output. As the next bindings array does not match the condition in its first and third variable, the positions 1 and 3 in the match mask are set to *'0'* and consequently the bindings array is discarded (Fig. 5c). The last bindings array results in a global match as well and is forwarded to the output (Fig. 5d).

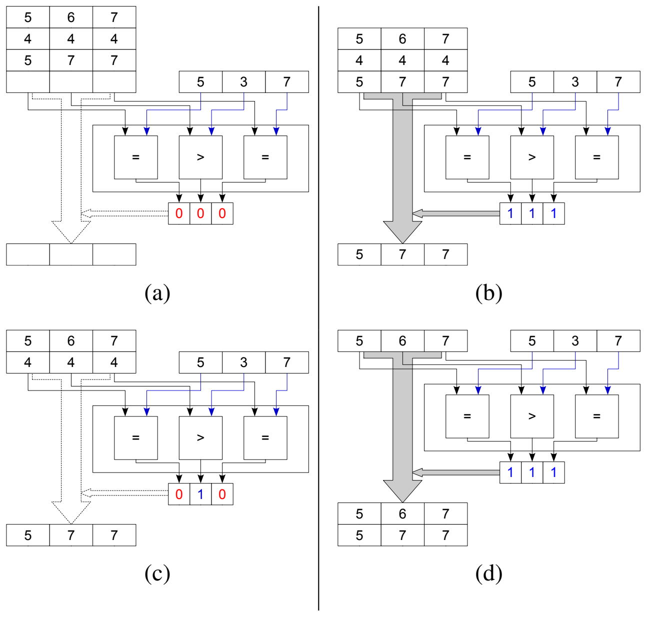


Fig. 5: Example of the Parallel Filter.

## B. *Pipelined Filter*

Contrary to the Fully-Parallel Filter, the Pipelined Filter (Fig. 6) does not divide the input bindings array $w$, respectively

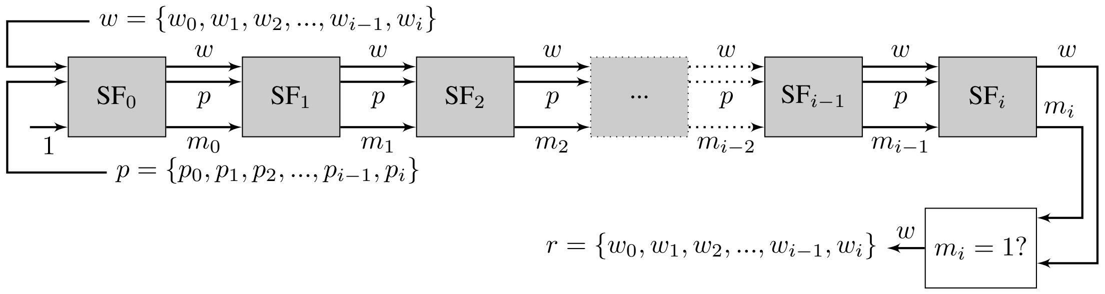

$w = \{w_0, w_1, w_2, ..., w_{i-1}, w_i\}$
SF$_0$
SF$_1$
SF$_2$
...
SF$_{i-1}$
SF$_i$
$w$
$p$
1
$m_0$
$m_1$
$m_2$
$m_{i-2}$
$m_{i-1}$
$m_i$
$p = \{p_0, p_1, p_2, ..., p_{i-1}, p_i\}$
$r = \{w_0, w_1, w_2, ..., w_{i-1}, w_i\}$
$m_i = 1?$

Fig. 6: Schematic of the Pipelined Filter.

the pattern $p$ into several bindings and distributes them into the sub-filters. It applies the whole bindings array as well as the pattern to the head of pipelined sub-filter units (*vertical intra-operator parallelism*). Each SF$_i$ is responsible for one specific binding in the data stream, and indicates a partial match by setting its output $m_i$. This output is used by the succeeding sub-filter and *iff* the partial match is set to $'1'$ then the succeeding sub-filter will consume $w$ and $p$ to evaluate its own condition. Otherwise the whole bindings array will be dropped within the pipeline and thus never reaches the end of the pipeline and will not be considered by succeeding operators. It is expected that the resource utilization is higher than for the Fully-Parallel Filter as the whole bindings array is forwarded from sub-filter to sub-filter which introduces additional registers to store the value. We can not shorten the bindings by the size of one binding with each pipeline step because the overall filter already consumes the next bindings array to forward it to the head of the sub-filter pipeline. Otherwise in case of a match the complete matching bindings array would not be available anymore. However, even with a higher resource utilization we expect a better scalability with increasing the data width. An example data flow of the Pipelined Filter is shown in Fig. 7. Each pipeline stage consists of one sub-filter and is responsible for one dedicated variable in the bindings array (Fig. 7a). The first incoming bindings array satisfies the first condition and thus is forwarded to the second pipeline stage. In the second stage it is checked for the next condition and meanwhile the next bindings array is applied in the first stage (Fig. 7b). As the second bindings array does not satisfy the first condition it is discarded and will not be evaluated in the subsequent stages. While the first bindings array is checked for the third condition, the first condition is checked for last bindings array (Fig. 7c). As the first as well as the third bindings array satisfy the whole filter expression they pass the whole pipeline and are forwarded to the output (Fig. 7d-e).

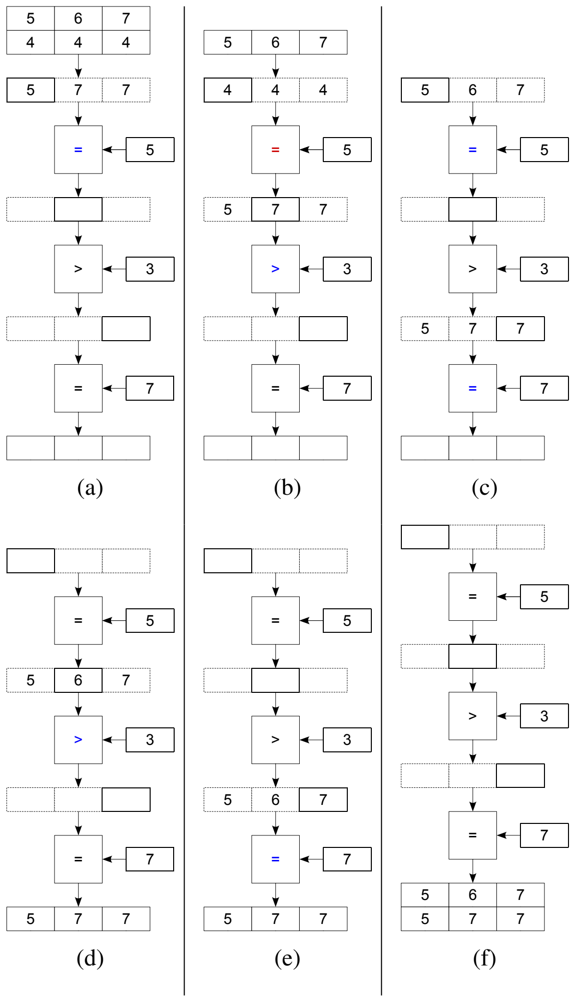


Fig. 7: Example of the Pipelined Filter.

## C. General Filter Expressions

Obviously, the described approaches only implement conjunctive conditions. In case of the Fully-Parallel Filter the conjunction is introduced by the match vector as it is checked against the constant vector *"1..1"*. In order to realize disjunction we can define more constant vectors and connect the results of all vectors by logical *OR*; e.g. $C_0C_1C_2C_3 \vee C_4C_5C_6C_7$ can be evaluated by comparing the provided match vector $m$ with the match masks *"11110000"* respectively *"00001111"* and connect both results by logical OR. Due to the structure of the Pipelined Filter where each local match is only forwarded to the next sub-filter the implied conjunction can not be broken up. Introducing a match vector forwarded from sub-filter to sub-filter would cause more utilization overhead. Thus, taking the advantages of FPGAs into account we can configure multiple Pipelined Filter, each representing one conjunction, in parallel and finally evaluate their results. For an expression containing up to 6 disjunctions only one LUT is needed to evaluate the global result. This introduces a negligible signal propagation delay caused by the logic.

## V. Evaluation

With respect to comparability, the implementation for the general-purpose CPU written in C follows the iterator concept [18]. This is recommended especially for big data sets as intermediate results are not computed completely at once. This results in a smaller memory footprint for the operators but also causes some overhead. As described earlier the modular approach using the generic operator template introduced the signals *read* and *valid* to control the data flow. Consequently, both implementations for CPU as well as for the FPGA are affected and thus fairly comparable. We suppose that the bindings needed for the filtering are located at the beginning of the bindings array. This can be done easily by connecting the circuit in this manner. Leftover data can be simply appended. To demonstrate the achievements under various conditions we varied several parameters. The *match rate* (MR) describes the selectivity of the filter expression. Furthermore, we introduce the parameter *mismatch position* (MP) to model the fact that depending on the filter condition the filter operator can detect a mismatch without evaluating the whole expression (e.g. a complex conjunction is always false if the first member is false). Thus, the mismatch position determines after how many (sub-)comparisons the filter operator can detect a mismatch, which allows it to skip the evaluation of the rest of the bindings array and go on with the next input ($MP_i$ indicates a mismatch at position $i$). For instance, in case of $MP_1$, the filter can detect a potential mismatch within the first evaluation step. Of course, in case of a match, the filter still has to compare the whole item, as it does not know about the data composition. Vice versa, if the mismatch position is set to the last binding in the bindings array, then the filter will detect the global mismatch in the last evaluation step of the whole expression. Thus, the computational costs are almost[2] equal compared to a global match. The proposed architecture is generic and independent of the width of the bindings and bindings arrays. However, for evaluation purpose we consider in the first part of the experiments a conjunction of 8 conditions and the mismatch position models the complexity of these conditions. Each data point represents the average of 10.000 single measurements respective to the parameters executed on a general-purpose CPU (Intel® Core™2 Quad Q9400, 2.66 GHz, 6 MByte Cache, compiled with GCC 4.8.2). The data was preloaded before measuring the time in order to ensure warm caches and to achieve the maximum performance on the CPU. The hardware-accelerated filter operators are described in the hardware description language VHDL and are synthesized for a Xilinx Virtex-6 FPGA (XC6VHX380T) [10]. The FPGA runs at 200 MHz which is the default clock rate provided by the equipped clock generator (13 times lower than the CPU) - unless otherwise specified.

### A. Throughput

By conception both approaches of the hardware-accelerated filter operators running at the same clock provide the same throughput. The difference in total processing time for the Pipelined Filter is negligible[3] and, thus, it achieves the same throughput as the Fully-Parallel Filter. Due to clarity we will reference in the next sections only to one of both hardware approaches. Later we will show the distinction between both hardware-accelerated approaches considering maximum frequency and area consumption. Fig. 8 shows the throughput of both FPGA-accelerated filter operators. Most noticeable is the fact that the FPGA performs independently of the data structure and shows for different MR and MP the same performance.

Fig. 9 outlines the total throughput of the software-based and hardware-based filter operators. Contrarily to the hardware-accelerated filter operators, the performance of the software solution is heavily effected by the composition of the bindings arrays. If the match rate is low then the position of the mismatch has a high impact on the throughput, e.g. if $MP_8$ and MR=0 then the approach on the CPU has to verify all subexpression in order to detect a mismatch in the last condition. Thus, the performance drops. For $MP_1$ and the same match rate this approach detects the mismatch in the first subexpression and consequently drops the bindings array immediately and requests the next item to be evaluated. With increasing the match rate the influence of MP becomes less for the software-based approach because a mismatch is detected in a late evaluation step and the computational effort to detect a match or a mismatch becomes the same. Fig. 10 shows the corresponding reached speedup.

### B. Scalability

At the beginning of the evaluation section we mentioned that the throughput of both approaches in the FPGA is equal at the same clock rate. Due to the different design ideas and their complexity the resource requirements are different. This results in a trade off between resource consumption and timing requirements which effects the maximum achievable clock

[2]*almost* because in case of a global match some overhead for providing the result is introduced.

[3]The pipelined filter introduces a delay of 8 clock cycles in order to fill the pipeline.

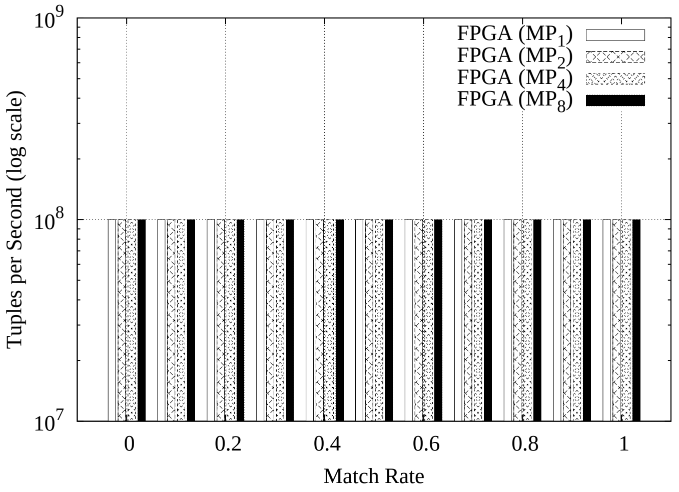


Fig. 8: Throughput of Fully-Parallel and Pipelined Filter executed on the FPGA. Both approaches achieve the same total throughput.

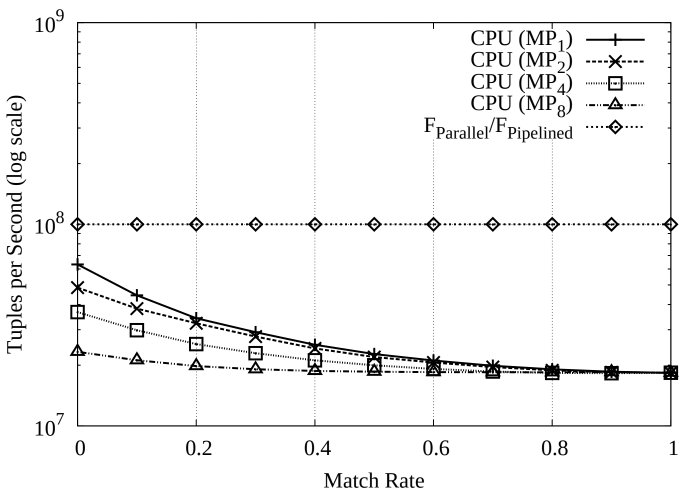


Fig. 9: Throughput of the software- and hardware-based filtering.

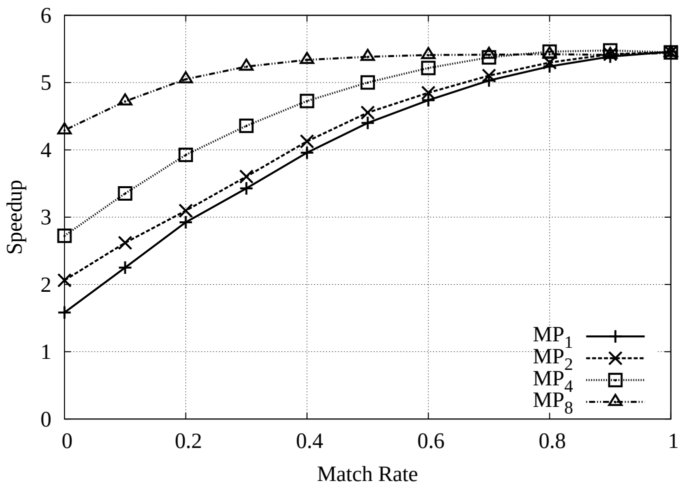
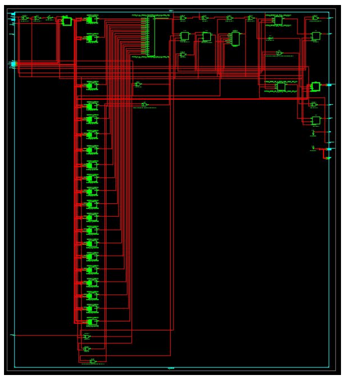
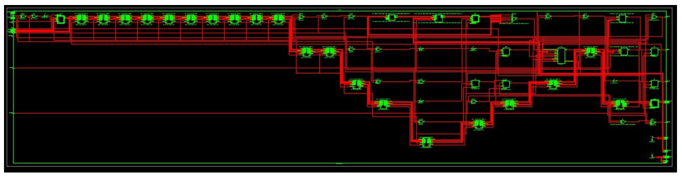


Fig. 10: Speedup factors achieved by the FPGA-accelerated Filter Operators.



(a) Fully-Parallel Filter



(b) Pipelined Filter

Fig. 11: RTL of Filter Operators.

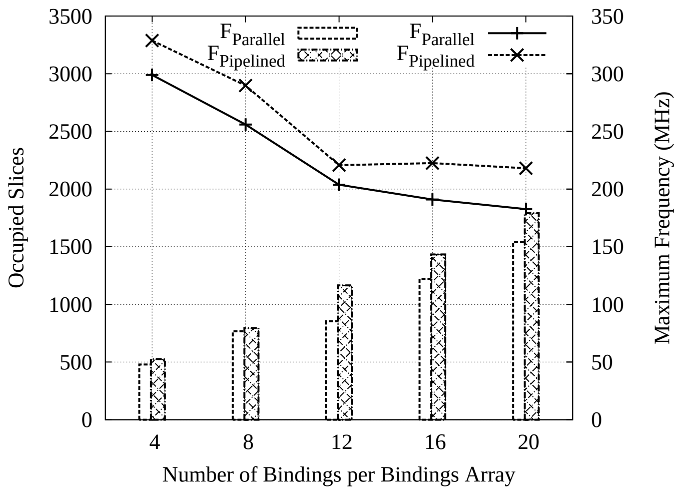


Fig. 12: Resource consumption (bars) and maximum achievable clock rate (lines) of both approaches (after *Place and Route*).

frequency and thus the effective total performance. To obtain the maximum achievable clock rate we synthesized the whole design under varying the data width of the bindings array. We tightened the clock frequency by constraining it in the designs UCF-file until the *place and route* is not able to meet the timing requirements anymore. Furthermore, we varied the design goal and chose the highest achieved value for each filter type and parameter. Additionally, we noted the resource consumption. Fig. 11 shows both filter operators composed of 20 sub-filters as RTL. The Fully-Parallel Filter (Fig. 11a) consists of multiple collateral sub-filters while the sub-filters in the Pipelined Filter (Fig. 11b) are connected as a chain. Fig. 12 shows the maximum achievable clock rate depending on the numbers of sub-filters respectively width of the bindings array. Each binding has a width of 32 bits. Furthermore, the number of utilized slices is shown. With increasing the amount of sub-filters the maximum frequency decreases for both approaches but the Pipelined Filter always reaches a higher clock rate (minimum 10% for 12 bindings per array) than the Fully-Parallel Filter. Latter one drops under 200 MHz for 16 or more sub-filters. As a drawback, the resource utilization of the Pipelined Filter is in general higher due to the additional needed registers between the sub-filter units. However, in case of 20 bindings per array the Pipelined Filter achieves a 19% higher clock rate than the Fully-Parallel Filter while using 16% more slices. One advantage of the FPGA-implementation is the fact that the throughput increases linearly with increasing the clock cycle. Thus, the Pipelined Filter reaches a higher throughput at its maximum possible clock frequency.

## C. Assessment of Results

The previously presented results show that evaluating filter expressions on an FPGA lead to a significant speedup in comparison to a general-purpose CPU system. While both approaches on the FPGA reach the same total throughput at the same clock rate, they slightly differ in scalability according to the width of the bindings array. They can provide one bindings array each two clock cycles. The theoretical throughput of one bindings array per clock cycle is not reached due to the flow control introduced by the operator template. If an operator consumed the preceding operators output then it raises the corresponding *read* flag. The preceding notices the invalidity of its current output and needs at least one clock cycle to provide the next bindings array. Thus, one clock cycle delay is introduced for synchronization between two operators. Without this synchronization it can not be guaranteed that no data gets lost. This is critical especially if lower operators can not consume intermediate results as fast as the upper operators produce them. If both, the current and succeeding operator, are able to handle incoming bindings array at a maximum speed of one bindings array per clock cycle then this can be realized by simply setting the *read* and *valid* flag to constant $'1'$. Consequently, the filter operator could double the total throughput to 200M bindings arrays per second. Additionally, an *almost overload* flag could be added. Succeeding, potentially slower operators could raise this signal after reaching a threshold. The preceding operators can slow down using the *read*/*valid* signals in order to avoid data loss. Furthermore, the filter expression is supposed to reduce the amount of data in an early step of the overall query evaluation. Depending on the selectivity of the expression the filter operator might consume data at a maximum speed of one bindings array per second but will not provide data at the same speed because many bindings arrays might not satisfy the filter expression and consequently will be dropped. This will be investigated in future work.

## VI. Summary and Future Work

In this paper, we presented two approaches of FPGA-accelerated filter operators. Both implementations defeat significantly the software solution executed on a general-purpose CPU. At a 13 times lower clock rate the Fully-Parallel and the Pipelined Filter achieve a speedup of more than 5 which results in a total throughput of 100M items per second at 200 MHz. Furthermore, we evaluated how both approaches differ in terms of resource utilization and maximum achievable clock rate. While the Pipelined Filter typically has a higher resource consumption it reaches higher clock rates. As the next logical step, after evaluating single operators we will orchestrate different operators (all implementing the presented Operator Template) to complex operator graphs at runtime. We expect this will result in further improvements by taking advantages of the previously described *intra-query / inter-operator parallelism* and *operator pipelining*. The established natural data flow *between* succeeding operators can not be achieve by CPU-based systems without additional overhead (e.g. intermediate results need to be stored in additional memory/register). Furthermore, in the final hybrid hardware/software system the impact of the communication (e.g. PCI Express) between the host system and the FPGA needs to be more investigated. As the FPGA works in parallel to the host system the execution of the hardware-accelerated operator graph can begin with the arrival of the first data items. After a latency a steady stream of data from the host system through the operator graph and back can be established and cascades the communication overhead. Additionally, the data source can be attached directly to the FPGA and the host system only receives the results (which are mostly orders of magnitude smaller). Thus, it is promising that the overall performance of query evaluation can be increased significantly by executing operator graphs on an FPGA.

## References

[1] 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)*, Helsinki, Finland, July 1 - 5 2013. IEEE.

[2] Hans-Otto Leilich, Günther Stiege, and Hans Christoph Zeidler. A search processor for data base management systems. In *Proceedings of the fourth international conference on Very Large Data Bases - Volume 4*, VLDB'1978, pages 280–287. VLDB Endowment, 1978.

[3] D.J. DeWitt. Direct - a multiprocessor organization for supporting relational database management systems. *Computers, IEEE Transactions on*, C-28(6):395 –406, June 1979.

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

[5] Rene Mueller, Jens Teubner, and Gustavo Alonso. Streams on wires: a query compiler for fpgas. *Proc. VLDB Endow.*, 2:229–240, August 2009.

[6] 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.

[7] 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.

[8] 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.

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

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

[11] Xilinx. Partial Reconfiguration User Guide. [Online] http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_5/ug702.pdf, April 2013.

[12] 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.

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

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

[15] Thomas Neumann and Gerhard Weikum. RDF3X: a RISCstyle Engine for RDF. In *VLDB*, Auckland, New Zealand, 2008.

[16] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: Sextuple indexing for semantic web data management. *Proc. VLDB Endow.*, 1(1):1008–1019, August 2008.

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

[18] G. Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. on Knowl. and Data Eng.*, 6(1):120–135, February 1994.