Hardware-accelerated Join Processing in Large Semantic Web Databases with FPGAs

Stefan Werner[1], Sven Groppe[1], Volker Linnemann[1], Thilo Pionteck[2]
[1]Institute of Information Systems, University of Lübeck, 23562 Lübeck, Germany
[2]Institute for Computer Engineering, University of Technology Dresden, 01187 Dresden, Germany
{werner, groppe, linnemann}@ifis.uni-luebeck.de, thilo.pionteck@tu-dresden.de

***Abstract* — The increasing amount of data to be processed by database systems asks for a continuous increase in processing power. While traditional system designs can hardly cope with these performance requirements, dedicated hardware accelerators provide the required processing power. However, dedicated hardware accelerators are inflexible and cannot be adapted to the requirements of a dedicated query. In this paper, a concept is introduced to improve the performance of a Semantic Web database by developing a flexible FPGA-based hardware accelerator. The feasibility of this approach is shown by implementing different types of join operators as one of the most important and most time consuming operators in query execution. The performance comparison between the proposed FPGA implementation and a software solution in C on a general-purpose processor shows a significant speed-up up to 10 times.**

Keywords — Join Processing; Semantic Web Database; FPGA; Hardware Accelarator; Reconfigurable Computing

I. INTRODUCTION

The current World Wide Web (WWW) was originally designed for human consumption and provides easy, instant access to a huge amount of online information. Search engines are used to cope with the information flood. These engines compare given keywords with plain text from a website. However, they are not able to decipher the meaning of the keywords. As a result of the ambiguity of keywords, users are left with unwanted results. Furthermore, keywords are occasionally substituted for other words with the same meaning, causing incomplete or missing results. The Semantic Web (SW) is intended to extend the WWW to a machine-understandable web "in which information is given a well-defined meaning, better enabling computers and people to work in cooperation" [1]. There is a growing number of applications that benefit from SW technologies and the data sets are becoming increasingly large (millions to billions [2]). A fast query execution on this vast amount of data is essential to develop high performance semantic web applications. In the past, the increase in the required processing power was complied by technological advances. Shrinking feature sizes enabled a continuously increase of the clock frequency until the *power wall* called for different solutions. As a consequence, the trend of multi-core CPUs raised up. They require parallelized software to achieve a performance enhancement. Furthermore, there are specialized hardware accelerators, such as graphics cards or network adapters, which have traditionally been used to solve specific tasks efficiently at a higher throughput. However, hardware accelerators are designed for a specific task only. Thus, usually they cannot be applied in application domains showing a huge variety in processing. With the aid of *Field-Programmable Gate Arrays* (FPGAs) the gap between the flexibility of general-purpose CPUs and the performance of specialized hardware can be closed. Through the possibility of (partial) reconfiguration [3], even at runtime, the FPGA offers more flexibility and can be used in different fields of applications. With regards to SW databases, the entire chip area of the FPGA can be used in different tasks, such as index generation and query execution. After the index is created, the operator graphs of different queries can be mapped on the FPGA and can be executed simultaneously. In this paper we will focus on the architecture of the hybrid hardware/software system and one of the key challenges in query execution: join processing. Experiments show that the implementations of various join algorithms in hardware outperform common CPU systems.

Related research about hardware-accelerated database operations is outlined in Section II. Section III explains the basics of semantic web databases and gives a short overview about the advantages of FPGAs. Afterwards the intended architecture of the integrated hardware/software system is introduced. The considered join operations that are implemented on an FPGA are described in Section IV. Section V shows a comprehensive performance comparison between the hardware-accelerated join and a general-purpose CPU system. Finally, the results are summarized in Section VI and potential future work is discussed.

II. RELATED WORK

This section briefly reviews related work regarding the use of FPGAs to enhance database applications. Ideas to increase the performance of database operations through the use of specialized hardware components first occurred during the late

1970's. Leilich et al. [4] presented a special processor particularly to improve search operations. The *DIRECT* architecture [5] considered multiple coprocessors to support intra-query and inter-query parallelism. Both systems retain the von-Neumann architecture, which limits the achievable performance gain in the area of stream-based operations. Because of the limited technological capabilities and high manufacturing costs, these approaches were not successful. The increasing availability of FPGAs in the last decade resulted in numerous innovations. Mueller et al. [6, 7] presented the query-to-hardware compiler *Glacier*, which is able to compose a digital circuit using various implemented operators. The compiler generates VHDL files which subsequently have to be synthesized and loaded into the FPGA. The synthesis is time consuming and thus is not ideal for online processing. The commercially available Netezza architecture [8] introduces the FAST-engine to reduce the amount of data by executing pre-computation steps (decompression, projection and restriction) while the data flows from the storage to the software system. Dennl et al. [9] introduced an FPGA-based methodology for accelerating SQL queries in relational database systems. The authors mentioned the necessity of join processing, but they focused only on projection and restriction such as in Netezza. Assuming a small data bus width, the data is split into several, properly sequenced parts (*chunks*). For handling of intermediate results, *spare chunks* are allocated in the data stream and each module knows when it has to insert its results. In order to build a pipeline of modules, the authors use a post-order traversal through the operator tree to preserve the dependencies of the operators. The *chunks* and the linearization of operators save chip area, but also decrease the overall performance. Teubner et al. [10] presented *Handshake join*, a window-based approach to execute joins on data streams. Handshake join allows items (players) *walking by* each other in opposite directions to *shake hands* with each item (player) they encounter. All items in a predefined window are considered in parallel to compute an intermediate result. Since the window size is limited by the chip size, this approach is not able to compute the join results of all items.

This paper presents a unique concept of a hardware-accelerated semantic web database. As one of the key tasks in query execution, this paper considers different join algorithms and compares the results with a low level software solution in C for general-purpose CPU's.

III. Conception of the Hardware Accelerated Semantic Web Database

This section introduces some of the basics about the semantic web and gives a short overview about the advantages of FPGAs. Finally, this section describes in which stages the FPGA shall be used to improve the performance of the Semantic Web database LUPOSDATE.

A. Semantic Web – an Extension of the World Wide Web

The Semantic Web intends to establish a machine-readable web by adding methods for systematically analyzing data and inferring new information. Therefore, the World Wide Web Consortium (W3C) developed a number of semantic web standards. Two of the most important ones will be introduced in the following section. The Resource Description Format (RDF) [11] 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 set of triples forms an *RDF graph* with the subjects/objects serving as nodes and the predicate serving as labeled directed edges. The same resources and literals are represented by the same unique node. The *SPARQL Protocol And RDF Query Language* (SPARQL) [12] is the most important query language for the semantic web and enables access to the data by selecting specific parts of the RDF graph. Similar to the RDF triples, the core component of SPARQL is a set of triple patterns *(s,p,o)*. In these triples, the known RDF terms are specified and the unknown terms are replaced by variables. Each triple pattern matches a subset of the RDF data when the terms are equal to the triple in the RDF data. The appearance of the same variable in several patterns implicitly leads to a join. As a consequence, join computations are frequent. Tab. I gives an example of three RDF triples and a SPARQL query. There are two triple patterns in the `WHERE` clause to constrain the input RDF data. The `SELECT` clause identifies two variables, `$a` and `$c`, that will appear in the query result. The first triple pattern matches the first two triples and the second triple pattern matches the last two triples in the RDF data. The variable `$a` is a common part of both triple patterns and implies a join between the results of each triple pattern. Thus, the final result of the SPARQL query is `{$a=<ID2>,$c=<ID5>}`. As already described in [13], SPARQL queries are suitable for parallel execution because each triple pattern can be applied on the RDF graph independently to other triple patterns.

TABLE I. Example RDF Data and SPARQL Query

RDF Data	SPARQL Query
<ID1> <records> <ID6> <ID2> <records> <ID5> <ID2> <origin> <DLC>	SELECT $a $c WHERE { $a <records> $c. $a <origin> <DLC>. }

B. Field-programmable Gate Arrays (FPGA)

FPGAs are integrated circuits which can be configured by the customer after manufacturing *in the field*. For prototyping and performance evaluation a Xilinx Virtex-6 FPGA is used within this paper. The chip area is divided into a grid of configurable logic blocks (CLB). In Virtex-6 FPGAs, each CLB contains 2 slices, which in turn consist of 4 look-up tables (LUT) and 8 flip-flops. Each CLB element is connected to a switch matrix for access to the general routing matrix [14]. The LUTs have 6 inputs and one output. Accordingly, each LUT can implement any arbitrarily defined 6-input Boolean function with one output. Furthermore, each LUT can be configured as two 5-input LUTs with separate outputs but common addresses or logic inputs. More complex functions can be implemented by combining several of these LUTs in a configurable interconnection network.

In addition to the CLBs, small and fast embedded memory blocks (BRAM) are distributed in the FPGA. This enables the logic to be located close to the BRAM, and considerably reduces the data access time. The performance advantages of FPGAs issue from the capability to execute thousands of computations concurrently. Through the massive parallelism and a relatively low clock rate the FPGA can achieve a high performance while consuming significantly less energy than GPU or multicore systems [15].

C. *Hardware Acceleration for LUPOSDATE*

The approach used in this paper to manage and query RDF data is based on the software system LUPOSDATE [16, 17]. Fig. 1 gives an overview of the functionalities and query processing phases of the LUPOSDATE system. After parsing the SPARQL query, redundant language constructs are eliminated to simplify the following process. Afterwards, an operator graph is generated in order to apply logical and physical optimizations. The logical optimization reorganizes the operator graph into an equivalent operator graph. It generates the same output for any input as the original operator graph, but needs less execution time to evaluate the query. Physical optimization aims to obtain the operator graph with the best estimated execution times by choosing a concrete implementation (physical operator) for each logical operator. Finally, the optimized operator graph is executed by using formerly generated indices. At this point, the approach intends to enhance the former model to create an integrated hardware/software system which uses the FPGA to accelerate index generation as well as query evaluation (indicated by FPGA boards in Fig. 1). In order to achieve this, it is necessary to add an additional step in the physical optimization to map the final operator graph to the available resources on the FPGA. It also allows the reuse of previously configured operators. The software and hardware components operate concurrently. This allows that the FPGA executes one or more queries while the software system parses and optimizes the next incoming query.

Fig. 2 shows different scopes of FPGA applications. It starts with an entire software system. Then, it is followed by a hybrid system, which only accelerates one expensive operator. Finally, it ends with a system where the software component simply acts as a controller. The dynamic partial run-time reconfiguration enables the opportunity to use the full chip area and resources of

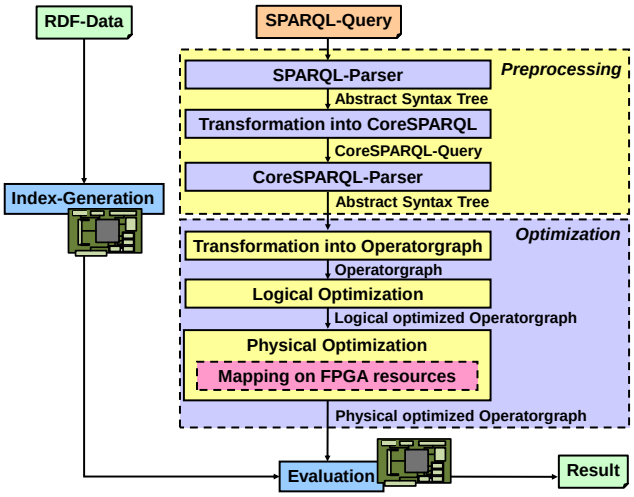

Figure 1. Functionalities and overview of query processing of the LUPOSDATE system

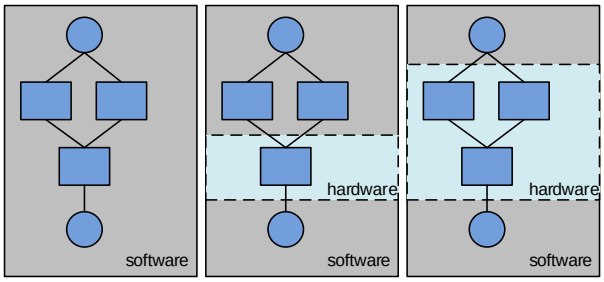

Figure 2. Scopes of FPGA application in query execution

the FPGA in the different phases. Reconfiguration plays an important role in query execution. The extended LUPOSDATE system generates an operator graph for a given SPARQL query and maps it on the FPGA resources. It is expected that not all queries benefit from hardware execution. In addition, reconfiguration time as well as communication latency must be evaluated to develop a heuristical query scheduler and to determine reasonable usage of hardware acceleration. Instead of various pre-synthesized operator graphs, each required single operator needs to be implemented and synthesized once in order to connect them at run-time to an operator graph. To achieve this, the chip area is separated into small tiles with one operator and one interconnection network per tile.

Fig. 3 shows an operator template with the interface signals. Each specific operator must implement this interface and requires at least one predecessor to provide an intermediate result on the left input. Additionally, a binary operator (e.g. join) must have a right predecessor. A unary operator (e.g. projection) does not utilize the right input. The interface of the result signals of the current operator is equal to the interface of the input signals. These can be used to provide the intermediate result as an input for the succeeding operator, or to retrieve the final results. Each set of signals contains the bit vector *data* and the bit signals *valid*, *finished* and *read*. The *data* vector represents the incoming bindings. To signalize the availability of valid data, the *valid* signal is used. The *finished* signal shows that there will be no more valid data in the future. Finally, the *read* signal invalidates the data and the preceding operator provides the next intermediate result. With these well-defined interfaces, operator graphs are easily created by connecting different operators. Fig. 4 outlines the described idea by using different operators in three operator graphs: *A*, *B* and *C*. In an operator graph, the operators are working in parallel and are similar to an operator pipeline providing *intra-query* and *inter-operator parallelism*. However,

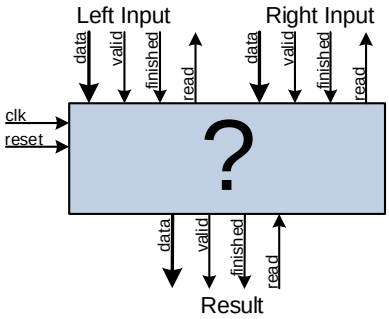

Figure 3. Operator template

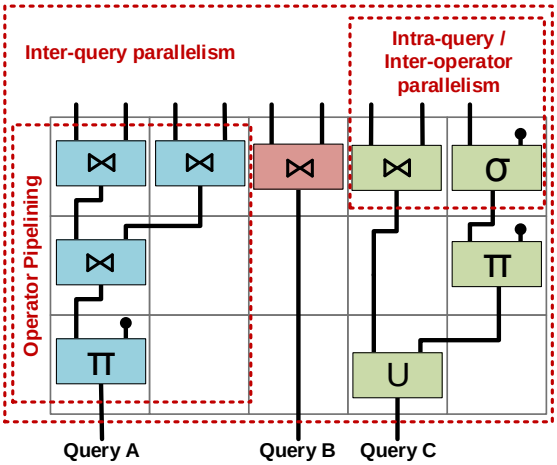

Figure 4. Multi query processing

at the beginning, the lower operators need to wait for the first results from the preceding operators. In addition to the operator parallelism, the different operator graphs are also acting in parallel (*inter-query parallelism*). Furthermore, while an operator graph is executed, a new operator graph can be mapped in another chip area. This allows masking the reconfiguration time because even the amount of time for small bitstreams cannot be neglected [3]. Fig. 5 outlines the idea of a query pipeline in the proposed hardware/software system. After parsing and reconfiguring the incoming query *A*, the software component is able to parse and prepare the next incoming query *B* in parallel, while query *A* is still in process. Similarly, query *C* can be configured for execution while both queries *A* and *B* are not yet finished. If one query is finished, the chip area will be released and can be used for another query. If multiple queries are incoming simultaneously, the partial reconfiguration time can be masked instead of waiting for previous queries to be completed.

The LUPOSDATE system can be configured to map the components of RDF triples to a unique numerical representation. The reversible bidirectional mapping is stored in a dictionary to enable the computation of specific results. Consequently, this approach leads to improvements in transfer speed and greatly reduces storage requirements. It is more convenient for FPGAs to handle numeric values as opposed to unlimited strings. Each variable is located in a specific position of an array which is called a bindings array. At the start of the evaluation, the position of the variables in the array is appointed and remains unchanged during the entire query. Fig. 6 shows a join operator with two incoming bindings. In both bindings, the variable *$b* has a specific value, which implies the usage of *$b* as a join attribute. For all the remaining variables, a specific value appears in only one of the two bindings (*$a*, *$d* in the left and *$c*, *$e* in the right preceding operator). Consequently, the join is executed if the join

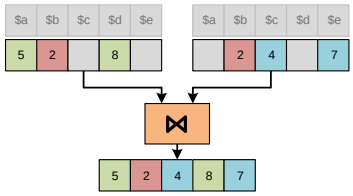

Figure 6. Join of two bindings

attribute *$b* in both bindings accomplishes the join condition. The remaining specific values are inherited into the result. The bindings array is directly mapped to the data path on the FPGA, which corresponds to the *data* signal (cf. Fig. 3). In this section, the general architecture of the intended hardware/software system is described. As a first step for realizing this system, the join operator was implemented. Because this is only one operator, the reconfiguration aspect is not considered in this context.

IV. Join Algorithms - Crucial for Query Evaluation

In this section, the various implemented join algorithms are described. The input is defined as two sets of bounded variables from preceding operands, where R is the left operand and S is the right operand. Variables which appear in both preceding operands are considered as *join attributes*.

The **Nested Loop Join** (NLJ) in its simplest form contains a nested loop iterating through the left and right operand. Despite the naive approach of join processing, the simplicity of the NLJ enables it to perform well for very small, unsorted data sets. Thus, this join algorithm is useful as a part of more complex algorithms. A drawback is that all elements of one operand are considered multiple times, resulting in the need to temporarily store the whole input of at least one operand. On the FPGA, the comparison of the join attributes of the two operands, the inheriting of the values to the result and the request of the next operand are parallel processed in a single clock cycle. Contrary, the sequential execution on the CPU requires multiple clock cycles to solve these tasks.

The **Merge Join** (MJ) requires data that is sorted according to the join variables. It initially reads the solutions of both preceding operands and checks if they are equal to the join attribute. Due to the sorting, all remaining solutions have to be equal or larger than the current solution. If the join attributes are not equal, the next solution of the operand with the smaller value is requested. Otherwise, if the join condition is fulfilled, all solutions with the same join attribute can be read to return the joined solutions. Like RDF3X [18] 6 indices corresponding to the six collation orders ((SPO), (SOP), (PSO), (POS), (OSP) and (OPS)) are used in the intended architecture to permit the frequent usage of the fast MJ. Through the knowledge about the order, the FPGA

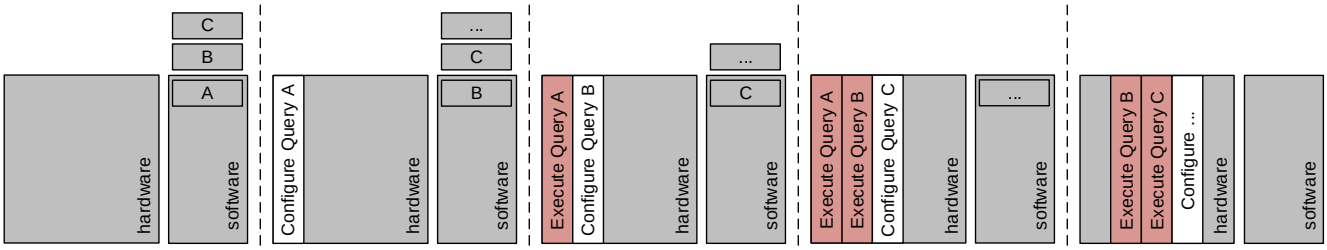

Figure 5. Query pipeline

implementation is able to generate results while searching for the next matching join pairs. This is useful if there are long rows of matching and non-matching join pairs.

The **Asymmetric Hash Join** (AHJ) is divided into two phases. In the build phase, each binding of the smaller operand is processed. Then, depending on a calculated hash value over the join attribute, they are stored in a hash table. The following probe phase iterates through the larger operand. It tries to find join partners in the previously created hash table of the smaller operand by using the same hash function on the join attribute. The AHJ is only applicable for equi-joins, and its performance is affected by the quality of the hash function used. Typically, well known hash functions contain many additions, multiplications and modulo operations. Even though there are dedicated multipliers/accumulators available in advanced DSP48E1 slices [19], expensive multiplications and additions should be avoided. Therefore, this approach uses a *hash mask* with the same data width as the join attribute to describe which bit of the join attribute will contribute to the final hash value. This step can be done in parallel and causes no delay cycles. Ideally, the number of 1-bits in the hash mask is equal to the address width of the hash map. Fig. 7 shows an example with a hash mask width and join attribute width of 8 bits. In this example, the hash value width is 5 bits, which is equal to the number of 1-bits in the hash mask. Furthermore, implementing the hash table in BRAM enables a fast hash table lookup in one cycle. The build phase increases the latency because in a complete operator graph, a succeeding operator has to wait until the preceding operator provides the first intermediate results.

The **Symmetric Hash Join** (SHJ) uses two hash tables in parallel to decrease the latency until the first results are provided. The two hash tables, H_L and H_R, store the bindings of the left and right operands, respectively. If a valid left binding is incoming, the hash value according to the join attribute is calculated and the binding is stored in H_L. Then, the probe phase is initiated to find a corresponding join partner in H_R. Additionally, the right bindings will be stored in H_R and checked against H_L. In the best case scenario, the first two bindings fulfill the join condition and can immediately be provided to the next operator. Besides the previously mentioned advantages, the FPGA implementation benefits from parallel calculating the hash values of the two operands and parallel searching for join partners in the corresponding hash tables. Again, the sequential execution on the CPU does not allow processing these steps in a single clock cycle.

Finally, all described join operators on the FPGA can take advantage from the composition of a single result. If a join is executed, the *data* signals of the two preceding operands just need to be redirected to the *data* signal of the result interface. This is easily done by a multiplexer. On the other hand, the software solution on the general-purpose CPU needs to iterate sequentially through the two operands to construct the result. Obviously, these steps require multiple clock cycles and have a strong impact when joins are frequent.

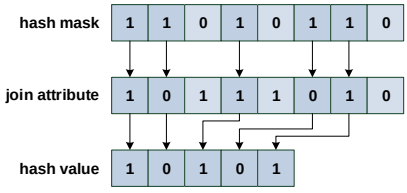

Figure 7. Hashing

V. Evaluation

The described join algorithms are implemented on the FPGA as well as software solution in C to obtain comparable results from a general-purpose CPU system (Intel Core 2 Duo T7500, 2.2 GHz, 4MB Cache, 3 GB RAM, Windows 7 (32 bit), compiler gcc 3.4.4). The implementation in C follows the usual iterator concept [20] which enables pipelining in the whole operator graph. As the solutions are computed one by one a huge size of intermediate results is avoided. The hardware accelerated join cores are implemented using the hardware description language VHDL. Fig. 8 shows the logical structure of the evaluation framework on the FPGA. Due to the previously introduced operator template (cf. Fig. 3), the join core needs no knowledge about its preceding operators (*Op X* and *Op Y*) and consumes their outputs as its two inputs. Vice versa the preceding operator *Op Z* consumes the results produced by the join core. To emulate the two preceding operators of the join operator, the data is located in the BRAM outside the join operator. The operators *Op X* and *Op Y* only provide these data. When the evaluation starts, the first bindings array is provided to the join operator. The evaluation ends when the join operator sets its *finish* signal, which means that the join operator has consumed all bindings of the preceding operators and will not provide more results. For the measurements on the FPGA the *Dini Group DNPCIe_10G_HXT_LL* board equipped with a *Xilinx Virtex-6 XC6VHX380T* is used. In order to achieve the maximum performance on the general-purpose CPU system, the join processing for each run is executed 100 times to ensure that the data is located in RAM and cache. The time it takes to load the input from the hard drive is not included. In this evaluation, 100 data sets are randomly generated and each bindings array contains 4 bindings, each 16 bits wide.

A. Throughput

This section focusses on the execution time of each join operator. Therefore, the size of one operand and the number of possible join attribute values (JA) are varied. The number of JAs has an impact on the number of possible join partners. A small number of JAs leads to a high probability of joins. For example, JA=1 means that each binding has the same join attribute value and each binding of the left operand needs to be joined with each binding of the right operand which corresponds to a Cartesian product. On the other hand, increasing the number of possible join attribute values leads to a decrease in join probability.

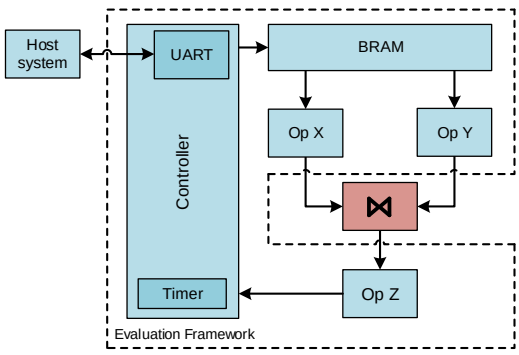

Figure 8. Evaluation framework (Connections between operators are simplified)

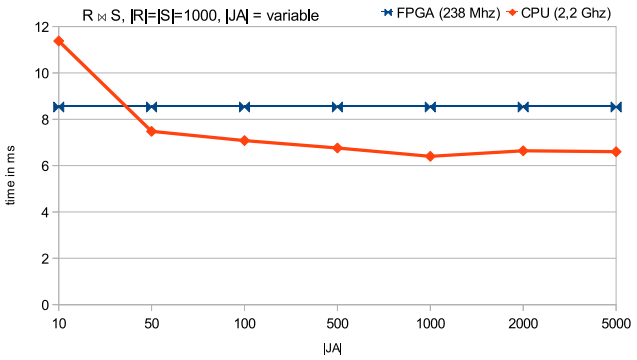

Figure 9. NLJ – Execution time – vary the number of potential join partners

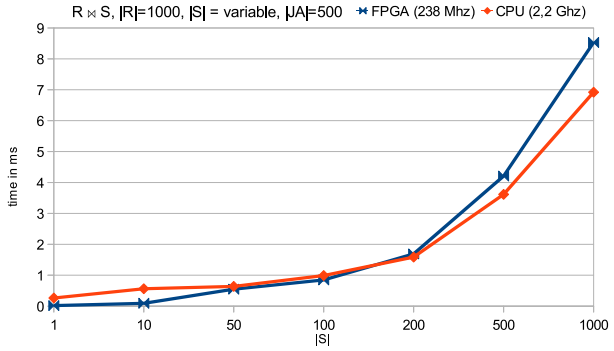

Figure 10. NLJ – Execution time – vary the input size of one operand

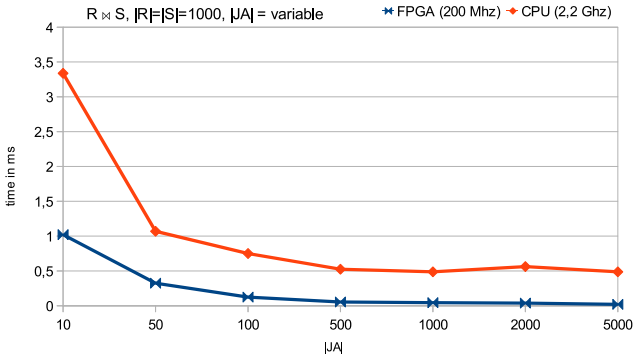

Figure 11. MJ – Exection time – vary the number of potential join partners

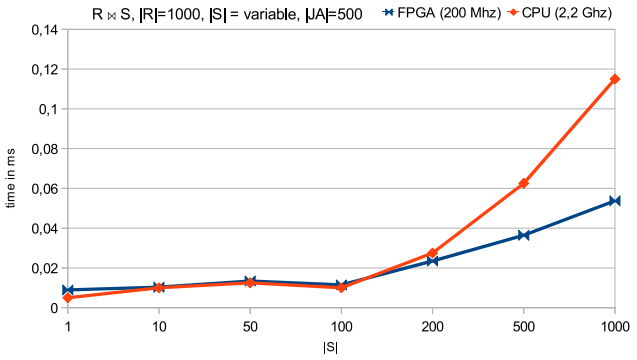

Figure 12. MJ – Execution time – vary the input size of one operand

1) Nested Loop Join

Fig. 9 and Fig. 10 show the comparison between the FPGA and the C implementation of the NLJ. For a small number of JA, joins are frequent. While the C implementation needs to iterate through each attribute of the bindings to join, the FPGA can take advantage of building the results in parallel. This drawback of the CPU decreases when the number of JA increases and the number of joins decreases. However, even at a lower clock frequency of 238 MHz, the FPGA shows a competitive performance.

2) Merge Join

As mentioned before, the input data for the MJ needs to be sorted. Therefore, the data is presorted. Thus, the time for sorting the data is not included in the total time. This assumption is reasonable based on the use of indices in the final intended system. Increasing the JAs in both approaches leads to a decrease of execution time (Fig. 11). Also, an increased speed up of 1.5 up to 3 by FPGA is noticeable. Fig. 12 shows the execution time depending on the input size $|S|$ of one preceding operator. While the performance of the CPU and FPGA is equal till $|S|=100$, the performance loss of the general-purpose CPU caused by frequent joins is much more pronounced than on the FPGA.

3) Asymmetric Hash Join

In all test cases, the AHJ on the FPGA defeats the software solution (Fig. 13) by a speed up of at least 1.3. If the frequency of joins is high which means a small |JA|, the gap between the software solution and the FPGA increases. The highest reached speed up is 5.7 (Fig. 14).

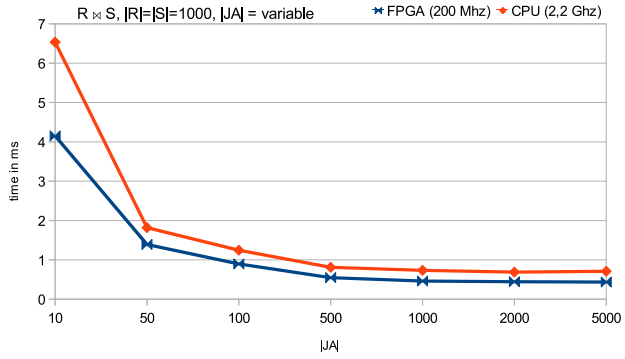

Figure 13. AHJ – Execution time – vary the number of potential join partners

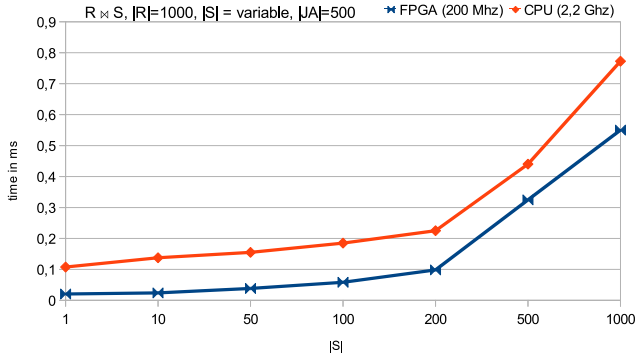

Figure 14. AHJ – Execution time – vary the input size of one operand

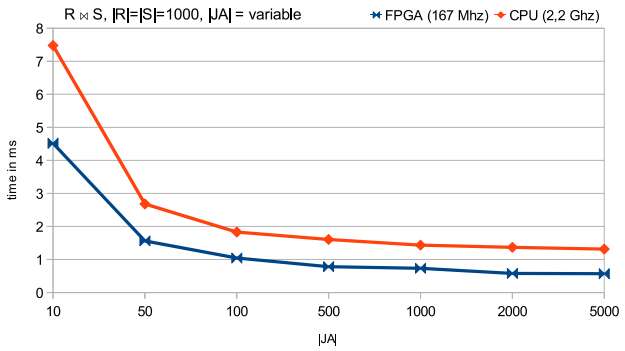

Figure 15. SHJ – Exection time – vary the number of potential join partners

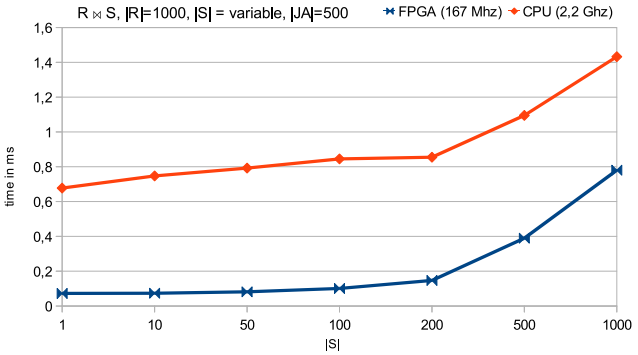

Figure 16. SHJ – Execution time – vary the input size of one operand

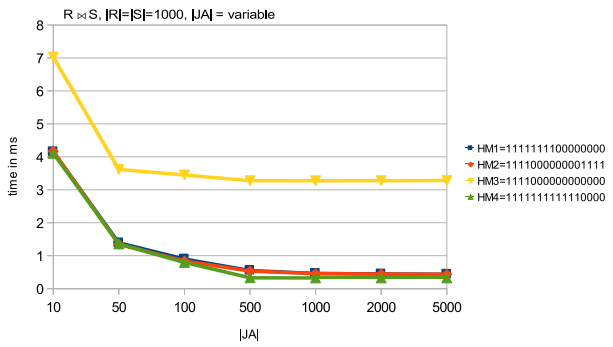

Figure 17. AHJ with different hash masks – vary the number of potential join partners

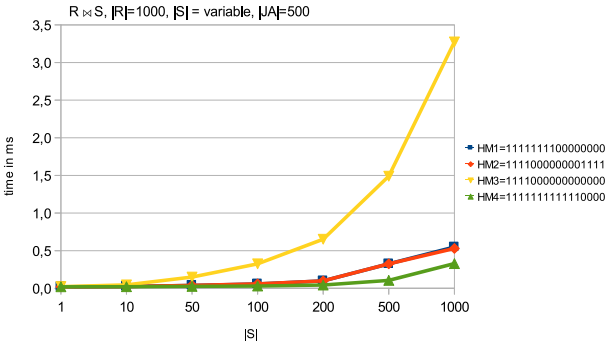

Figure 18. AHJ with different hash masks – vary the input size of one operand

4) Symmetric Hash Join

As mentioned before the SHJ uses two hash tables. Due to its higher circuit complexity, more logic resources are needed and the signal propagation delay increases. Thus, the SHJ can just be used at a frequency of 167 MHz in the FPGA. Even at this lower clock frequency, the FPGA reaches a speed up between 1.3 (Fig. 15) and 10.2 (Fig. 16) compared to the general-purpose CPU.

B. *Impact of the hash mask*

Certainly, the hash mask has an impact on the hash join performance. In the previous measurements the hash mask HM1=*1111111100000000* was chosen. That means only 8 bit of the 16 bit wide join attribute are considered to address the hash table. Fig. 17 as well as Fig. 18 show the execution time of the AHJ using different hash masks. Also HM2 uses 8 bit like HM1 but the 4 most significant as well as the 4 least significant bit are set. Because of the uniform distribution of the data both hash masks show the same performance. Contrary, HM3 considers only 4 bit and as expected the execution time increases significantly. HM4 uses 12 bit and thus is able to increase slightly the performance compared to HM1 and HM2.

C. *Device Utilization*

Besides the single throughput of one join core, the device utilization is an important fact. In join computation, many schemes are known to split and distribute the input data set over multiple processing units. Therefore, it is interesting to know how many join operators could fit into the FPGA. Tab. II shows the number of available Slices, LUTs and BRAM of the *Xilinx Virtex-6 XC6VHX380T-2 (FF1923)* used for evaluation. The numbers of utilized Slices, LUTs and BRAM units of each join operator are summarized in Tab. III. While the consumption of registers and LUTs is significantly lower, the available BRAM becomes a bottleneck. In that case, the BRAM suffices to implement 16 to 25 join cores, but a huge amount of logic area remains unused. Consequently, in the future, multiple memory layers and a caching strategy will be used to minimize the BRAM usage of each operator and to enable the usage of hundreds of join cores. Particularly, this is not practical with general-purpose multi-core systems and the energy consumption would be significantly higher.

TABLE II. CHARACTERISTICS OF THE XC6VHX380T-2

Available Slices	Available LUTs	Available BRAM Blocks
59,760	239,040	768 (each 36Kbit)

TABLE III. DEVICE UTILIZATION

Join Type	Used Slices		Used LUTs		Used BRAM Blocks	
	Total	*%*	*Total*	*%*	*Total*	*%*
NLJ	519	0.87	1,362	0.57	45	5.9
MJ	489	0.82	989	0.41	30	3.9
AHJ	603	1.01	1,200	0.50	30	3.9
SHJ	642	1.07	1,509	0.63	45	5.9

VI. Conclusion and Future Work

In this paper, the concept of an FPGA-accelerated hardware/software semantic web system was outlined. The FPGA is indented as reconfigurable hardware to accelerate index generation as well as query evaluation. As one of the most important operators in query execution, various join algorithms have been implemented as a prototype on an FPGA. The final evaluation showed that even at a lower clock rate, the FPGA implementations are able to provide at least a competitive performance such as general-purpose CPU with 10 times higher clock rate. In most cases, the FPGA outperforms the software solution and is able to provide a significantly increased performance. At this point, only one operator of a whole operator graph was examined. It is expected, that the composition of multiple operators to an operator pipeline will achieve further improvements and will be shown in additional experiments. The join operator was evaluated as a single piece of a whole operator graph. Consequently, the remaining operators (e.g. filter, projection, aggregate functions) need to be implemented to achieve a full support of SPARQL. Furthermore, at this point, run-time reconfiguration was not considered but it is an important issue when complete operator graphs can be configured. Additionally, it is expected that not all of the queries can benefit from the execution in hardware (e.g. reconfiguration time is longer than execution time). Hence, a query scheduler has to be developed. It would need to approximate the ratio of reconfiguration and execution time with the help of a weighting function to decide at runtime which operator graph should be executed on the FPGA. Another time consuming task of semantic web databases is the index generation. Furthermore, the developing of hardware-accelerated index generation is in progress. As shown the Merge Join is the fastest of the join operators. With the help of the indices the sorted data can be used to benefit as most as possible from the Merge Join. Currently, the proposed system is still an early stage prototype. For a faster communication, the used FPGA board provides an 8-lane Gen. 2 PCIe interface and 3 separate 10GbE LAN, using SFP+ modules. Because of the limited BRAM space, a multi-layer memory hierarchy with different sizes, speed grades (e.g. BRAM, QDRII, DDR3) and pre-loading strategies are necessary.

References

[1] T. Berners-Lee, J. Hendler, and O. Lassila, "The Semantic Web," *Scientific American Magazine*, 2001.

[2] Linked Open Data, "Connect Distributed Data across the Web," URL: /http://linkeddata.org/, 2012.

[3] K. Papadimitriou, A. Dollas, and S. Hauck, "Performance of partial reconfiguration in fpga systems: A survey and a cost model," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 4, no. 4, pp. 36:1–36:24, Dec. 2011. [Online]. Available: http://doi.acm.org/10.1145/2068716.2068722

[4] H.-O. Leilich, G. Stiege, and H. C. Zeidler, "A search processor for data base management systems," in *Proceedings of the fourth international conference on Very Large Data Bases - Volume 4*, ser. VLDB'1978. VLDB Endowment, 1978, pp. 280–287. [Online]. Available: http://-portal.acm.org/citation.cfm?id=1286643.1286682

[5] D. De Witt, "Direct - a multiprocessor organization for supporting relational database management systems," *Computers, IEEE Transactions on*, vol. C-28, no. 6, pp. 395 –406, June 1979.

[6] R. Mueller, J. Teubner, and G. Alonso, "Streams on wires: a query compiler for fpgas," *Proc. VLDB Endow.*, vol. 2, pp. 229–240, August 2009. [Online]. Available: http://portal.acm.org/-citation.cfm?id=1687627.1687654

[7] ——, "Glacier: a query-to-hardware compiler," in *Proceedings of the 2010 International Conference on Management of Data*, ser. SIGMOD '10. New York, NY, USA: ACM, 2010, pp. 1159–1162. [Online]. Available: http://-doi.acm.org/10.1145/1807167.1807307

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

[9] C. Dennl, D. Ziener, and J. 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*, vol. 20, pp. 45–52, 2012.

[10] J. Teubner and R. Mueller, "How soccer players would do stream joins," in *Proceedings of the 2011 international conference on Management of data*, ser. SIGMOD '11. New York, NY, USA: ACM, 2011, pp. 625–636. [Online]. Available: http://doi.acm.org/10.1145/1989323.1989389

[11] World Wide Web Consortium (W3C), "RDF/XML Syntax Specification (Revised)," URL: http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/, 2004, W3C Recommendation.

[12] ——, "SPARQL Query Language for RDF," URL: http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115, 2008, W3C Recommendation.

[13] J. Groppe and S. Groppe, "Accelerating Large Semantic Web Databases by Parallel Join Computations of SPARQL Queries," *ACM Applied Computing Review (ACR)*, vol. 11, no. 4, pp. 60–70, 2011. [Online]. Available: http://www.sigapp.org/acr/Issues/V11.4/ACR-11-4.pdf

[14] Xilinx, "Virtex-6 FPGA CLB User Guide," URL: http://www.xilinx.com/support/documentation/user_guides/ug364.pdf, February 2012.

[15] J. Fowers, G. Brown, P. Cooke, and G. Stitt, "A performance and energy comparison of fpgas, gpus, and multicores for sliding-window applications," in *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, ser. FPGA '12. New York, NY, USA: ACM, 2012, pp. 47–56. [Online]. Available: http://doi.acm.org/-10.1145/2145694.2145704

[16] J. Groppe, S. Groppe, A. Schleifer, and V. Linnemann, "LuposDate: A Semantic Web Database System," in *Proceedings of the 18th ACM Conference on Information and Knowledge Management (ACM CIKM 2009)*. Hong Kong, China: ACM, November 2 - 6 2009, pp. 2083–2084.

[17] S. Groppe, "LUPOSDATE Open Source," URL: https://github.com/luposdate/luposdate, 2012.

[18] T. Neumann and G. Weikum, "RDF3X: a RISCstyle Engine for RDF," in *VLDB*, Auckland, New Zealand, 2008.

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

[20] S. Groppe, *Data Management and Query Processing in Semantic Web Databases*. Springer Verlag, Heidelberg, 2011. [Online]. Available: http://-www.ifis.uni-luebeck.de/~groppe/SemWebDBBook/