2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines

On-the-fly Composition of FPGA-Based SQL Query Accelerators Using A Partially Reconfigurable Module Library

Christopher Dennl, Daniel Ziener, Jürgen Teich
Chair for Hardware-Software-Co-Design
Department of Computer Science
University of Erlangen-Nuremberg
Erlangen, Germany
Email: {christopher.dennl,daniel.ziener,teich}@cs.fau.de

***Abstract*—In this paper, we introduce a novel FPGA-based methodology for accelerating SQL queries using dynamic partial reconfiguration. Query acceleration is of utmost importance in large database systems to achieve a very high throughput. Although common FPGA-based accelerators are suitable to achieve such a high throughput, their design is hard to extend for new operations. Using partial dynamic reconfiguration, we are able to build more flexible architectures which can be extended to new operations or SQL constructs with a very low area overhead on the FPGA. Furthermore, the reconfiguration of a few FPGA frames can be used to switch very fast from one query to the next. In our approach, an SQL query is transformed into a hardware pipeline consisting of partially reconfigurable modules. The assembly of the (FPGA) data path is done at run-time using a static system providing the stream-based communication interfaces to the partial modules and the database management system. More specifically, each incoming SQL query is analyzed and divided into single operations which are subsequently mapped onto library modules and the composed data path loaded on the FPGA. We show that our approach is able to achieve a substantially higher throughput compared to a software-only solution.**

I. Introduction

Today's database systems have to execute many queries on large databases, e.g., Google has to answer about one billion search queries per day. To keep the response time low, queries must be executed as fast as possible. Queries are the interface between the database management systems, which hold the databases, and the database applications which rely on data stored in such databases. Therefore, speeding up query execution could increase the performance of database systems and the database applications could benefit from a lowered response time. Until now, software optimizations are done exhaustively to increase the performance of database systems but limits in terms of CPU clock frequency are reached, thus software execution time is limited, too. It is likely that data loads will increase further. Hence, we have to think of further possibilities to accelerate the query execution. On the one hand, today's multi-core CPUs offer the possibility to increase software performance if software is parallelized. On the other hand, we can take hardware accelerators into consideration to speed up execution by exploiting natural parallelism of reconfigurable hardware like FPGAs. FPGAs are already used in different fields of application, e.g., image processing, network packet processing, or other high-throughput applications. Consequently, it is worthwhile to investigate the usage of FPGAs in the field of database applications in general and for query execution purposes in particular. The possibility to change the configuration of an FPGA at run-time within milliseconds allows us to handle different queries by simply switching to the appropriate configuration. However, it is unaffordable to build one special accelerator for each possible query. Therefore, we make use of the possibility of partial reconfiguration. Partial reconfiguration allows us to switch the FPGA configuration partially, thus it is possible to load modules into the FPGA at run-time without the need of a complete reconfiguration. Furthermore, partial reconfiguration is faster than a full reconfiguration of an FPGA because the bitstreams are smaller.

In this paper, we present a module library consisting of such modules which covers an important subset of typical operations occuring in SQL queries. Furthermore, we present methods to map a query into a data path consisting of modules of our library. Consequently, we can cover a subset of possible queries and can execute a specific query by loading the appropriate, pre-synthesized modules into the FPGA at run-time.

In Section II we introduce related work. This includes existing approaches as well as the used technology. We continue in Section III with a problem definition and present architectural concepts in Section IV. Implementations are shown in Section V and the results are discussed in VI. This paper ends with a conclusion in Section VII.

II. Related Work

There exist already some static approaches which use FPGAs for query processing, i.e., they do not utilize partial reconfiguration which leads to some drawbacks. The first approach is implemented by *Glacier* [1], a query-to-hardware compiler and library. It implements a set of streaming operators which can be composed to a digital circuit. This circuit is able to execute a specific query. *Glacier* converts an algebraic query plan, which represents such a query, into an appropriate digital circuit by generating VHDL files. Afterwards, the circuit is synthesized and the resulting bitstream is loaded into the FPGA which is now ready to execute the query on an appropriate data stream. However, this approach is only suitable for scenarios with few queries which are known in advance. Synthesizing digital circuits is

978-0-7695-4699-5/12 $26.00 © 2012 IEEE
DOI 10.1109/FCCM.2012.18

a time-consuming step which can take minutes to hours, thus *Glacier* is unapplicable for applications with many unknown queries and high throughput requirements. We will show how to overcome this drawback by utilizing partial reconfiguration.

Another approach is implemented by Netezza's *FAST*-engines [2] as shown in Figure 1.

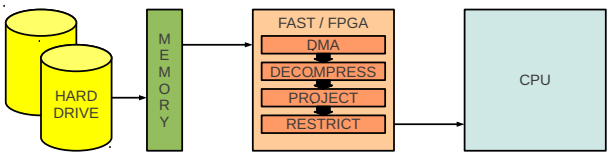

Figure 1. Overview of Netezza's *FAST*-engine [2] providing a fixed pipeline with several computing engines

It provides a pipeline consisting of several computing engines which is reconfigurable by setting internal registers. It is still a static design as no FPGA-based (partial) reconfiguration is applied at run-time. One purpose of a *FAST*-engine is to reduce the amount data, which is passed from the disk to the CPU, by performing projections and restrictions on the requested table data beforehand. Furthermore, a *FAST*-engine hides the slow I/O transfer rate of a hard-disk by utilizing its decompression engine. This engine decompresses data stored on the disk into uncompressed data stored in RAM, thus the I/O bottleneck of disks is weakened. We cannot state the exact capabilities of each engine because they are neither public nor documented. However, the design of a fixed pipeline forces the projection engine only to drop columns which are not needed later anymore. This is non-optimal in terms of amount of data reduction. Furthermore, following our own approach, a trade-off between flexibility of the hardware and complexity of the queries needs to be made. The more flexible the hardware is to cover a high variety of queries, the less is the maximum complexity of a query because more flexible hardware needs more resources than specialized hardware. We show how partial reconfiguration can be used to resolve the drawback of a fixed pipeline and allows us to handle both complex and non-complex queries by just selecting appropriate modules.

The design and implementation of our partially reconfigurable system along with its modules requires also to solve dynamic interconnect problems. For this purpose, we propose to use the *ReCoBus*-Builder tool as described in [3]. This framework provides communication macros to exchange data between the static part of a system and partial modules as well as macros to guide the Place & Route step during implementation. Especially, we rely on the I/O Bar macro which is used to design pipelined structures. I/O Bars are a composition of several pre-routed wires whose purpose is to pipe a data stream. Partial modules can be plugged on and off the I/O Bar at run-time and have the possibility to bypass or manipulate the data stream.

III. Problem Definition

First of all, we have a closer look on the *Structured Query Language* (SQL) [4] and its offered operations and data types. Afterwards, we select a powerful subset for which we will present a methodology for automatic accelerator generation on the basis of a partial reconfigurable architecture. SQL queries provide the interface between database applications and data stored in relational databases. In relational databases, data is organized as *tables* which are defined by one or more *attributes*. Each row of a table corresponds to one specific data tuple stored in this table and each attribute has a specific data type like integer, string or float. SQL defines CRUD operations (create, read, update, delete) to interact with the data stored in such tables. Furthermore, the SQL offers operations to constrain the so-called CRUD operations, e.g., it is likely for a delete operation not to delete all tuples of a table but only tuples which meet a specific requirement. Amongst others and looking at benchmarks like TPC-H [5], there are four commonly used operations in query processing: projections, restrictions, aggregations and joins.

Projections: Projections are used to select specific attributes of a tuple. In other words, projections select specific columns of a table. They can considerably decrease the amount of data which is passed from the database to the application because only needed data is retrieved, thus I/O load can be reduced. In the SQL syntax, they are expressed using the `SELECT`-statement.

Restrictions: Whenever a database application wants to retrieve or modify certain tuples which have to meet one or more requirements, restriction operations are used. Restrictions are Boolean expressions and only tuples which fulfill a restriction are considered for further processing. Whereas projections select specific columns of a table, restrictions select specific rows, thus data can be reduced further. To define a restriction in SQL syntax, the `WHERE`-statement is used.

Aggregations: Aggregations are functions to combine several values of one or more attributes of a group of data tuples to one value. Figure 2 shows an example usage of the aggregation function `SUM()`.

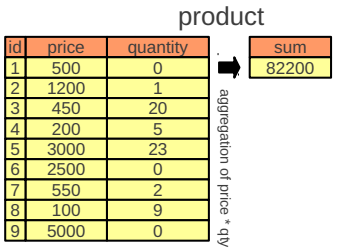

Figure 2. Application of the aggregation function *SUM()*; the product *price * quantity* of each tuple is summed up

Joins: Joins combine two tables to one compound table containing the combined information of both tables.

Formally, a join is the cartesian product of two tables followed by a restriction.

In this paper, we apply Netezza's concept of using FPGAs as a precomputational step between a data source and a data sink. Moreover, we will focus on projections and restrictions and show how partial reconfiguration can be used to assemble flexible computation pipelines at run-time. Furthermore, we show how restrictions of different complexities can be handled by offering an appropriate module library for different needs. However, the SQL defines a wide variety of possible data types and operators working on these types. Thus, we select a representative subset with common types and operations. In this work, we offer the following operators for restrictions:

- $+, -, *,$
- $AND, OR, NOT, XOR, NAND, NOR,$
- $<, \leq, =, \neq, \geq, >$.

Furthermore, in this first version of our module library we designed modules to support integers and fixed-length strings. The comparison operators are defined on both types whereas the arithmetic ones work only on integers. However, our library can be easily extended to support other types, e.g., float, by simply adding appropriate partial reconfigurable modules.

Figure 3 shows an overview of our approach. An SQL query is transformed into a hardware pipeline consisting of partially reconfigurable modules. The assembly of the (FGPA) data path elements is done at run-time using a library of predefined SQL operator modules that will be explained in the next section. This way, an input table is filtered by a restriction term, thus the outgoing tuple data stream contains only records which fulfill the restriction.

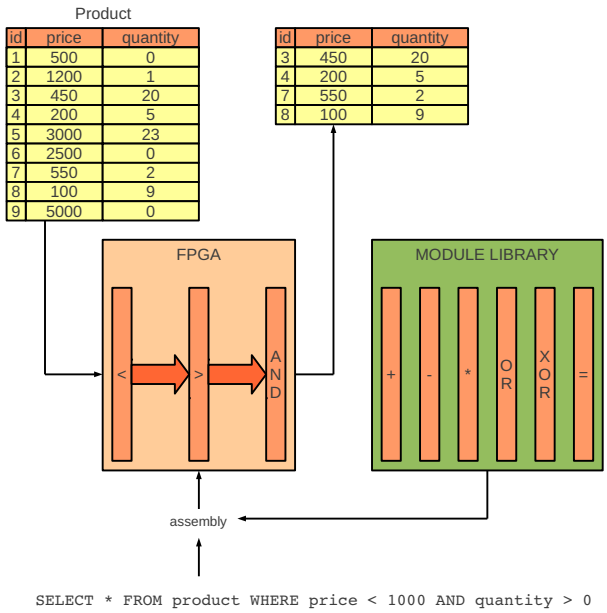

Figure 3. Query transformed into a hardware module pipeline; Tuples from an input table are filtered by a restriction term

IV. Architectural Concepts

Before being able to design the library of partial modules, the architecture and data processing concepts will be explained. These include a module interconnect interface called I/O Bar, which must be the same for all modules, as well as processing the data in a suitable way. The I/O Bar provides the following interface:

- Control bus with several control signals:
  - `global_reset` to reset all modules in the pipeline
  - `local_reset` to reset a specific module in the pipeline
  - `data_valid` and `config_valid` to indicate the arrival of a new tuple or configuration data on the data bus
  - `project` to tell the projection module whether to keep or drop the data on the data bus
- Opcode bus to address configuration registers inside the modules
- Data bus to transport tuple and configuration data

For data processing, we assume a query data stream is formed and flowing from some data source to some data sink through the FPGA. One pattern of query processing of real databases is that usually tuples are wider than usual data bus widths. Hence, tuples must be split into several parts, which we call *chunks* in the following and chunks need to be properly sequenced, see, e.g., Figure 4. A chunk has the same width as the data bus and contains certain attributes of a tuple. Now, modules can access exactly these attributes needed for operation by fetching the appropriate chunks out of the data stream. Each chunk of a tuple has a unique index and the modules are configured with the appropriate chunk indices, e.g., if a adder module needs chunk 2 and 4 for its operator inputs, it is configured with index 2 for the first operand and index 4 for the second operand. During stream processing, each module counts the bypassing chunks and loads its operand registers if an appropriate chunk arrives. Furthermore, the modules are configured with the count of chunks per tuple. This way, a module can recognize when a tuple ends and a new tuple begins. This information is important because projections and restrictions are evaluated row-wise, i.e., each tuple is evaluated independently. Operands are not forced to be tuple attributes only but also constant values which can be configured beforehand. As stated above, single attributes must be located in single chunks in order to be able to address them via indices but this is not suitable for string attributes because strings are usually longer than the size of one chunk. In this case, the indices denote the start and the end chunk of a string but we have to assume that one string is contiguous. However, integer attributes can be smaller than one chunk or several integer attributes are located inside one chunk, therefore modules should provide the possibility to cut out smaller attributes of chunks. Otherwise, such attributes have to be aligned to the chunk width, which

would lead to a space overhead. Figure 4 shows a snapshot of a typical data processing stream. The modules are configured with the appropriate chunk indices which denote the position of the operands inside one tuple.

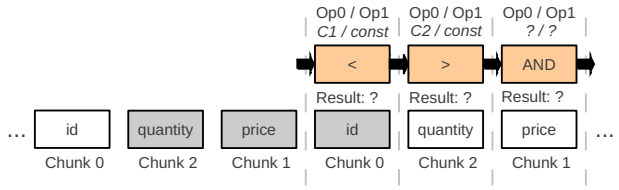

Figure 4. Snapshot of processing tuples of data; In the example, tuples consist of three attributes and three chunks and the modules are configured with indices which denote the position of the operands. The shown data path consists of three cascaded modules for processing the query introduced in Figure 3. It is not determined yet how results are handled. Figure 5 shows the completed snapshot.

The more difficult part is the handling of intermediate results which arise when we want to evaluate more complicated restriction terms. In general, there are two possibilities to handle this situation: The first alternative is to pass intermediate results in parallel to the data stream. The second one is to include the results into the data stream. The first possibility requires a second data bus which leads to a higher resource usage of the system whereas the second possibility decreases the maximum throughput of the system because the tuples get blown up. We would prefer the first method if resources would be available, but for our prototype implementation, we stick to the second one because there are typically not enough resources available in a target FPGA. To insert intermediate results inside the tuple stream, we allocate a sufficient count of *spare chunks* per tuple beforehand and configure the modules with the right result index. Hence, each module knows when it has to insert its results. In order to retrieve the right result, the result index must not be smaller than the operator indices. However, they might be equal, thus old intermediate results can be overridden by new ones. Figure 5 shows an adjusted tuple stream processing. Modules are configured with an additional result index. Thus, they can place their results inside the stream into spare chunks which are allocated beforehand.

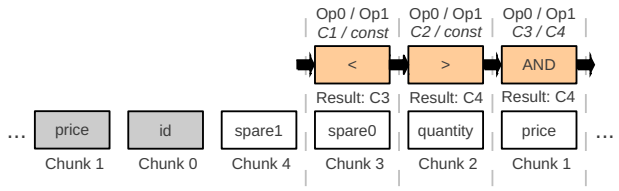

Figure 5. Adjusted data processing, which was introduced in Figure 4, to handle intermediate results; *spare chunks* are appended at the end of each tuple to provide slots for intermediate results

As soon as the tuples leave the pipeline, the spare chunks are dropped and only useful data is transferred for further processing. To decide whether a tuple fulfills a restriction or not, we have to take a look at the spare chunk which holds the result of the last operator of the pipeline. The result is either *true* or *false*. In some cases, it is possible to use common tuple chunks as spare chunks. Whenever a chunk will be projected by a projection module and is not used further, it can possibly be used as a spare chunk. As stated above, this possibility is not for free because we suffer a decreased throughput which is quantified by Equation (1).

$$\text{Eff. Throughput} = \frac{TC}{TC + SC} \cdot \text{Max. Throughput} \qquad (1)$$

- TC: Number of Chunks per Tuple
- SC: Number of Spare Chunks per Tuple

Implementing a projection module into the pipeline is rather easy because we only have to evaluate the `project` signal on the control bus. Furthermore, we can freely place it before and/or after the restriction pipeline. In order to assemble an appropriate pipeline which evaluates a restriction, we need several information and methods, respectivly. We have to compute the minimal count of spare chunks which have to be appended at the end of each tuple as well as the assignment of modules to these spare chunks. Furthermore, we have to derive a linear order of operations which respects the dependencies inside the restriction term while the term is evaluated. Thus, we can assemble a linear pipeline consisting of several modules.

A. Computing Minimal Count of Spare Chunks

In order to solve these three problems, we have to choose a sufficient data structure to represent restrictions and need algorithms which work on this data structure. We represent a restriction by the help of a *Binary Expression Tree*, which is recursively defined by Equation (2):

$$T = Operand \mid T \times Operator \times T \qquad (2)$$

Either tree T is an operand, or it is an operator with a left and right sub-tree. I.e., operands are the leaves of the tree and operators are the inner nodes. Figure 6 shows an example.

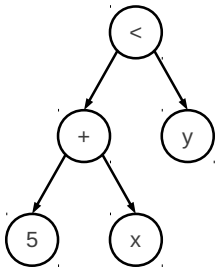

Figure 6. Binary expression tree representation of the restriction $5+x < y$

We may now compute the minimal count of spare chunks by applying the recursive algorithm denoted by Equation

(3) [6]. We have to call this algorithm on the root node of a given binary expression tree.

$$\#reg(v) = \begin{cases} 0, & isLeaf(v) \\ max(\#reg(l(v)), & \\ \#reg(r(v))), & \#reg(l(v)) \neq \#reg(r(v)) \\ \#reg(l(v)) + 1, & \text{otherwise} \end{cases} \quad (3)$$

- $isLeaf(v)$: Function to determine whether node v is a leaf or not
- $l(v)$, $r(v)$: Functions to retrieve the left / right sub-tree of a node v

If a node v of the tree is a leaf, it is an operand, and we do not need an additional spare chunk. If the count of spare chunks of the sub-trees differs, we can take the maximum for the actual node because one of the sub-trees has free spare chunks. We only need an additional spare chunk if both sub-trees have the same count, thus we cannot reuse a free spare chunk of one of the sub-trees.

B. Operator Node to Spare Chunk Assignment

Knowing the count of spare chunks allows us to compute an assignment of operator nodes to spare chunks. Thus, we can configure each operator module with the right operand and result indices. Algorithm 1 shows a recursive algorithm which assigns spare chunk indices to operator nodes.

Algorithm 1 Function to assign spare chunk indices to operator nodes

```
function ASSIGN(v, current, max)
    v.spare_index ← max - current
    if !isLeaf(l(v)) && !isLeaf(r(v)) then
        assign(l(v), current, max)
        assign(r(v), current+1, max)
    else if !isLeaf(l(v)) then
        assign(l(v), current, max)
    else if !isLeaf(r(v)) then
        assign(r(v), current, max)
    end if
end function
```

- v: Currently visited node
- current: Current index counter
- max: Maximum index

To get a complete assignment, we have to call $assign(root(T), 0, \#reg(root(T)) - 1)$. The algorithm traverses each inner node and labels the currently visited node with an index. The index denotes the spare chunk which holds the result of the assigned node. It is ensured that the index of a node is greater or equal to the index of the sub-trees, thus a result is not written until both operands are fetched.

C. Linearization of a Binary Expression Tree

In order to build a pipeline of modules, we need a linear order of operations which preserves the dependencies of the operators. A operator must be placed later in the pipeline than its child nodes. We choose a post-order traversal of the tree because it generates the reverse polish notation of a binary tree which corresponds exactly to our dependency requirement. In post-order traversal, first the left and right sub-trees are processed and afterwards the node itself. The traversal can omit the leaves because they are no operators. Figure 7 shows the application of all three algorithms on the restriction $(((a \cdot b) - ((c - d) \cdot e)) + f) < (g + (h - i))$.

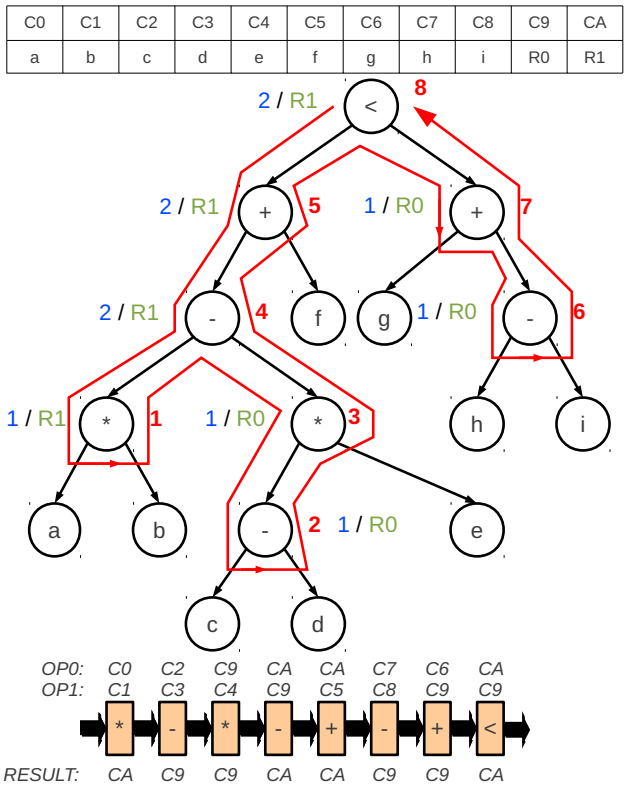

Figure 7. Application of all three algorithms and the resulting pipeline

So far, we have shown how to transform an SQL query containing projections and restrictions into a flexible partially reconfigurable module pipeline which processes a database tuple stream. Now, we can take a closer look on the prototypical implementation and results.

V. Prototypical Implementation

In this section, we describe a prototypical implementation of our system and module library. This includes the partitioning of the FPGA into static and dynamic regions as well as a sufficient choice of bus widths for the I/O Bar. We chose the XUP Virtex-II Pro Development System of Xilinx with an XC2VP30 FPGA as available target platform. Note that the proposed methodology works exactly the same on recent Virtex5 / Virtex6 platforms.

First of all, we have to partition the FPGA into a static and dynamic reconfigurable part and design the I/O Bar. As stated above, the I/O Bar consists of three different buses and the bus widths are defined as follows:

- 8 bit control signals, which leaves additional spare control signals for possible future use. One separate bit is used per control signal,
- 16 bit opcode signals to address registers inside the partially reconfigurable modules, and
- 32 bit data bus to transport tuple and configuration data.

These widths are chosen to keep the resource overhead of the communication macro small, allowing us to use more resources for the implementation of our partial modules. In detail, we use one slice per row and module, i.e., two bits per row and module. We split up the I/O Bar into a north and a south one because the PowerPC Cores divide the available space inside the FPGA. The north I/O Bar is connected to the south one on the east side which allows us to treat the both I/O Bars as a single one. The data is flowing from west to east in the north part and from east to west in the south part. Figure 8 illustrates the system design and shows a possible placement of modules.

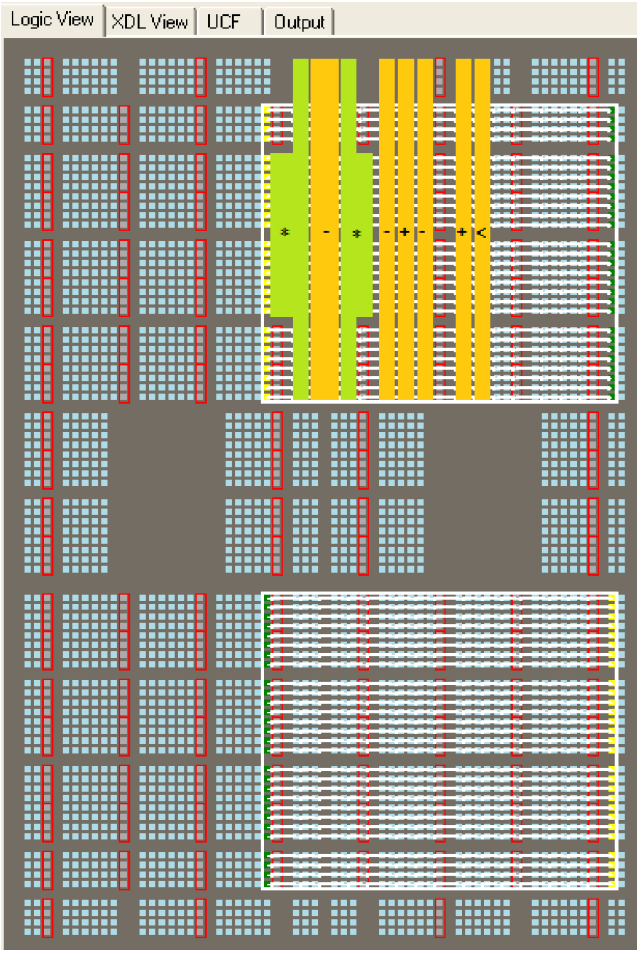

Figure 8. Partitioning of the FPGA and I/O Bar layout; the shown module pipeline corresponds to the restriction shown in Figure 7

Each of the two dynamic regions has a size of 32x24 CLBs and contains 5 BlockRAM / Hardware Multiplier columns. Each CLB contains 4 slices with each slice having 2 LUTs and 2 flip flops. Due to the reconfiguration scheme of the FPGA [7] and the limitations of the ReCoBus-Builder, the width of a partial module must be a multiple of 2 CLBs. The frequency of the chosen clock is 100 MHz, i.e., the maximum achievable throughput is 100 MHz · 32 bit = 400 $\frac{MB}{s}$. According to Equation (1), the effective throughput decreases in dependence of the number of spare chunks.

A. Arithmetic-Logical Modules

We implemented a set of arithmetic-logical modules that offer different feature combinations. Table I shows the investigated combinations.

Table I
INVESTIGATED MODULE FEATURE COMBINATIONS

Immediate Values	ALU	Byte-wise Access
✓	✓	✓
✓	no	✓
no	no	no

Having a programmable ALU inside a module has the advantage that the operation can be changed by simply switching the opcode of the ALU once the module is plugged onto the I/O Bar. No further reconfiguration is required. Omitting constants and byte-wise access is useful for inner nodes which do not have constants or small attributes as inputs. Amongst others, this is the case for inner nodes without leaves. The more features are omitted, the narrower a module becomes, i.e., a restriction can be more complex but the reconfiguration effort to reconfigure a new data path increases. Figure 9 shows the data path of an ALU module supporting all discussed features.

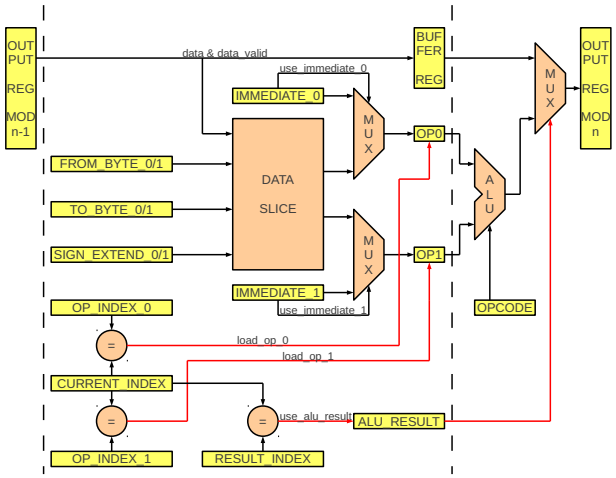

Figure 9. Arithmetic-logical module with all features; by omimitting certain features of this module, the module gets narrower and needs less resources

It has a latency of two cycles but is fully pipelined. The operand registers are loaded when the current index is equal to the operand indices. If the result index is equal to the current index, the ALU result is forwarded instead of the current chunk. The data slice module cuts smaller attributes

out of chunks at the provided byte boundaries. Furthermore, it can sign-extend the operands. A multiplication module is relying on hardware multipliers inside the BlockRAM columns, which puts location constraint on modules with ALU or the multiplication module.

B. String Comparison Modules

Two different string comparison modules have been investigated, one based on flip flops and another one based on BlockRAM for storing the string attributes. The former one has no location constraint but can hold only small strings up to 16 single-byte characters whereas the latter one relies on BlockRAM and can hold strings up to 8192 single-byte characters. Both modules are fully pipelined, thus all modules in the library achieve the same throughput, i.e., may be freely chosen. A distributed RAM version is not investigated yet.

VI. Results

We investigate several aspects of our system. We take a look at the resource usage of the modules and the consequential complexity of restrictions. The overhead which is brought with the dynamic approach is also a topic as well as the execution time of different queries compared to a software counterpart. We assume the data to be located in DDR memory for our experiments. To fully utilize the maximum throughput of this system in applications, the storage architecture must provide the corresponding bandwidth.

A. Resource Usage & Limitations

First of all, we show the resource usage of our module library. Table II, Table III, and Table IV show the usage of the arithmetic-logical modules. In 6/4/2 CLB columns (32 CLBs per column) we have 768/512/256 slices available, respectively for each module. Furthermore, the narrowest modules have the best slice utilization, i.e., the overhead in comparison to a complete static design is very low. As stated above, the module with ALU and the multiplication modules require an additional BlockRAM column with its integrated hardware multipliers.

Table II
Resource usage of the ALU module supporting all features (Figure 9)

Slices	CLB Columns	Slice Utilization
531	6	69%

Table III
Resource usage of module without ALU

Operator	Slices	CLB Columns	Slice Utilization
$+, -$	321	4	63%
AND, OR, XOR	305	4	60%
NOT	238	4	46%
$NAND, NOR$	305	4	60%
$<, \leq$	322	4	63%
$=, \neq$	314	4	61%
$\geq, >$	322	4	63%
$*$	321	4	63%

Table IV
Resource usage of module without ALU, byte-aligned access, and immediate values

Operator	Slices	CLB Columns	Slice Utilization
$+, -$	239	2	93%
AND, OR, XOR	222	2	87%
NOT	194	2	76%
$NAND, NOR$	222	2	87%
$<, \leq$	240	2	93%
$=, \neq$	230	2	90%
$\geq, >$	240	2	93%
$*$	238	2	93%

Table V
Resource usage of string comparison modules

Method	Slices	CLB Columns	Slice Utilization
Flip Flop	366	4	71%
BlockRAM	340	4	66%

The resource usage of the string comparison modules are shown in Table V, respectively. The string comparison module based on BlockRAM requires an additional BlockRAM column.

As long as resource and location requirements are fulfilled, the modules can be freely chosen and combined to arbitrary pipelines which execute a corresponding restriction. A trade-off has to be made between reconfiguration effort and query complexity. If only queries of low complexity may be expected, placing some general modules might be a good choice because their functionality can be changed without reconfiguration.

We chose 16 bit registers for indexing chunks. Hence, one table row entry can have a maximum size of $2^{16} \cdot 32$ bit = 256 KiB. Integer attributes can have a size of 1, 2, 3, or 4 byte and string attributes can contain up to 16 or 8192 single-byte characters and must be a multiple of 4 bytes.

B. Query Execution Time

To measure query execution times, we choose different queries and set up a test table which contains 12 four byte integers (id,a,b,c,d,e,f,g,h,i,j,k) and a 16 character string attribute (str) per entry. Thus, the size of one entry is 64 bytes which makes a total of 16 chunks to hold all attributes of one row. We fill the table with 2^{22} random entries which sum up to a total workload of 256 MiB. Our test system for the software is a Intel Core i7 Q820 @ 1,73 GHz with 8 GB RAM. We have a standard installation of MySQL 5.1 running on Windows 7 Professional 64 bit.

To ensure an equitable starting position, we created a INMEMORY table in the database system, i.e., the complete workload is located in main memory and not on hard-disk. The primary key column *id* is indexed, i.e., for specific search queries, the database system does not need to perform a full table scan, which results in a huge performance gain. Our hardware always does a full table scan regardless of the query.

We prepared sample queries and present the execution times of some queries next. Query 1 and 2 is a search on indexed attributes, query 3 and 4 includes a string comparison, and query 5 and 6 is the most complex restriction with

24 operations which can be executed on our current FPGA target.

```
Q1: SELECT * FROM test_table WHERE id=4711;
Q2: SELECT id FROM test_table WHERE id=4711;

Q3: SELECT * FROM test_table WHERE
str < 'abcdefghabcdefgh' OR g > 500;

Q4: SELECT id FROM test_table WHERE
str < 'abcdefghabcdefgh' OR g > 500;

Q5: SELECT * FROM test_table WHERE
    ((((a+b)-(c-d))<((e+f)+(g-a)))
                AND
    (((b+c)-(d-e))<((f+g)+(a-b))))
XOR
    ((((c+d)-e)<(f+g))
          OR
    ((a-b)>(c+d)));

Q6: SELECT id FROM test_table WHERE
    ((((a+b)-(c-d))<((e+f)+(g-a)))
                AND
    (((b+c)-(d-e))<((f+g)+(a-b))))
XOR
    ((((c+d)-e)<(f+g))
          OR
    ((a-b)>(c+d)));
```

Table VI shows the execution times in software and FPGA hardware. The overhead which is created by the spare chunks is also considered, i.e. the software can work on the clean 256 MiB workload, whereas the hardware has to work on the workload with overhead because spare chunks are needed (net load column). However, as soon as arithmetic operations are involved, query execution benefits from our FPGA solution. The throughput and execution time is independent from projections in our implementation but the software can benefit from them. The measured times represent the raw execution times excluding the time needed for analyzing a query or assembling the pipeline. Inside the FPGA, it is the time between the entry of the first tuple and the leaving of the last tuple out of the pipeline. In MySQL, it is the time elapsed during the *executing* thread state, which can be looked up in a query profile. However, this is only a little snapshot. Amongst others, the results are highly dependent of the table layout, i.e., wider table rows are beneficial for the FPGA because the influence of spare chunks decreases. On the other hand, simple search queries on indexed columns may be executed faster in software as long as the FPGA makes no use of such structures and does only full table scans.

Table VI
COMPARISON OF QUERY EXECUTION TIMES; 16 CHUNKS ARE NEEDED TO HOLD THE ATTRIBUTES

Query	Spare Ch.	Net Load	Software	Hardware	Speed-Up
Q1	1	285MB	0,0002s	0,713s	x0,0003
Q2	1	285MB	0,0002s	0,713s	x0,0003
Q3	2	302MB	3,49s	0,755s	x4,62
Q4	2	302MB	1,06s	0,755s	x1,40
Q5	4	335MB	5,16s	0,839s	x6,15
Q6	4	335MB	2,43s	0,839s	x2,90

VII. CONCLUSIONS & FUTURE WORK

In this paper, we presented a flexible approach for on-the-fly query processing based on a dynamically reconfigurable data path assembled at run-time of modules of a partially reconfigurable module library. It was shown how partial reconfiguration can be used to construct flexible pipelines to execute real SQL queries consisting of projections and restrictions. We eliminated the need of running a full synthesis for each query, which is done by *Glacier*. We are also not bound to a fixed pipeline as Netezza's *FAST*-engines. We experienced that query execution can benefit a lot from an FPGA solution as soon as restrictions on non-indexed columns are involved. However, we cannot provide detailed reconfiguration times yet, because we are using JTAG for partial reconfiguration. This shall be changed in the future by using the ICAP interface. In future work, we will move to newer technologies such as Virtex 6 which offers the possibility to implement wider buses or more complex operations like aggregations. The library shall be extended also to support more data types like floats or multi-byte strings, which are common in today's database environments. Furthermore, a multi-query system executing several queries in parallel shall also be considered. On a higher level of abstraction, developing methods to compute reconfiguration schedules and module selections for different optimization goals could be another direction of investigation as well as automatic query-to-data-path compilation.

ACKNOWLEDGMENT

We want to thank our industry project partners at IBM Deutschland Research & Development GmbH in Böblingen for supporting our research in this field.

REFERENCES

[1] R. Mueller, J. Teubner, and G. Alonso, "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

[2] P. Francisco, "The Netezza Data Appliance Architecture: A Platform for High Performance Data Warehousing and Analytics," IBM, Tech. Rep., 2011.

[3] D. Koch, C. Beckhoff, and J. Teich, "ReCoBus-Builder - A novel tool and technique to build statically and dynamically reconfigurable systems for FPGAS," in *International Conference on Field Programmable Logic and Applications, 2008. FPL 2008.*, sept. 2008, pp. 119 –124.

[4] "Information Technology Database Languages SQL Part 1: Framework (SQL/Framework)," ANSI/ISO/IEC 9075-1:2008.

[5] "Transaction Processing Performance Council - Ad-hoc Decision Support Benchmark," http://www.tpc.org/tpch/default.asp, [Online; accessed 02-January-2012].

[6] P. Flajolet, J. Raoult, and J. Vuillemin, "The number of registers required for evaluating arithmetic expressions," *Theoretical Computer Science*, vol. 9, pp. 99 – 125, 1979.

[7] B. Bridgford, C. Carmichael, and C. W. Tseng, "XAPP 779: Correcting Single-Event Upsets in Virtex-II Platform FPGA Configuration Memory," Xilinx, Tech. Rep., 2007.