RePro Vide: Towards Utilizing Heterogeneous Partially Reconfigurable Architectures for Near-Memory Data Processing

Andreas Becher [1], Achim Herrmann [2], Stefan Wildermann [3], Jürgen Teich [4]

Abstract: Reconfigurable hardware such as Field-programmable Gate Arrays (FPGAs) is widely used for data processing in databases. Most of the related work focuses on accelerating one or a small set of specific operations like sort, join, regular expression matching. A drawback of such approaches is often the assumed static accelerator hardware architecture: Rather than adapting the hardware to fit the query, the query plan has to be adapted to fit the hardware. Moreover, operators or data types that are not supported by the accelerator have to be processed in software. As a remedy, approaches for exploiting the dynamic partial reconfigurability of FPGAs have been proposed that are able to adapt the datapath at runtime. However, on modern FPGAs, this introduces new challenges due to the heterogeneity of the available resources. In addition, not only the execution resources may be heterogeneous but also the memory resources. This work focuses on the architectural aspects of database (co-)processing on heterogeneous FPGA-based PSoC (programmable System-on-Chip) architectures including processors, specialized hardware components, multiple memory types and dynamically partially reconfigurable areas. We present an approach to support such (co-)processing called RePro Vide. In particular, we introduce a model to formalize the challenging task of operator placement and buffer allocation onto such heterogeneous hardware and describe the difficulties of finding good placements. Furthermore, a detailed insight into different memory types and their peculiarities is given in order to use the strength of heterogeneous memory architectures. Here, we also highlight the implications of heterogeneous memories for the problem of query placement.

Keywords: FPGA; Shared Memory; Query Acceleration; Near-Memory Processing

Acknowledgement: This project is funded by the DFG priority program SPP2037.

1 Introduction

The usage of FPGAs for database operator acceleration has been of interest in the research community for many years [MT09]. Their unique ability to configure logic resources like flip-flops, look-up tables (LUTs), block RAMs (BRAMs), and digital signal processors (DSPs) to implement complex hardware modules inspired a great body of literature on

[1] Friedrich-Alexander Universität Erlangen-Nürnberg (FAU), Erlangen, Germany andreas.becher@fau.de
[2] Friedrich-Alexander Universität Erlangen-Nürnberg (FAU), Erlangen, Germany achim.herrmann@fau.de
[3] Friedrich-Alexander Universität Erlangen-Nürnberg (FAU), Erlangen, Germany stefan.wildermann@fau.de
[4] Friedrich-Alexander Universität Erlangen-Nürnberg (FAU), Erlangen, Germany juergen.teich@fau.de

doi:10.18420/btw2019-ws-04

accelerated database operators and systems. The ever increasing amount of data produced every day also increased the research efforts to utilize the energy efficiency of FPGA-based implementations. Research projects as well as industrial use cases, like Microsoft's Catapult [Pu14] and Baidu's FPGA-based data analysis [Ou16], have demonstrated various benefits of using FPGA-based co-processors for data processing (see also [Be18]):

- I/O rate processing of pipelined, non-blocking operators.
- Energy efficiency and reduction of power consumption and/or . . .
- . . . speedup through parallel and specialized hardware implementations of OLAP operators.
- Resource efficiency by taking workload from processors and providing query-specific hardware acceleration.

However, the majority of related work focuses on accelerating one or a small set of specific operations like sort, join or regular expression matching. A drawback of such approaches is the assumed static hardware-accelerator architecture: It is not possible to adapt the hardware to the query, rather the query plan has to be adapted to fit the hardware. Moreover, operators or data types that are not supported by the accelerator can't be accelerated. Here, dynamic hardware reconfiguration could provide a viable means to instead adapt the hardware to the query by reconfiguration of acceleration modules at runtime. Consequently, it would be possible to provide query-specific acceleration which is optimized for the respective use case while, at the same time, provide a more efficient utilization of the limited FPGA resources. Yet, the challenge of dynamic reconfiguration is to efficiently adapt the FPGA configuration to a query and to time-multiplex query processing under scarce resources [Zi16]. In particular, ensuring that the reconfiguration pays off is a major challenge, as the FPGA reconfiguration itself can take from a few milliseconds (partial reconfiguration) to seconds (complete reconfiguration) [Be16a].

In this paper, we present a query processing platform based on heterogeneous memory and processing resources. We formalize the problem of placing queries onto such highly heterogeneous platforms and discuss how such placements may be optimized. To exemplify the impact a non-optimal usage of the heterogeneous memory may have, a characterization of the available memory is conducted and an example presented.

2 Related Work

FPGA acceleration of single data processing operations has been excessively investigated in recent years. The important *join* operator has been implemented on FPGAs successful and evaluated in various works. Here, FPGA implementations for various algorithms have been evaluated, e. g., hash join [Ha13], sort-merge join [CO14], as well as hardware-software

co-designs [Be15] and reconfigurable implementations [UIO15]. In addition, as sorting is part of many database operators (e. g. sort-merge join) different algorithms have been implemented on FPGAs over the years, see e. g. [CO14; MTA12; Su13]. Also, FPGA implementations for complex operations exist. E. g., István et al. [ISA16] and Becher et al. [BWT18] propose run-time parametrizable operators for *regular-expression matching* on FPGAs.

Whereas the above cases investigate the acceleration of single operators only, the following approaches consider also the FPGA acceleration of multiple operators within a query such as filter, sort, aggregate, join, and group by as implemented on a FPGA by Baidu [Ou16]. A hardware/software co-design including an FPGA accelerator that consists of a feed-forward pipeline of hardware kernels for selection, projection, and sorting is proposed by Sukhwani et al. [Su15]. Sidler et al. [Si17] extended a CPU-based system by an FPGA implementing complex operators. The FPGA can offer acceleration for the Skyline operator, stochastic gradient descent, and regular expression matching.

However, even if all the approaches above are combined, only a very limited number of queries or datasets still can be automatically processed. The *major* drawback of these approaches is that they can only accelerate queries that contain the specific operations provided by the FPGA design. As soon as a query does not contain these operations or requires the operation on different data types (e.g., FLOAT or DATE instead of INTEGER), no hardware acceleration is supported, leading to a poor utilization of available resources. Particularly, implementing every possible operator (with every possible data type) would lead to huge FPGA designs and unnecessary overheads and is therefore not applicable. Making use of the run-time reconfigurability of FPGAs allows to circumvent the need for huge FPGAs while supporting a wide range of operators. This provides query-specific acceleration optimized for the respective query by loading only the operators needed. Saved FPGA resources can therefore be used to improve the specific operators instead of implementing unused logic circuits. Wang et al. [Wa16] show that even for a single query, it makes sense to provide multiple accelerators and reconfigure between these accelerators while processing a query. They propose a methodology for automatically generating multiple execution plans for a given query. At runtime, the execution plan is chosen that has the lowest execution time according to a cost model. While this work validates the benefit of adapting the hardware to the query by means of reconfiguration, it has the major drawback that each accelerator is tailored to a specific query. Synthesis of an accelerator can take hours. Thus, it is considered infeasible to assemble accelerators for dynamically arriving queries at run-time. In addition, the approach is based on full reconfiguration which implies reconfiguration overheads in the range of seconds.

Contrary, Ziener et al. [Zi16] present a methodology based on partial reconfiguration for *on-the-fly data-path generation* of a query-stream pipeline. Query streams can be accelerated by composing and placing pre-synthesized modules (e. g., aggregation and restriction operators) on the FPGA by means of partial reconfiguration. Becher et al. [Be18] propose a reconfigurable architecture for near-data processing called *Reconfigurable Data Provider*

(ReProVide). ReProVide proposes a layout of reconfigurable hardware resources based on multiple partially reconfigurable areas, into which query-/operator-specific accelerators can be dynamically loaded. Also, this approach makes use of a library of pre-synthesized reconfigurable hardware accelerators. The major difference between [Zi16] and [Be18] is that the former makes use of slot-style partial reconfiguration (which is not supported by standard FPGA tools) whereas the latter exploits island-style partial reconfiguration (which is supported by standard FPGA tools), see [KBT09]. In addition, the latter approach also exploits the coupling and co-processing of queries on FPGA-based PSoCs using the available on-chip processor system. These works introduce the design of reconfigurable accelerator modules for different operations and the design of the partially reconfigurable architecture. In addition, [Zi16] introduces cost models for assessing the performance of query processing based on these accelerator modules. Both works, however, do not provide solutions of the problem of automatically mapping a query onto a respective architecture.

3 Heterogeneous Partially Reconfigurable Architectures for Near-Memory Processing

We consider heterogeneous partially reconfigurable architectures for near-memory processing based on the following concepts. The data is stored in non-volatile memory such as SSDs or in volatile memory like DDR-SDRAM directly attached to the platform. It, thus, does not restrict whatsoever the datastructures used on the storage media and the low-level data management. Data access and modification is therefore only permitted by commands to the platform. A heterogeneous processing architecture consisting of CPUs and specialized hardware is assumed in order to combine the flexibility and ease of software implementations with the energy efficiency and performance of specialized hardware implementations. The *programmable logic* of the FPGA-based SoC platform is divided into a *static* part, which contains all components like hardware controllers and interfaces that stay fixed, and one or multiple *partially reconfigurable regions*, into which reconfigurable hardware accelerators can be loaded. The proposed architecture of a ReProVide platform is an example of such a heterogeneous partially reconfigurable architecture for near-memory processing. The target is to pull the (sparsely stored) data of interest out of one or more high-volume data sources. Only transmitting this information-rich subset of data to the host system has the potential of significantly reducing the dominant factor of power consumption in data-center networks: data transport. This means the reduction of data without an increase of execution time is of utmost importance. The platform, therefore, has to utilize the individual strengths of heterogeneous resources in order to accomplish the needed reduction.

3.1 ReProVide Architecture

The design of a ReProVide platform is based on top of existing FPGA technology like Xilinx Zynq All Programmable SoCs that contain programmable (FPGA) logic, multi-core CPUs,

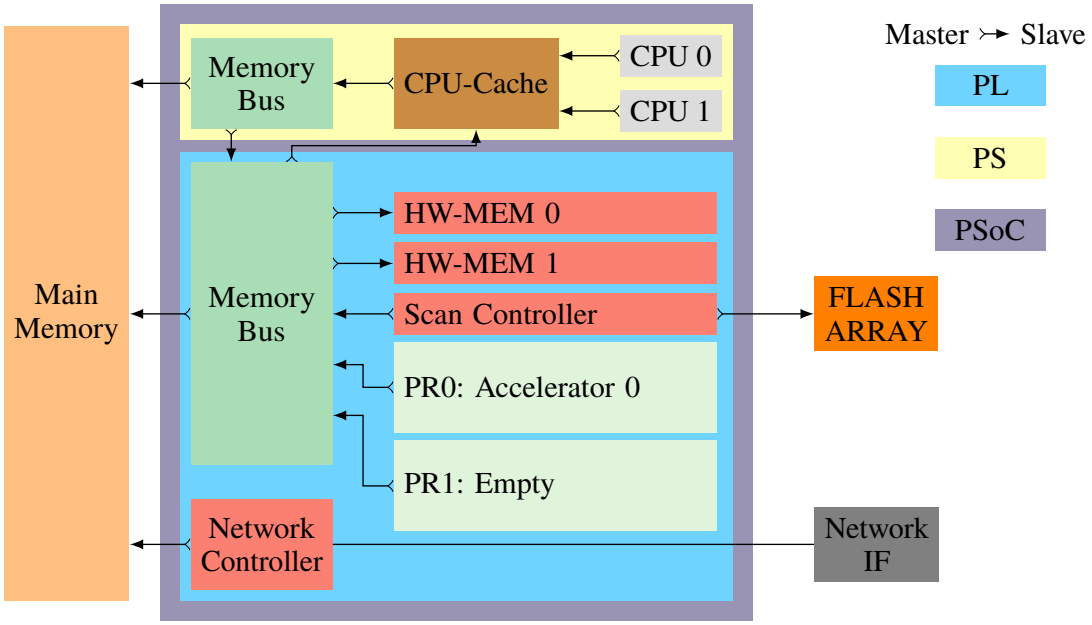


Fig. 1: Architecture of a ReProVide platform. Next to statically implemented IPs such as the *Network Controller*, *Scan Controller*, and heterogeneous (on- and off-chip volatile and non-volatile) memory resources, also partially reconfigurable areas are shown (*PR0, PR1*). These areas are reconfigured during runtime, loading presynthesized hardware modules. Additionally, a Processing System (PS) containing a dual core ARM-based processor is contained on the SoC.

and peripherals. This FPGA-based SoC makes use of hardware reconfiguration to adapt datapaths and accelerators for being able to process different online analytical processing (OLAP) and data mining operators on data from its attached heterogeneous volatile and non-volatile data sources.

Fig. 1 depicts the architecture of a proposed ReProVide platform containing a tightly-coupled processor system (PS) and programmable logic (PL). While the table management is executed on one core of the processing system, handling of the data is mostly dealt with within the programmable logic. To accelerate typical OLAP query operations, multiple partially reconfigurable regions (PRs) within the programmable logic allow offloading of operators. Requests are received via a high-speed network interface and forwarded to the management software for further processing. We will go into details in Section 3.2. To relief the processing system from the task of result transmission, a specialized *Network Controller* implemented in the static system allows sending the resulting data to the requesting host with minimum intervention from the management software. It utilizes a circuit for data reordering which we will introduce in Section 3.1.1.

A host may request data using a specified schema (row-store or column-store layout) while the management might use another schema to store the data on the attached memories. In addition, hardware accelerators are either specialized for a specific schema or are able to cope with different schemes. The first option would, however, increase the development time of the accelerator library and reduce the chances an already configured accelerator

can be reused. For the second option, to evade using additional logic in the accelerator itself or executing software to detangle the various schemes, we introduce a dedicated and parameterizable hardware component called *Scan Controller* to do the job of data loading and schema-on-fly transformations. Therefore, this controller also has direct access to the attached FLASH storage.

3.1.1 Scan Controller

The main task of the *Scan Controller* is to translate the schema which is used to store the data in the available memories and strip not needed attributes. A programmable data reordering engine called ReOrder unit [Be16b] together with scatter-gather DMA-engines are utilized to achieve this task. When combined, the *Scan Controller* can be programmed to gather data from multiple locations (particularly the case in column-oriented table store) to provide compacted tuples for further processing. In the case of row-oriented stored tables, not needed attributes are striped off to relief the interconnect from transferring unnecessary information and various memories from reserving space for this.

This functionality allows to define a single and common schema on which all operators work and may therefore disconnect the operator implementation (e.g., hardware accelerator) from the storage format.

Additional assistance for hardware accelerators is given by the insertion of so-called *placeholders* (intermediate/temporary attributes) into the tuple stream. These *placeholders* can be used to store intermediate results within the tuple, e.g., from arithmetic operations like *additions*, without the need to dynamically change the size of the tuple or the overhead of an additional result stream.

Fig. 2 depicts a simple query from (a) a row-based table and (b) a column-based table.

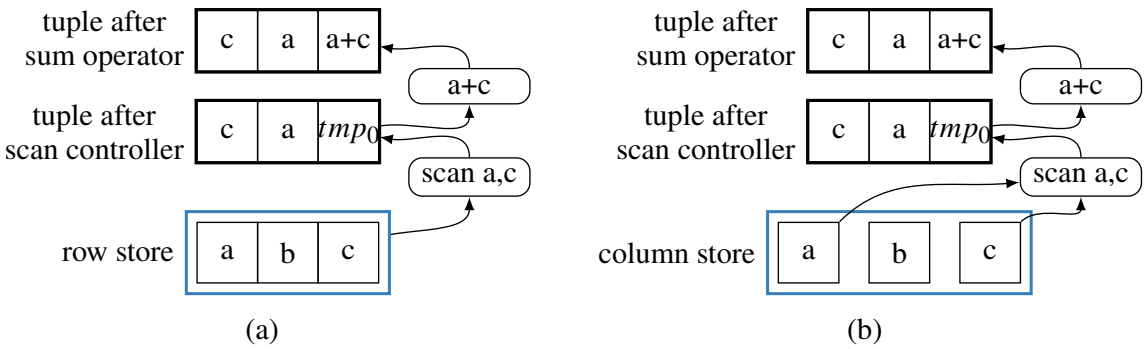


Fig. 2: A tuple (a, b, c) being processed according to the query `SELECT c,a,(a+c);` using a row-oriented storage (a) and a column-oriented storage (b). First, the scan controller produces a new tuple with placeholders and the requested order of attributes. In the next step, the *sum* function is applied to this tuple and the result is written to the previously created placeholder field (tmp_0).

Note that the *sum* operator sees the same input tuple in both cases and can therefore use the same accelerator. As can be seen, the *Scan Controller* can perform simple table transformations transparently to abstract from the storage layout of the tables. Not only does this simplify the operator implementations and increase the overall execution speed as less partial reconfigurations are necessary, it enables optimizations going beyond classical row-/column-oriented storages, e. g. partitioning a table onto different memory types or by related attributes.

A subset of this transformation capabilities is also implemented in the *Network Controller* to remove no longer needed attributes and *placeholders* before transmitting results to the requesting host.

3.2 Query Management

Once a query is received from a requesting host, a local query management is required to actually execute the query on the ReProVide platform. The query manager will be executed on the CPU. It involves three basic steps as illustrated in Figure 3.

The first step is *query placement* of the *query execution plan* (QEP) onto the available resources on the ReProVide platform. This is a *hardware/software partitioning* problem according to [Te12], and consists of the following three steps: (a) *allocation* of resources, (b) *binding* of tasks onto resources, and (c) *scheduling* of tasks on shared resources. For query placement on reconfigurable architectures, this hardware/software partitioning problem has to be adapted as detailed in Section 4.

The task of *query compilation* is then to configure the platform and particularly the selected accelerators with the runtime parameters according to the operator parameters in the query execution plan. This step is operator-specific and beyond the scope of this paper. It is e. g. illustrated in [BWT18] for accelerators for regular expression matching.

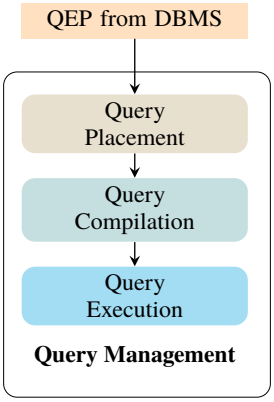


Fig. 3: Overview of runtime management. The lines indicate the flow of how a QEP is processed when obtaining it from a requesting host.

Finally, *query execution* is responsible for orchestrating the query processing according to the schedule generated in the first step. This particularly involves scheduling the reconfiguration of accelerators on shared partial regions, the memory allocation, as well as exception handling which particularly occurs for under- or overflowing buffers.

4 Query Placement Problem

A query execution plan (QEP) describes the order in which operations have to be applied on the data to execute a given query. In order to utilize the full potential, a given query execution plan has to be mapped onto the available resources of a ReProVide platform. Such a mapping can be optimized for query execution time, resource consumption, and energy efficiency or a combination of these objectives. In this subsection, we introduce the formal model of the placement problem for QEPs onto a ReProvide platform.

4.1 Generating the Query-Specific Architecture Configuration

One part of the query placement problem is to decide how to configure the reconfigurable areas to process a given query. Here, the *QEP* is represented as a directed graph $G_Q = (O, D)$ containing the set of operators O (vertices) and their dependencies on other operators $D \subseteq O \times O$ (directed edges), see Fig. 4(a), left for an example.

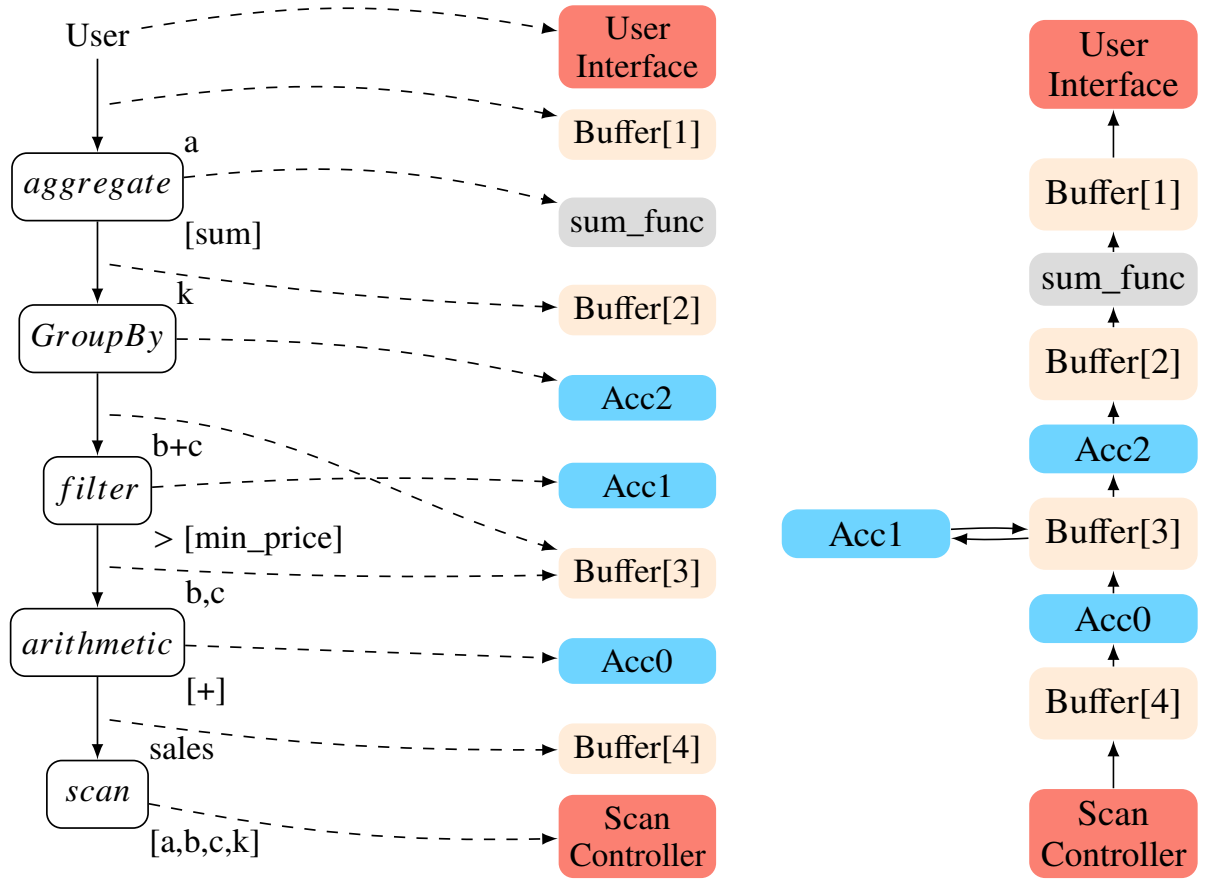
Problem graph. We assume a set H of pre-synthesized hardware modules as well as a set S of software functions for accelerating operators on a ReProVide platform is given. The set of available acceleration functions being available in an *accelerator library* are denoted by $L = H \cup S$. Each function $l \in L$ covers the acceleration of one or multiple operators. Now, for generating the *query-specific configuration*, first a set of accelerators A has to be allocated (instantiated) from this library. Then, each operator node $o \in O$ of the query plan has to be mapped onto an accelerator that supports the respective operator. This operator mapping is given by

$$\beta_O : O \to A. \tag{1}$$

Furthermore, a set of buffers B has to be allocated for storing the data that is exchanged between the accelerators. Each data dependency $d \in D$ of the QEP has to be assigned to one allocated buffer:

$$\beta_D : D \to B. \tag{2}$$

Fig. 4(a) illustrates a mapping of operators onto accelerators and of data dependencies onto buffers. Based on the set $T = A \cup B$ of allocated accelerators and buffers as well as



(a) Example QEP and a query-specific allocation of accelerators and buffers. Operators are mapped onto accelerators, and data dependencies are mapped onto buffers according to the dashed edges. The width of each buffer (i. e. tuple size) is given in brackets and omitted if *zero*.

(b) Problem graph generated from the QEP resource allocation (a). Again, the width of each buffer is given in brackets and omitted if *zero*.

Fig. 4: Problem graph generation from QEP.

the mapping of operators and data dependencies, the *query-specific configuration* can be represented as a query-specific *problem graph* $G_P = (T, F)$, where edges F connect the buffers with the accelerators accessing them. Buffers b connected to an accelerator a by an edge $(b, a) \in F$ or $(a, b) \in F$ are called *input buffers* or *output buffers* of a, respectively. Fig. 4(b) represents the problem graph of the example. Note that the flow of data as shown in a problem graph is opposite to the direction of edges of the QEP.

Buffer allocation. The proper *dimensioning of buffer capacities* is a major optimization control knob. As we will discuss below, it may have a huge influence on achievable performance numbers like throughput. However, on the other hand, it has to adhere to several constraints. The Scan Controller (described in Section 3.1.1) supports to load attributes from various data sources with different layouts. Its purpose is to load all attributes required by the subsequent operators. Moreover, the output buffer of each accelerator has to provide

space for storing all those attributes that are still required by subsequently executed operators. This means that the accelerator (a) copies the respective attributes from its input buffer(s) to its output buffer together with (b) writing the results calculated by the accelerator itself.

Fig. 4(a) illustrates this for an example. The $scan$ operator has to load attributes $[a, b, c, k]$. Operator $arithmetic$ has to write attributes $[a, b + c, k]$. The filter has to write attributes $[a, k]$, and so on.

Consequently, the *width* of a buffer is fixed for one problem graph and given by the size of one tuple of the respective attributes. However, the *depth* of a buffer $b \in B$ (i.e., the amount of tuples that can be stored in the buffer) is an optimization control knob, which we denote by $n_b \in \mathbb{N}_{\geq 0}$.

Let $width(b)$ be the statistically[5] derivable size of one tuple in buffer b.

Then, the $capacity$ of each allocated buffer $b \in B$ is denoted by

$$capacity(b, n_b) = width(b) \cdot n_b. \tag{3}$$

In Section 5, we empirically study the influence of this optimization variable on throughput. Several operators support stream processing of tuples (e.g., most arithmetics and filters), and thus support arbitrary $n_b \geq 1$. However, so-called *blocking* operators (e.g., *sort*, *unique* and *join*) depend on several or even all tuples from the preceding operators. We therefore include $n_{min} : O \rightarrow \mathbb{N}_{\geq 0}$ which gives the minimum amount of tuples the operator needs to access for calculation. Note that this may rise to the size of a whole table if operators such as *joins* need random access on the intermediate result. Therefore, the minimum buffer depth depends on the operators executed on the accelerators accessing the buffer, and is obtained as:

$$\forall o \in O, b \in B : (b, \beta_O(o)) \in F : n_b \geq n_{min}(o) \tag{4}$$

The problem graph generation may already apply domain knowledge to create optimized buffer allocations. One such optimization is the instantiation of multiple buffers to support streaming execution of QEP operations. Another is to reduce the over-provisioning of buffers: Without accurate statistics, some buffer might be highly over-dimensioned if designed to hold the whole table to guarantee no information is lost.

4.2 Determining a Configuration of the Reconfigurable Architecture

From the problem graph specification, we still have to determine a *query-specific configuration* (QSC) of the platform. The actual reconfigurable architecture (as presented in Section 3)

[5] The size can be determined accurately if a tuple consists of fixed length columns only. If variable length columns are present, a meaningful average is statistically obtained. Query execution has to handle occurring exceptions when buffers are too small to hold a single tuple due to the length of the variable length fields.

provides static resources that have to be programmed (or re-configured) according to such a configuration. Notably, this is also a mapping problem which we formalize in the following.

Architecture Graph. In order to embed a problem graph generated from a QEP onto a concrete ReProVide platform, also a graph representation can be used. A heterogeneous architecture is described by a directed architecture graph $G_A = (R, L)$. This architecture graph models the available resources, i. e., CPU cores, memories, co-processors, and partially reconfigurable areas, as well as the connections between these resources. Fig. 5 (right) visualizes an architecture graph of the ReProVide platform shown in Fig. 1. The vertices represent these resources $R = R_P \cup R_M \cup R_C$, being processing R_P, memory R_M, and communication resources R_C. Memory resources are characterized by a maximum storage capacity $capacity(m), \forall m \in R_M$.

Directed links $l \in L$ connect resources which can communicate with each other: $L \subset R \times R$. Note that the direction of an edge describes which node is the *initiator* (or *Master*) of a data transfer (outgoing edges) and therefore does not describe the direction of the flow of data itself. This is consistent with the *Master Slave principle* that is also illustrated by Fig. 1, and allows to express separated memory domains (e.g., the network controller can only access the main memory in Fig. 5). Let $C_{G_A}(v)$ with $v \in R$ define the set of all resources connected to v in G_A. A pair of resources $(a, b) : a, b \in R$ is connected if a directed path from a to b or b to a exists. For simplicity, we assume all links to allow for bidirectional (duplex) data transfers.

Mapping the Problem Graph. The problem of determining a query-specific configuration (QSC) on the reconfigurable target platform can thus be formalized as mapping the problem graph $G_P = (A \cup B, F)$ onto the architecture graph. This involves *binding* each accelerator $a \in A$ of the problem graph to one processing resource $r \in R_P$ of the architecture graph, given by

$$\beta_A : A \rightarrow R_P. \tag{5}$$

Here, obviously, hardware accelerators can only be bound to partially reconfigurable regions, and software accelerators to software processing resources. Furthermore, each buffer $b \in B$ has to be bound onto one memory resource $r \in R_M$:

$$\beta_B : B \rightarrow R_M. \tag{6}$$

The bindings have to adhere to the following mapping constraints.

Routing contraints: Each accelerator a can access the memories containing its input and output buffers from the processing resource $\beta_A(a)$ on which it is executed, or formally with

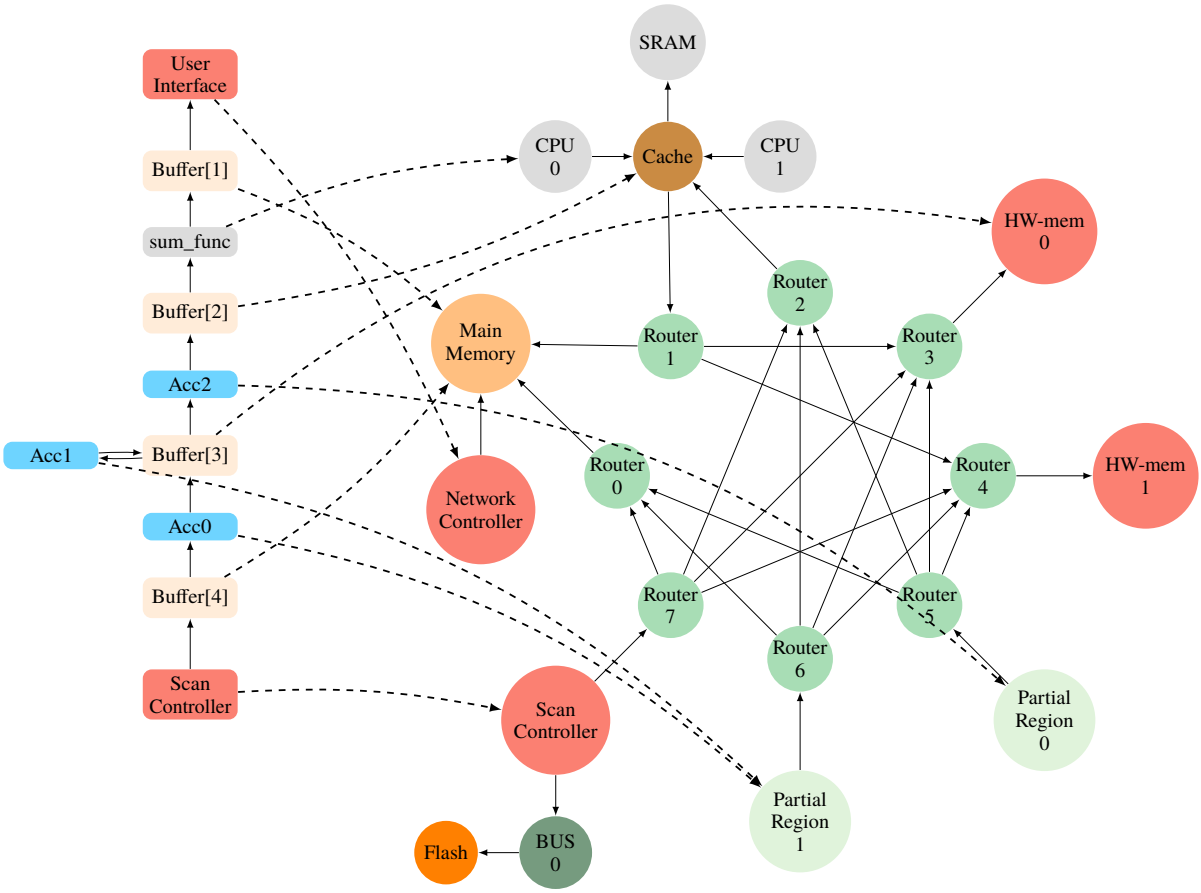


Fig. 5: Problem graph (left) and its binding to the architecture graph of a heterogeneous architecture such as the ReProVide platform depicted in Fig. 1. The communication resource *BUS0* models the half duplex connection between the *Flash*-Memory and the *Scan Controller*.

C_{G_A} representing the set of reachable resources:

$$\forall a \in A, b \in B, (b,a), (a,b) \in F : \beta_B(b) \in C_{G_A}(\beta_A(a)) \tag{7}$$

Memory contraints: We assume that the buffers are statically allocated and then used throughout the execution of the query. As discussed before, the size of buffers depends on decision variables which are subject to the constraint in Eq. (4). When binding buffers to memory resources, it has to be ensured that the memory resources provide sufficient capacities to hold the assigned buffers, i.e.,

$$\forall r \in R_M : \sum_{\substack{b \in B: \\ \beta_B(b)=r}} capacity(b, n_b) \leq capacity(r) \tag{8}$$

Fig. 5 illustrates a valid binding of the exemplary G_P onto the architecture graph G_A of a ReProVide. As illustrated for $Acc0$ and $Acc1$, accelerators may be bound to the

same processing resource, in this case partially reconfigurable region PR1. In such cases, it is necessary to provide *scheduling strategies* to resolve possible conflicts. In case of data-dependent accelerators as in this example ($Acc1$ requires the results from $Acc0$), the scheduling priority is given according to the dependencies. In case of accelerators without dependencies, i.e., accelerators of two different QEP branches before a *join* operation, all accelerators on the branch that is prioritized by the join operator will be scheduled before the accelerators on the other branch.

While we used a single query as a driving example, please note that multiple concurrent queries can be placed. The same constraints apply for the allocation and binding. The scheduler, however, might be adapted to apply prioritization if resources are shared.

4.3 Optimal Query Placement

So far, the steps and constraints for determining a *feasible* query-specific configuration (QSC) were discussed. However, in general, an *optimal placement* is desired which optimizes performance numbers like throughput or latency. Multiple factors influence the performance numbers such as the set of allocated accelerators, the mapping of operators onto these accelerators, and the mapping of accelerators onto the available resources. The big advantage of hardware accelerators is their determinism. Particularly, the performance of streaming operations can be predicted. So, quite accurate cost models exist for such accelerators. Particularly, Ziener et al. [Zi16] propose cost models that allow to estimate latency and throughput of a pipeline of such accelerators. The accelerators in the library of ReProVide are pre-synthesized. This means that optimizing a single accelerator for latency and throughput are offline tasks and thus not part of the query placement problem.

However, reconfiguration between accelerators during processing of a query introduces additional offsets which are not considered in [Zi16]. ReProVide make use of island-style reconfiguration, which means that, when a module is loaded into a reconfigurable area, the complete area is reconfigured. Therefore, the reconfiguration time depends not on the accelerator but on the size of the reconfigurable area. For reasonably sized areas, reconfiguration time can stay within few milliseconds, see [Be16a]. However, determining the size of the reconfigurable areas is an offline task and thus not part of the query placement problem.

For ReProVide platforms, we assume a self-triggered execution, i. e. each accelerator starts working when it receives a ready signal from its succeeding accelerators and the directly preceding accelerator signals that it has filled the input buffer. When an accelerator starts execution, it consumes the tuples from its input buffer and produces the output in its output buffer. Subsequently, it sends a ready signal to its preceding accelerator and triggers the execution of the succeeding accelerator. With this execution scheme, the dimensioning of the allocated buffers and their mapping onto the available memory resources can significantly influence performance numbers as latency and throughput. Particularly, a buffer depth of

$n_b = 1$ does not necessarily lead to the best pipelined schedule, even when all accelerators should have the same latency. In order to motivate the huge impact, we take a look into the characteristics of different memory types in the next subsection.

The optimal query placement depends on the allocation of accelerators and buffers, as well as their binding onto the architecture. The problem formalization of this paper is an extension of the system-level synthesis problem, which is proven to be NP-hard [BTT98][6]. Thus, finding an, at least, optimized placement is a complex task which can only be efficiently tackled by heuristics. Particularly, for query processing, an optimized placement should be available within a few milliseconds. While processing, additional queries may arrive and a new placement has to be found covering all running queries. This means the placement might change every time a new query arrives or completes, making it even more important to be adapted quickly. Whereas the main contribution of this paper is on understanding and formalizing this problem, the investigation of efficient heuristics is future work.

5 Heterogeneous memory system

ReProVide platforms allow to use three different types of memories: (I) local memory like FPGA BlockRAM or CPU caches, (II) DDR-SDRAM and (III) Flash memory. In order to optimize the mapping of the buffers of a problem graph to physical memory, one must know how these different memories behave in terms of throughput, latency, capacity and costs. This section classifies the memories according to these properties. The values used in this section are mostly taken from literature and therefore inaccurate, but this is still sufficient for the purpose of characterization. Additionally, the characteristic values for an example of a concrete ReProVide platform will be exemplified.

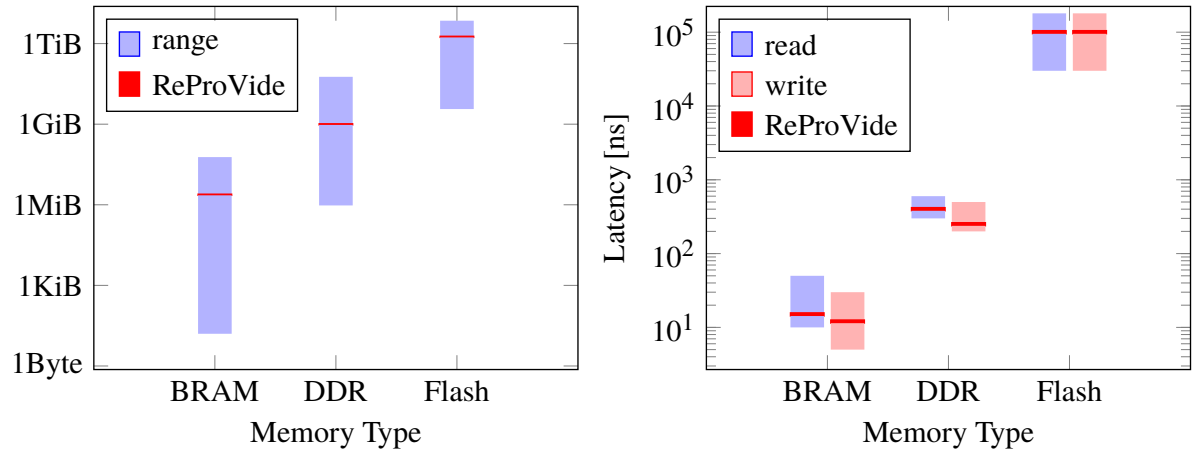
5.1 Memory Characterization

Capacity As ReProVide's purpose is the processing and filtering of big amounts of data, it is necessary to have enough capacity to store the intermediate data. The capacity depends on the particular hardware in use. Typical ranges for the different memories are depicted in Fig. 6(a). Local memory usually has capacities from a few bytes up to a few megabytes, DDR-SDRAM can have a few gigabytes and Flash memory goes up to many terabytes.

Cost Of course, it would be possible to have, for example, a terabyte of DDR-SDRAM, but compared to flash memory, this would be much more expensive in terms of both cost and energy. According to Xilinx Vivado, one Byte of BRAM has a power consumption in the scale of µW. DDR-SDRAM is more efficient with nW per Byte [LC03]. Flash memory

[6] This means that also the query placement problem is NP-hard, as the system-level synthesis problem can trivially be reduced onto it.

has the lowest power consumption per Byte: as it is non-volatile, just holding data requires no energy at all. When in use, its power consumption is in the magnitude of pW/B [LMP09]. Also, the price of 0.2 $/GB for flash memory is much lower than for DDR-SDRAM, which costs about 6 $/GB [Ha17].



(a) Capacity (b) Latency

Fig. 6: Typical capacity and latency of different memory types and the Xilinx zc706-based ReProVide prototype.

Latency But capacity often comes at the cost of latency. Here, latency is defined as the time it takes from request on until the first byte has been processed, i. e. read or written. It is desirable to keep the latency low as this will improve the data processing performance, especially for a small amount of data. Fig. 6(b) depicts typical latencies for different memory types in nanoseconds: The small BRAMs are usually placed tightly to the processing unit, leading to very low latency down to one cycle. The bigger DDR-SDRAMs require controllers to manage memory banks, increasing the latency to a few dozen of clock cycles. Flash memories often additionally use internal buffering and a serial interface. Both may increase the latency further into mid microseconds range [Gr09].

Throughput Throughput is the main characteristic when it comes to transferring big blocks of data. The throughput of a memory depends on different factors. The most obvious ones are bus-width and frequency. These factors impact the throughput linearly, means when doubling one of them, also the throughput doubles. They are both determined by the hardware implementation, so the query placement has no influence on them. A factor which can be influenced by query placement is the blocksize (i. e. the *capacity acc. to Eq.* (3)), and thus, how many data is transferred within one request. When transferring only small blocks, latency is the biggest component of the transfer time. With the blocksize high enough, the latency becomes negligible as it can be compensated through pipelining, and the overall throughput rises. Fig. 7 depicts the normalized optimum read and write throughput of the memory types, depending on the transferred blocksize. As it has the lowest latency, BRAM

generally has the highest throughput independent of the blocksize. Flash memory on the other side requires huge blocks to reach an acceptable throughput. Note that also the access pattern has a high impact on the throughput: sequential access is faster than random access.

For example, a ReProVide platform based on a Xilinx zc706 can access the DDR-SDRAM either over the Zynq's high performance ports (HP) or over the Accelerator Coherency Port (ACP). Sadri [Sa13] found out that the ACP has equal performance of up to 1,7 GB/s as the HP port, as long as the transfer size does not exceed the cache size of 512 KByte. Afterwards, the throughput drops to 600 MB/s due to invalidating the caches. The ACP performance is also effected by background applications running on the CPU, since they also use the caches. So are the high performance ports effected by each other, since they have to share the internal interconnect.

The examplary ReProVide platform uses a raid of SATA-3 SSDs as flash memory. The SATA-3 protocol has a theoretical throughput of 600 MB/s. However, the access pattern has huge impact on the actual throughput. Sequential-like throughput is only achieved with blocksizes greater than 32 KByte. Furthermore, there are effects like garbage collection and defragmentation running inside the SSD firmware to consider which may reduce the achievable throughput dramatically [Se18]. However, these effects are highly specific to the use case and the manufacturer.

In summary, there is a tradeoff between fast but small and expensive memories and slow but big and inexpensive memories. If one needs fast random access with small blocksizes, BRAMs offer the best performance if the amount of available storage is sufficient. DDR-SDRAM can be used if a higher capacity is needed or in case the blocksizes to access the memory are bigger. Lastly, flash memory is the best choice for storing huge amounts of data which is mostly read using sequential access or huge datablocks. Therefore, the buffer

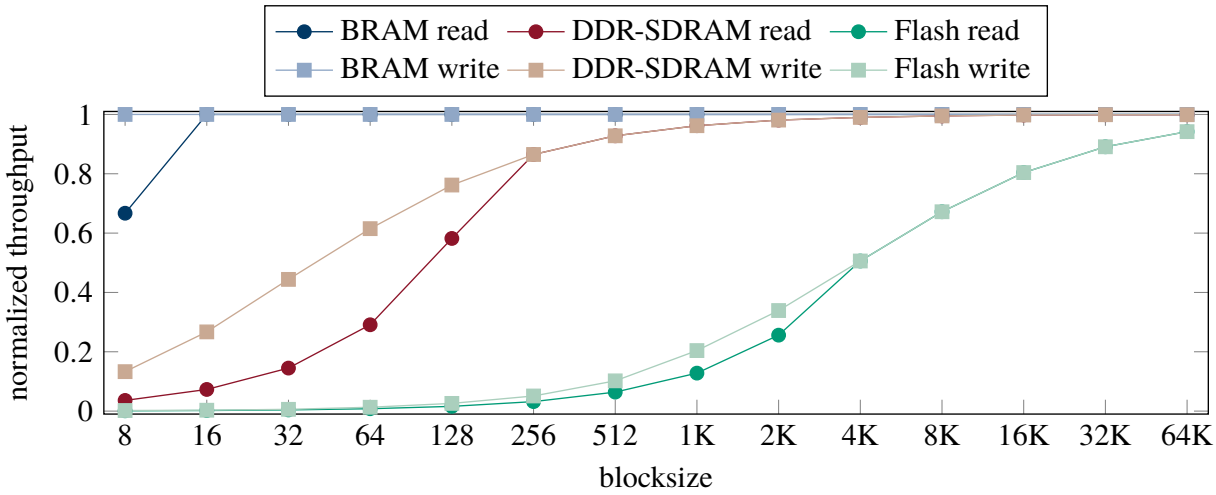
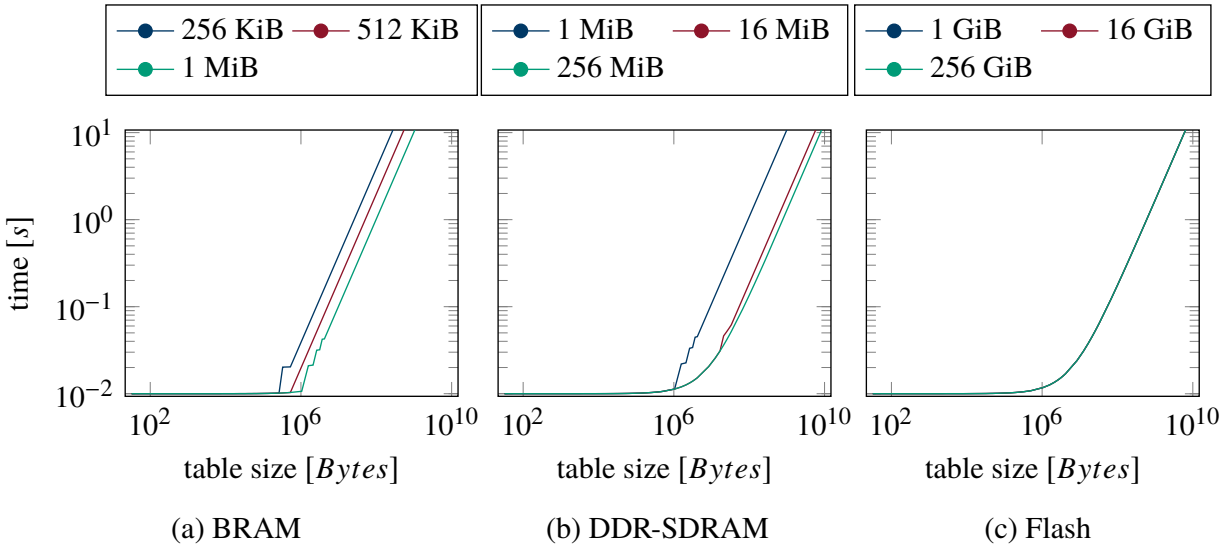


Fig. 7: Random access throughput of different memory types in relation to the size of each transfer (blocksize). The throughput is normalized to the maximum sequential throughput of each memory type.



(a) BRAM (b) DDR-SDRAM (c) Flash

Fig. 8: Execution time for two accelerator scheduled on the same partial reconfigurable area to process tables of different sizes. A buffer is allocated for exchanging data between both accelerators. Reconfiguration between accelerators is performed as soon as all tuples are produced in/consumed from the buffer. Per memory type, the execution time for different sizes allocated for this buffer is depicted.

allocation as part of query placement as presented in Section 4.1 has direct impact on the execution time of the query to process.

5.2 Exemplary Impact of Buffer Allocation and Binding on Query Execution Time

In order to give an example of the impact of different buffer allocations, we consider two accelerators mapped onto one partial region and exchanging tuples via a shared buffer, comparable to operators $Acc0$ and $Acc1$ in Fig. 5[7]. Each time accelerator $Acc0$ has filled its output buffer, a partial reconfiguration is triggered so that accelerator $Acc1$ consumes the data.

Fig. 8 depicts the execution time (y-axis) of these two accelerators processing tables of different sizes (x-axis). The execution time is depicted for all three memory types and for different sizes of the buffer used for exchanging data between the accelerators. We have chosen buffer sizes which are reasonable for the memory type with respect to its specific capacity ranges (see Fig. 6(a)). As one can see, the buffer size has a direct impact on the execution time. Increasing buffer size reduces the overall execution time as accelerators can process more tuples with less reconfiguration overhead.

Furthermore, the binding of buffers to the memory type has a direct impact on execution time. BRAM is the fastest if the table size is low (Fig. 8(a)). However, the throughput

[7] However, we ignore the scheduling of preceding and succeeding accelerators in the following example.

advantage is lost when table sizes grow: As the capacity of BRAM is limited, only small buffer sizes can be chosen (kilobytes to ~one megabyte). This means that for greater table sizes the reconfiguration has to be performed more frequently, and thus, reconfiguration overhead becomes dominant. Due to their higher capacities, the DDR-SDRAM and the Flash memories support bigger buffer sizes and are thus able to decrease the number of required reconfigurations. Particularly, Flash memories allow to choose buffer sizes in the range of gigabytes and thus to reduce the reconfiguration overhead even further. However, DDR-SDRAM has a higher throughput than Flash. Therefore, when choosing a sufficiently high buffer size (over 16 MiB in our example), the DDR-SDRAM is able to compensate the additional reconfigurations. This leads to DDR-SDRAM providing the lowest overall execution time (Fig. 8(b)), even compared to Flash (Fig. 8(c)). For a given table size, a simple heuristic can be derived to obtain a good buffer size.

Please note that, in this example, we assumed that the memories are exclusively accessed by the accelerators such that no congestion occurs. However, during operation of a ReProVide platform, concurrently executed accelerators of the same or even other parallel queries may access the interconnect and memories. This would reduce the actually achievable throughput. As such, also the congestion on interconnect and memories should be considered when solving the query placement optimization problem.

6 Conclusion and Future Work

While heterogeneity promises benefits in terms of reduced execution time, energy efficiency and costs for database processing, it is a burden too. Utilization of such heterogeneous architectures to gain benefits can be challenging. We presented a model to describe heterogeneous architectures such as a ReProVide platform and formalized the task of automatically generating a configuration from a given query execution plan while satisfying a number of physical constraints on computing, communication and memory resources. To allow for the optimization of this process, various factors have to be considered. Therefore, the available memory resources have been characterized and the effects of buffer allocation have been illustrated. With the presented model and characterization, we will investigate algorithms to find good solutions for the placement problem in the future.

References

[Be15] Becher, A.; Ziener, D.; Meyer-Wegener, K.; Teich, J.: A co-design approach for accelerated SQL query processing via FPGA-based data filtering. In: FPT. Pp. 192–195, Dec. 2015.

[Be16a] Becher, A.; Pirkl, J.; Herrmann, A.; Teich, J.; Wildermann, S.: Hybrid energy-aware reconfiguration management on Xilinx Zynq SoCs. In: ReConFig. Pp. 1–7, 2016.

[Be16b] Becher, A.; Wildermann, S.; Mühlenthaler, M.; Teich, J.: ReOrder: Runtime datapath generation for high-throughput multi-stream processing. In: ReConFig. Pp. 1–8, 2016.

[Be18] Becher, A.; B.G., L.; Broneske, D.; Drewes, T.; Gurumurthy, B.; Meyer-Wegener, K.; Pionteck, T.; Saake, G.; Teich, J.; Wildermann, S.: Integration of FPGAs in Database Management Systems: Challenges and Opportunities. Datenbank-Spektrum 18/3, pp. 145–156, Nov. 2018.

[BTT98] Blickle, T.; Teich, J.; Thiele, L.: System-Level Synthesis Using Evolutionary Algorithms. Design Automation for Embedded Systems 3/1, pp. 23–58, Jan. 1998, ISSN: 1572-8080.

[BWT18] Becher, A.; Wildermann, S.; Teich, J.: Optimistic regular expression matching on FPGAs for near-data processing. In: DaMoN. 4:1–4:3, June 2018.

[CO14] Casper, J.; Olukotun, K.: Hardware acceleration of database operations. In: FPGA. Pp. 151–160, 2014.

[Gr09] Grupp, L. M.; Caulfield, A. M.; Coburn, J.; Swanson, S.; Yaakobi, E.; Siegel, P. H.; Wolf, J. K.: Characterizing flash memory: Anomalies, observations, and applications. In: MICRO. Pp. 24–33, Dec. 2009.

[Ha13] Halstead, R. J.; Sukhwani, B.; Min, H.; Thoennes, M.; Dube, P.; Asaad, S. W.; Iyer, B.: Accelerating Join Operation for Relational Databases with FPGAs. In: FCCM. Pp. 17–20, 2013.

[Ha17] Havard: Historical Cost of Computer Memory and Storage, Accessed: November 20, 2018, Dec. 2017, URL: https://hblok.net/blog/storage/.

[ISA16] István, Z.; Sidler, D.; Alonso, G.: Runtime Parameterizable Regular Expression Operators for Databases. In: FCCM. Pp. 204–211, 2016.

[KBT09] Koch, D.; Beckhoff, C.; Teich, J.: Minimizing Internal Fragmentation by Fine-Grained Two-Dimensional Module Placement for Runtime Reconfiguralble Systems. In: FCCM. Pp. 251–254, Apr. 2009.

[LC03] Lee, H. G.; Chang, N.: Energy-aware Memory Allocation in Heterogeneous Non-volatile Memory Systems. In: ISLPED. Pp. 420–423, 2003, ISBN: 1-58113-682-X, URL: http://doi.acm.org/10.1145/871506.871609.

[LMP09] Lee, S.-W.; Moon, B.; Park, C.: Advances in Flash Memory SSD Technology for Enterprise Database Applications. In: SIGMOD. Pp. 863–870, 2009, ISBN: 978-1-60558-551-2, URL: http://doi.acm.org/10.1145/1559845.1559937.

[MT09] Müller, R.; Teubner, J.: FPGA: What's in it for a database? In: SIGMOD. Pp. 999–1004, 2009.

[MTA12] Müller, R.; Teubner, J.; Alonso, G.: Sorting networks on FPGAs. VLDB J. 21/1, pp. 1–23, 2012.

[Ou16] Ouyang, J.; Qi, W.; Wang, Y.; YichenTu; Wang, J.; Jia, B.: SDA: Software-Defined Accelerator for general-purpose big data analysis system. In: HCS. Pp. 1–23, Aug. 2016.

[Pu14] Putnam, A.; Caulfield, A.; Chung, E.; Chiou, D.; Constantinides, K.; Demme, J.; Esmaeilzadeh, H.; Fowers, J.; Gray, J.; Haselman, M.; Hauck, S.; Heil, S.; Hormati, A.; Kim, J.-Y.; Lanka, S.; Peterson, E.; Smith, A.; Thong, J.; Xiao, P. Y.; Burger, D.; Larus, J.; Gopal, G. P.; Pope, S.: A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In: ISCA. Pp. 13–24, 2014.

[Sa13] Sadri, M.; Weis, C.; Wehn, N.; Benini, L.: Energy and Performance Exploration of Accelerator Coherency Port Using Xilinx ZYNQ. In: FPGAworld. 5:1–5:8, 2013, ISBN: 978-1-4503-2496-0, URL: http://doi.acm.org/10.1145/2513683.2513688.

[Se18] Seagate Technology, L.: Lies, Damn Lies And SSD Benchmark Test Result, Accessed: November 20, 2018, 2018, URL: https://www.seagate.com/de/de/tech-insights/lies-damn-lies-and-ssd-benchmark-master-ti/.

[Si17] Sidler, D.; István, Z.; Owaida, M.; Alonso, G.: Accelerating Pattern Matching Queries in Hybrid CPU-FPGA Architectures. In: SIGMOD. Pp. 403–415, 2017.

[Su13] Sukhwani, B.; Thoennes, M.; Min, H.; Dube, P.; Brezzo, B.; Asaad, S. W.; Dillenberger, D.: Large Payload Streaming Database Sort and Projection on FPGAs. In: SBAC-PAD. Pp. 25–32, 2013.

[Su15] Sukhwani, B.; Thoennes, M.; Min, H.; Dube, P.; Brezzo, B.; Asaad, S. W.; Dillenberger, D.: A Hardware/Software Approach for Database Query Acceleration with FPGAs. International Journal of Parallel Programming 43/6, pp. 1129–1159, 2015.

[Te12] Teich, J.: Hardware/Software Codesign: The Past, the Present, and Predicting the Future. Proceedings of the IEEE 100/Special Centennial Issue, pp. 1411–1430, May 2012, ISSN: 0018-9219.

[UIO15] Ueda, T.; Ito, M.; Ohara, M.: A dynamically reconfigurable equi-joiner on FPGA, IBM Technical Report RT0969, 2015.

[Wa16] Wang, Z.; Paul, J.; Cheah, H. Y.; He, B.; Zhang, W.: Relational query processing on OpenCL-based FPGAs. In: FPL. Pp. 1–10, 2016.

[Zi16] Ziener, D.; Bauer, F.; Becher, A.; Dennl, C.; Meyer-Wegener, K.; Schürfeld, U.; Teich, J.; Vogt, J.; Weber, H.: FPGA-Based Dynamically Reconfigurable SQL Query Processing. TRETS 9/4, 25:1–25:24, 2016.