RESOURCE ELASTIC DYNAMIC STREAM PROCESSING ON FPGAS EXEMPLIFIED ON DATABASE ACCELERATION

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF SCIENCE AND ENGINEERING

2022

Kristiyan N Manev

Department of Computer Science

Contents

Word Count: 37521

List of Tables

List of Figures

Glossary

(re)configuration Changing partially or fully the loaded FPGA bitstream. 16, 116

chunk A part of a tuple that is transmitted in one clock cycle. 9, 51, 72, 104, 132

clock region height Defines region height. In UltraScale+, a clock region defines a fixed height of 60 CLBs, 24 DSPs or 12 BlockRAMs. 29, 60, 65, 116

DSPI ChannelID A DSPI bus encoding multiple virtual channels on a particular virtual stream . 10, 50, 51, 90, 110

DSPI ChunkID A DSPI bus encoding the currently transferred chunk. 50, 51, 93

DSPI instruction DSPI implements a credit-based system, where tokens are allocated in order to control the dataflow. DSPI implements six instructions that can be used to control token allocation and stream states. 48, 50, 70, 93, 95

DSPI StreamID A DSPI bus encoding a virtual stream . 10, 50, 51, 90, 93

external fragmentation Unutilized resource columns between placed modules. 37, 65, 102, 125, 140, 143

internal fragmentation Unutilized resources by a module when implemented for a specific resource footprint. 38, 101

module alternative Implemented module for different utility parameters (e.g., to merge-sort more input streams). 19, 24, 76, 99, 101, 102, 113, 119, 120, 125, 130, 139

module composing Stitching multiple modules together to implement a logically single operation (e.g., composed multiple filter modules to meet larger filter requirements). 19, 24, 59, 99, 102, 109, 110, 113, 139, 140

module resource footprint The specific resource requirements of the rectangular bounding box around implemented dynamic modules. 9, 62, 100, 116

module resource footprint variant Implemented module for different FPGA resource footprint to maximize placement options. 37, 66, 119, 120, 125, 143

module stitching Building an execution pipeline at runtime by placing module bitstreams. 9, 21, 40, 59, 120, 122

partial (re)configuration Changing the loaded FPGA bitstream for a partial region of the FPGA. 18, 27, 29, 47, 101, 116

tuple, record A data entry that contains multiple data fields. 9, 21, 34, 44, 70, 130

utility The amount of practical computation performed per pass of the data through the accelerator. 18, 19, 24, 25, 28, 51, 60, 62, 99, 139, 142

virtual stream/channel All virtual streams/channels share a single physical datapath to transfer actual data. The virtual streams/channels are differentiated by StreamID and ChannelID signals of DSPI. 9, 48, 50, 51, 59, 87, 95, 102

Abstract

While FPGAs are becoming mainstream in the deployment of datacenters and cloud systems, they are mostly used as updatable ASICs. This thesis shows that it is feasible to achieve acceleration for runtime-only known problems using dynamically built stream processing pipelines if we efficiently exploit the given FPGA resources and utilize additional techniques such as resource elasticity.

To achieve this, the requirements for stream processing accelerators are researched and an efficient dynamic stream processing protocol is proposed. Exemplifying our approach to database query processing due to its key importance in the Big Data era, a module library of the most used database operators is built. For this purpose, design factors for building scalable streaming accelerators are discussed and efficient accelerators for filter, sort, and join are proposed. Accelerators are fully decoupled from the external infrastructure through a partially reconfigurable generic DMA module that converts standard AXI interfaces into the proposed streaming protocol.

To improve the flexibility and throughput of the proposed system, resource elastic techniques are discussed and applied to key accelerators. This allows a runtime scheduler to fully tailor the execution pipeline to the runtime-only known problem and available FPGA resources. The performance and overheads in the prototyped system are evaluated and show that our approach is beneficial for FPGA acceleration.

Overall, with these contributions, this thesis shows stream processing acceleration achieved by execution pipelines that are adapting, through using resource elasticity, to the problem and available resources at runtime.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

Foremost, I would like to express my sincere gratitude to my supervisor, Dr Dirk Koch, for his guidance, support, and feedback throughout this project. Our meetings and conversations were key in deepening my knowledge and helping me progress with my research work.

I would like to thank Anuj Vaishnav for his valuable research support and our countless debates. I would like to thank Kaspar Mätas, Malte Vesper, and Khoa Pham for our fruitful collaborations. I would also like to thank everybody from our research group for the pleasant and mindful environment and our distracting daily lunch calls.

Finally, I would like to thank my parents, my family, and all my friends who have supported me through the past years.

Chapter 1

Introduction

Thesis statement: It is feasible to achieve acceleration for runtime-only known problems using dynamically built stream processing pipelines if we efficiently exploit the given FPGA resources and utilize additional techniques such as resource elasticity.

1.1 Motivation

The famous Moore's prediction about transistor scaling was targeting a relatively short period of time [72]. However, for many decades the technological improvement has led to the ever-increasing performance and complexity of computing systems with often little-to-none architectural investments. While technology nodes keep improving to this date, this process is expected to slow down as transistors keep approaching atom sizes (a silicon atom is about 0.2nm in diameter).

The exhaustion of the free lunch era results in an increasing amount of real-world performance originating from architectural and algorithmic improvements [39]. Traditional von Neumann architectures implemented in Central Processing Units (CPUs) provide flexible control logic, but often lack throughput in solving large data problems. Graphics Processing Units (GPUs) alleviate the problem by implementing a similar architecture, but in a massively parallel fashion. However, GPU instruction sets often implement simple operations, which results in large throughput for simple operations on large-scale data.

On the other hand, dataflow solutions attract more attention in recent years. Application-Specific Integrated Circuits (ASICs) can deliver the highest throughput and energy efficiency due to their optimized hardwired nature. However, modern technology ASICs require a large initial investment and lack post-fabrication flexibility, which drives

ASIC approaches only economically viable for large-scale deployment. Moreover, ASIC development is slow and cannot always keep pace with technological progress.

Field Programmable Gate Arrays (FPGAs) circumvent the sequential von Neuman bottleneck through a dataflow programming paradigm with a corresponding large throughput and high energy efficiency while keeping initial investments relatively low and allowing field-reprogrammability. In specialized large problems, both ASICs and FPGAs do tend to show significantly higher performance than CPUs and GPUs, and the small performance gap between ASICs and FPGAs is offset by using the most advanced process nodes for FPGAs and the possibility to update the hardware in the field [80].

The recent interest in FPGAs can also be observed in the recent acquisitions of Altera by Intel [69] and Xilinx by AMD [41]. Both FPGA companies have products co-designed with integrated ARM CPUs and reconfigurable fabric, but the aims of integrating programmable fabric into mainstream compute products are also visible in Intel's Xeon Phi co-processor integration [101] as well as in AMD's research into providing reconfigurable Arithmetic Logic Units (ALUs) [4].

The ongoing extensive work in FPGAs targets various problems including machine learning, edge computing, and big data acceleration [71, 15]. While FPGA deployment is currently becoming mainstream, these devices are mostly used as updatable ASICs in the sense that FPGA configuration is commonly only used at system start and with accelerators that had been crafted for dedicated problems. The motivation of this thesis is to allow a system to generate stream processing acceleration pipelines for problems only known at runtime and by making the best use of the currently available FPGA resources. Problems that can be implemented using operator modules can be accelerated by using partial runtime reconfiguration to build tailored execution pipelines at runtime. This thesis researches the methods, concepts, and benefits of implementing execution pipelines at runtime for FPGA acceleration of big-data problems. While the here proposed concepts apply to a wide range of dataflow problems, this thesis will exemplify the concept of stream processing on OLAP (Online Analytical Processing) database queries due to their costly processing requirements. The OLTP (Online Transaction Processing) queries are handled by the co-processing software since they usually operate directly on primary keys and have small complexity that cannot overcome accelerator overheads [20]. As a dedicated feature, this work will accelerate database queries that are only known at runtime as an example for dynamic stream processing.

It also presents resource elastic techniques to build processing elements which significantly improves flexibility for runtime query schedulers, maximizing performance for a resource target only known at runtime.

1.2 The Big Picture

Traditional Database Management Systems (DBMS) are fully software-based. They are responsible for all operations within a database including interfaces, data storage, and parsing and execution of SQL queries. The critical path of large-data database performance lies within the execution phase of complex queries. This makes databases a clear target for FPGA acceleration.

1.2.1 Overview

FPGA acceleration of queries that are only known at runtime can be implemented by over-provisioning of SQL operators and the activation only of the essential ones upon query parsing. The SQL standard defines 376 reserved keywords, many of which implement operators [44]. Thus supporting the entire SQL standard would require reserving static logic that can provide expensive operations such as logarithms, sin/cos, division etc. Indubitably this will be rather resource-expensive and uneconomical due to all possible SQL operator combinations. We highly anticipate that the dynamic construction of the execution pipeline for database acceleration is the most advantageous approach for general-purpose FPGA acceleration. In order to best utilize FPGAs, there are numerous challenges to be considered:

Memory organization : Data movement poses a large overhead risk. Main memory DBMS primarily store data in the system's DRAM. This allows for low-latency data availability. More traditional DBMS use mass storage devices for permanent data storage and utilize the system main memory only for intermediate data and caching. However, introducing external accelerators to a DBMS can also introduce a new memory space. GPUs and FPGAs have separate accelerator memories (where the host CPU is external) and overheads for data movement to and from the host memory space have to be considered. Straight forward solutions include FPGA acceleration where the host CPU is tightly integrated with the FPGA and where both share the same main memory (such as Xilinx Zynq FPGAs [125] and Intel Stratix 10 SX [43]). FPGA models that do not include

a hardwired CPU (such as Xilinx Virtex FPGAs) can either achieve memory uniformity by implementing a host CPU as a softcore with direct access to the FPGA's local DRAM or at least provide low-latency high-bandwidth memory connection of the accelerator's memory to the host CPU.

Efficient accelerators : The dynamic stream processing should be possible with a resource cost that is comparable to dedicated non-reconfigurable accelerators. The ideal accelerators have a wide datapath, large operating frequency, and implement large utility. *In this thesis, utility is defined as the amount of practical computation performed per pass of the data through the accelerator.*

Hardware layout and interfacing : The dynamic nature of the SQL queries in conjunction with the complexity of database operations characterizes examples for virtually all communication and integration requirements that exist in stream processing systems. This requires careful consideration of the mechanisms that enable multiple accelerators (operators) to co-operate while maintaining maximum energy efficiency and throughput. The system should not target a specific device type, but rather should be an abstracted specification that can serve a wide range of FPGA platforms. For this purpose, a mixture of standard and custom approaches and interfaces need to collaborate.

Software-hardware co-design : The software components of the DBMS need accurate models for accelerator capabilities and constraints in order to generate execution plans that will meet query requirements. The software also requires accurate cost models in order to optimize the stream processing pipeline implementation. This enables factual prediction about any partial reconfiguration overheads as well as for the expected execution time. Using these predictions, the DBMS can fall back to software execution in cases when the input query is not large or complex enough to benefit from FPGA acceleration. With this, the heterogeneous CPU and FPGA system will generally perform better than a pure CPU-based system and overheads for FPGA configuration and operation will only be spent if this improves performance.

Figure 1.1 shows our query execution pipeline using a high-level block diagram. In this project, hardware acceleration uses the FPGA Operating System (FOS) to reserve an acceleration slot and physical memory blocks and to partially reconfigure the hardware if needed [115, 116]. Performing runtime partial reconfiguration poses an

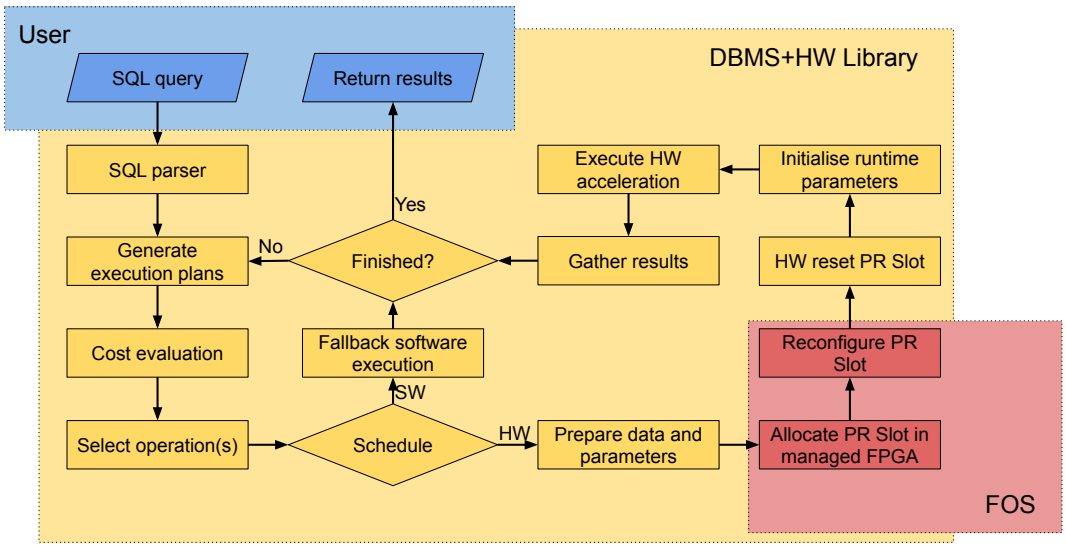

Figure 1.1: Multiple components from multiple systems interact to execute an SQL query. The bulk of operations in this figure are implemented by integrated DBMS and a hardware acceleration library.

overhead that needs to be evaluated against the possible acceleration benefits for a given problem. However, it is not necessary that partial reconfiguration needs to take place for every subquery accelerated in hardware as subqueries with the same operation (but different parameters) can subsequently reuse the same FPGA configuration if supported through generic operator modules.

Succeeding execution, the DBMS needs to reevaluate the execution plans and regenerate them if needed. Many SQL operators (such as join, filter, aggregate) can change the sizes of intermediate tables, which will inevitably lead to the regeneration of the execution plans in order to optimize the scheduler with the consideration of the new details.

1.2.2 Hardware Accelerated Processing Elements

The hardware integration into the DBMS is through enabling a module library (see Figure 1.2 b) that implements various database operation execution units. Ideally, these operators implement resource elastic techniques enabling trade-offs between resource requirements and throughput/utility. This is implemented by providing module alternatives - multiple modules implementing the same database query operator, but with varying size and computational limits. Additionally, operators could be implemented to be composable at runtime. Composable accelerators extend their functionality by

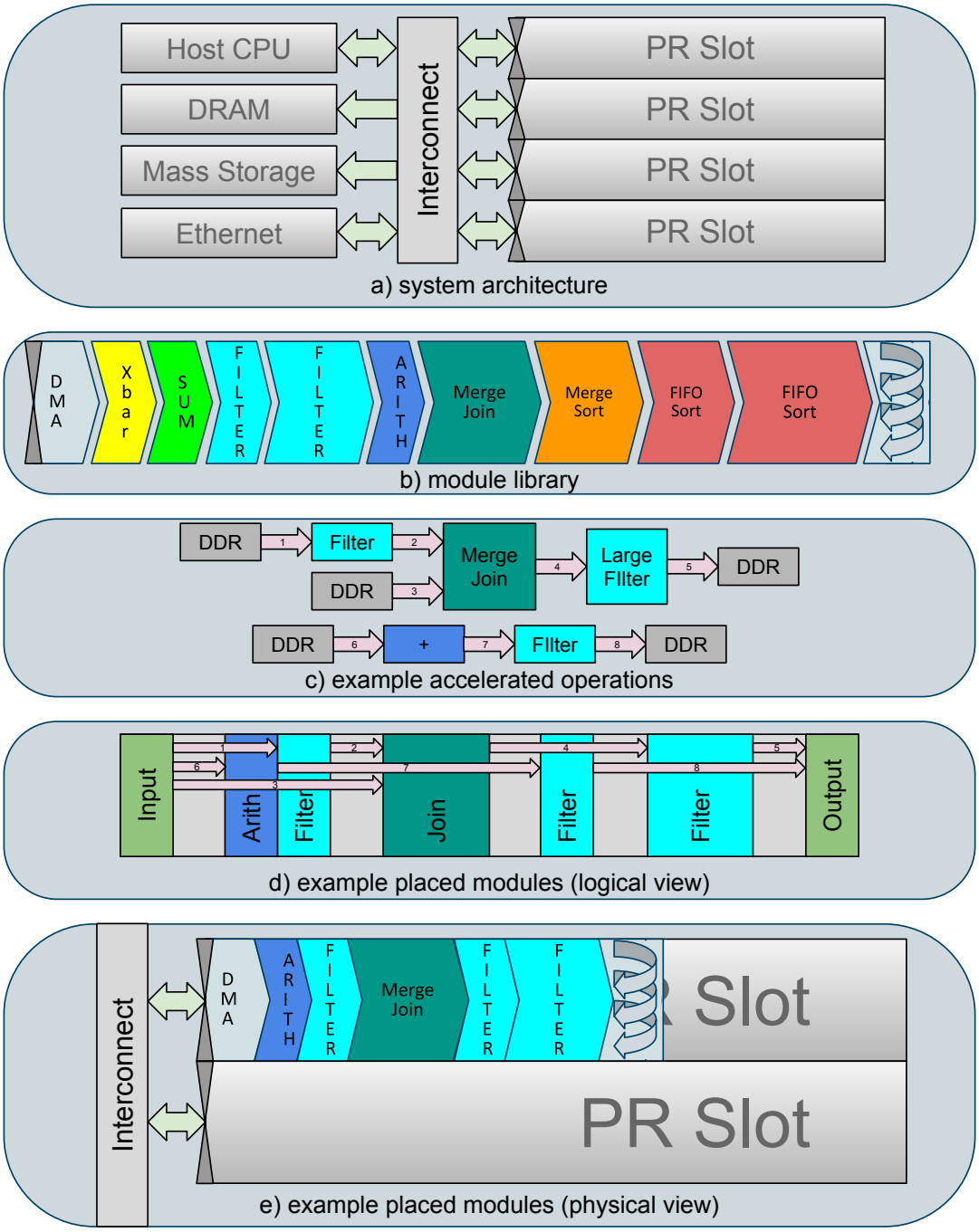

Figure 1.2: a) target system architecture (Host CPU, mass storage controller, and ethernet controller can be soft or hard. Interconnect can be soft, soft+hard, or through PCI express), b) module library containing partially-reconfigurable database operators, c) example subquery hardware execution plan, d) logical placement of example query, and e) physical placement of example query.

stitching together multiple smaller accelerator modules to serve a more complex operator or to provide operators with higher performance such that the smaller accelerators appear logically as a single larger module. Such a technique is not applicable to all SQL operations due to algorithmic restrictions. However, when applicable, it expands the placement options for the runtime scheduler.

The stitchable modules implement a custom uniform interface between them that is optimized for dynamic stream processing and resource elasticity. This custom interface (denoted by the triangle-shaped edges in the modules in Figure 1.2 b) takes into consideration many components and aspects of the system. For example, to ease partial reconfiguration tools it is beneficial to eliminate asynchronous handshakes between modules. Additionally, they should be suitable to fit all needs posed for SQL operators including careful consideration of the amount and size of data records and their organization.

The hardware layer integrated within the DBMS requires tight arrangement between a host CPU and FPGA partially reconfigurable slots (see Figure 1.2 a). The interconnect shown on the figure can be either: 1) extended in both PL-PS for FPGAs that share a die with a host CPU (like Xilinx Zynq), 2) implemented only in PL for softcore host CPUs, and 3) extended through PCIe to an external host CPU. In the latter case, the FPGA memory space can be mapped for direct access from an external host CPU using resizable Base Address Register (BAR) [1]. This does not necessary solve overhead considerations associated with data movement but does provide seamless operation for such topologies.

State-of-art research suggests that the future of cloud deployment will implement partially reconfigurable slots [117]. This is important since FPGAs employ a large set of resources - not only FPGA fabric but also DRAM sizes, DRAM throughput, mass storage and Ethernet. Cost efficiency will lead to cloud providers enabling partitioning of an FPGA chip to be used by multiple users due to the users' varying resource requirements and for enabling resource pooling of FPGA, memory, and I/O resources. Our system targets the utilization of such partially reconfigurable slots. The hardware layer needs to target standard interfaces, to allow for seamless integration and deployment into academically and commercially managed FPGA systems. The most commonly used standard interface in such systems is the Advanced eXtensible Interface (AXI) [5]. In our system, one of the modules needs to translate transactions between a streaming interface and the standard interface (see **DMA** in Figure 1.2 b). This module provides a high level of abstraction for the SQL operators by managing

all physical data allocation and providing data permutation and data prefetching units.

The system targets to support multiple subqueries (including queries from different users) at the same time by utilizing a shared datapath and encoding the data packets within the PR region (see Figure 1.2 c-d). Traditionally, the aforementioned cloud systems provide physical interfaces on only one side of the PR slot, thus we anticipate a dual-direction datapath that turns around at the end of the PR slot. The direction of the streamflow is inverted after all SQL operator modules (see Figure 1.2 e).

1.2.3 Hardware-Software Integration

The module library itself acts as an interface between hardware and software designers. It allows the abstraction and cooperation by allowing the accelerator developers to define numerous runtime aspects (see Figure 1.3):

Bitstream : Providing the already synthesized hardware eases the runtime system by: 1) discarding any impact from the large hardware synthesis times, 2) abstracting most complexity associated with hardware development, and 3) eliminating the software developers' need to interact with FPGA vendor tools.

Physical Specification : This parameter outlines the physical placement and footprint of any accelerator synthesized for a particular FPGA family. Upon module library generation, the system can use string matching algorithms to find possible accelerator placements [34]. This simplified representation of physical aspects is then used by the runtime to find all possible execution plans. This placement specification is also required for the module placement stage as a parameter to the module relocation tool [89].

Cost function : The large importance of accurate models for cost evaluation offloads their generation to the accelerator developer. Providing accurate models enables the runtime scheduler to make decisions that maximize performance and efficiency.

Driver : Drivers translate runtime user and meta data into module parameterization. Thus it is key that the accelerator developer handles the task of producing the lower level abstractions in the drivers and directions how the software developers can integrate them into the higher layers.

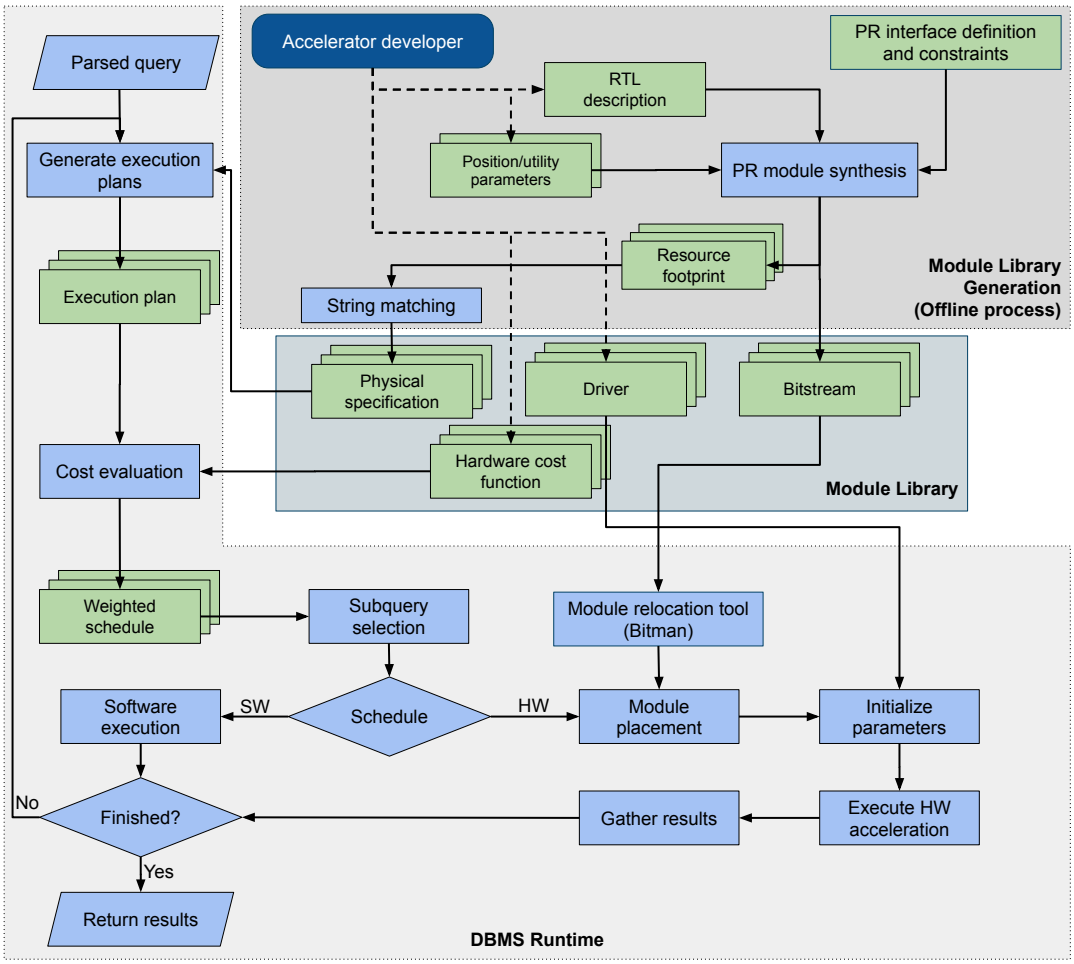

Figure 1.3: The module library is used as an interface between the accelerator development/synthesis and runtime DBMS query execution.

1.3 Scope and Contribution

The proposed system comprises a complex hardware-software co-design problem where the resource allocation problem is decided at runtime. This system, therefore, poses numerous challenges to be solved. This work focuses on the functionality of the hardware implementation by researching:

Dynamic Stream Processing Interface (Chapter 3) : We propose a custom interface for inter-module communication within the streaming PR region. The proposed interface satisfies all described module requirements for processing element specifics about data sizes and stream requirements. The interface targets shared multi-tenant environments by implementing standard external interfacing

for integration with FOS [115, 116] and allows for seamless software integration by supporting memory-mapped register spaces within the partially reconfigurable stream modules. *DSPI enables complex accelerators such as merge sorters to be implemented directly in the PR region, which has not been possible in related work [135, 118]. DSPI achieves this with low wire and logic interface overhead.*

Module Library (Chapter 4) : This work introduces the key concepts in designing scalable streaming accelerators with large utility (see Section 4.1). Using these concepts and algorithmic requirements, we propose efficient designs for most common compute operations as streaming modules with small resource requirements. Moreover, we present efficient approaches for the implementation of filter and sort modules, which are the most widely used operators in database query execution. *Our large utility methodology enables us to execute the complex filtering in TPC-H Query 19 using a single larger module, rather than 147 modules that are required in related work [135], resulting in* $13\times$ *resource cost improvement (see Section 4.3). Additionally, our large utility merge sorter improves performance over state-of-art FPGA sorters [97, 83, 84] by implementing more effective work in each pass of data through the chip (see Section 4.4).*

Resource elasticity (Chapter 5) : While related work on resource elasticity relies on straightforward parallelism provided natively by OpenCL and is limited to monolithic problems [117], we propose the implementation of such techniques with more complex workloads and requirements. This work showcases the techniques and benefits for resource elasticity in streaming accelerators. It proposes methods for resource-elastic designing of complex operators such as filter, sort, join. Building multiple module alternatives enables flexibility for the runtime schedulers allowing the minimization of major data runs and improving performance. Additionally, we research the implementation of composable modules. Designing modules to be composable further enhances the scheduler's pipeline alternatives, which often leads to improved performance. Abstracting the physical placement of subparts of a greater logical operator also enables the use of multiple chained PR slots, including using multiple FPGA boards.

Prototype Implementation and Evaluation (Chapter 6) : This thesis then builds the implemented accelerators as resource elastic modules that can be partially reconfigured. *All modules achieve a streaming throughput of 19.2GB/s. The throughput can scale up by increasing the datapath width and additional pipeline stages.* The work then evaluates FPGA fabric configuration times to enable precise cost models and minimize the runtime overheads. The evaluation concludes by presenting a case study using the TPC-H benchmark [108], where it saturates the available DDR throughput and achieves $2.5 - 5\times$ higher performance than a multicore system with PostgreSQL.

The researched topics provide solutions for the offline module library generation, module interfacing, scalability, utility, and partially their runtime integration and management. The research of integrating the proposed hardware into existing DBMS and exploring heterogeneous scheduling is currently ongoing by PhD candidate Kaspar Mätas [66].

1.4 Publications

- **Kristiyan Manev** and Dirk Koch. Large Utility Sorting on FPGAs. In *International Conference on Field-Programmable Technology (FPT)* 2018

  Researched and proposed a large utility sorter that is also incorporated into the proposed database system (see Section 4.4). It minimizes the number of runs of the data through the chip. This is particularly important for expensive operators such as sorting. The methodology of designing large utility operators is also researched in Section 4.1.

- **Kristiyan Manev**, Anuj Vaishnav, Charalampos Kritikakis, and Dirk Koch. Scalable Filtering Modules for Database Acceleration on FPGAs. In *International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART)* 2019

  Researched the application of boolean normal forms for the case of database filtering with complex requirements. Proposed large utility filter modules that are incorporated into the proposed database system (see Section 4.3). This work also discussed several accelerator interface requirements that have later been incorporated into DSPI (see Section 3.3).

- Anuj Vaishnav, Khoa Dang Pham, **Kristiyan Manev**, and Dirk Koch. The FOS (FPGA Operating System) Demo. In *International Conference on Field-Programmable Logic and Applications (FPL)* 2019

  Researched and implemented RTL accelerators into FOS (the FPGA Operating System), while the operating system was originally targeted mostly for accelerating using High-Level Synthesis (HLS). This thesis utilizes FOS to implement basic abstraction from low-level system components. The integration to FOS' interfaces is implemented through the proposed DMA module (see Section 4.2).

- **Kristiyan Manev**, Anuj Vaishnav, and Dirk Koch. Unexpected Diversity: Quantitative Memory Analysis for Zynq Ultrascale+ Systems. In *International Conference on Field-Programmable Technology (FPT)* 2019

  Researched, implemented, and released to the public a benchmarking tool for the state-of-the-art Zynq UltraScale+ systems. Together with co-authors utilized the tool to profile and analyze the FPGAs targeted by FOS and developed a set of guidelines for maximizing achieved memory performance. The work has been used throughout the designing of the proposed database system as guidance for efficient implementation of a credit-based system (see Section 3.1), accelerators (see Chapter 4), and evaluation (see Chapter 6).

- **Kristiyan Manev** and Dirk Koch. Resource Elastic Database Acceleration. In *International Conference on Field-Programmable Logic and Applications (FPL)* 2020

  This work researched the achievable benefits from the application of resource elastic techniques to available database operators. It showcases Pareto-Front schedules that trade off between cost and effective throughput while targeting a standard query benchmark [108]. This enables a runtime system to fully utilize the available resources and maximize performance. This thesis expands on the proposed resource elastic techniques and benefits in Chapter 5.

Chapter 2

Background

In this chapter, we outline the background in FPGA technology, databases, and database acceleration. First, we describe the target FPGA fabric, software integration, and partial runtime reconfiguration in Section 2.1. Section 2.2 introduces the concepts of relational databases. Finally, Section 2.3 outlines the related work in database acceleration.

2.1 Field Programmable Gate Arrays

FPGAs implement a 2D mesh of programmable resources with configurable interconnect wires. Implementing our module library (see Chapter 4) often required careful consideration of the underlying FPGA fabric to achieve higher efficiency or reduce resource cost. The programmable FPGA primitives include 1) Configurable Logic Blocks (CLB) which provide the main FPGA logic, 2) Digital Signal Processors (DSP) which provide specialized arithmetic, 3) on-chip memories such as distributed memory, BlockRAMs, and UltraRAMs.

For the basic software layer in our database acceleration system, we use the FPGA Operating System (FOS) [115, 116] on Xilinx Zynq UltraScale+ FPGA. More precisely, we target the largest FPGA that FOS currently supports (the Xilinx ZCU102). To employ partial runtime reconfiguration we utilize tools such as GoAhead [13], TedTCL [118], and BitMan [89].

2.1.1 FPGA Primitives

Configurable Logic Blocks

The main processing resources in FPGAs are Lookup Tables (LUTs), which are located in CLBs [124]. LUTs provide small memories to implement Boolean functions by looking up truth tables for a given input. In Xilinx UltraScale+ (US+) FPGAs, LUTs are implemented as two 5-bit address and 1-bit data memories with an optional multiplexer. They can be configured as 5-bit input 2-bit data or 6-bit input 1-bit data. Additionally, CLBs provide Flip-Flops (FF) to enable insertion of pipeline stages and accelerated carry chains module that optimizes adder and multiplier implementations.

Digital Signal Processors

To improve the implementation of multiply operations, US+ provides dedicated columns of DSP modules [127]. DSPs provide a hardwired multiplication operator followed by a configurable Arithmetic Logic Unit (ALU) that implements addition, subtraction, and all boolean operations. This feature can be used in various operating modules and was utilized by our large utility arithmetic module (see Section 4.1.4).

FPGA on-chip memory

FPGAs usually do not have access to caches [1] but rather have Direct Memory Access (DMA) by connecting to an external DDR memory controller or implementing a DDR memory controller inside the FPGA. To optimize the implementation and execution of accelerators, FPGAs intend the use on-chip scratchpad memories that provide precise access latency, high throughput, and relatively low capacity. Xilinx US+ provides three types of on-chip memories:

Distributed memory : A selective subset of the available LUTs in the FPGA fabric are LUTM. The configurability of LUTM resources is a superset of those of LUT resources. LUTM extend LUT providing a write port to the small 64×1-bit or 32×2-bit memories, thus enabling them to be used as small dynamic 8-byte memories [124]. LUTMs have the advantage of an asynchronous read port, which also enables direct asynchronous integration with compute. We utilize LUTMs in our modules throughout the implementation of our module library, but most notably to hold reference data in our filter module (see Section 4.3).

[1] Xilinx Zynq FPGAs can use the ARM APU caches through ACP AXI port

BlockRAM : These memories are implemented in dedicated memory columns in the FPGA fabric. They realize true dual-port 36 Kbit memories that are highly versatile [126]. The two synchronous access ports have individual clocks and separate programmable data width (i.e. the two ports can virtually treat the same memory as a memory of different depth-width organisation). Moreover, the dedicated resources support various address collision handling methods, built-in resources for memory cascading, and converting the memory into a FIFO. BlockRAMs are key components for implementing our sort and join modules. Also, we use BRAMs to translate between the shell clock domain that provides the I/O infrastructure and the accelerator clock domain.

UltraRAM : Similarly to BlockRAMs, UltraRAMs are also implemented as dedicated memory columns in Xilinx US+ fabric but are less versatile. They implement a 288 Kbit memory as a 4096x72-bit memory [126]. The cascading resources are applicable only for certain configurations and the provided two access ports use a shared clock and execute sequentially, thus not supporting address collision. Additionally, UltraRAM resources are only available in the largest and most expensive FPGA chips and are in rather limited supply. However, in future implementations of our proposed system for such devices, most of our components that utilize BlockRAMs will benefit from utilizing the larger UltraRAMs.

The introduction order of the three types of memories is sorted in ascending order by their capacity and descending order by their flexibility and availability. The inverse proportionality between flexibility and capacity outlines the available trade-offs in the selection of proper resources when designing FPGA accelerators.

The FPGA resources are organized into clock regions. In UltraScale+ devices, a clock region defines a fixed height of 60 CLBs, 24 DSPs or 12 BlockRAMs.

Interconnect

In addition to the compute and memory resources, FPGAs provide a programmable interconnect. The interconnect implements multiplexers for the available on-chip wiring between compute and memory resources. The routing fabric is mostly homogeneous regardless of the resources available which is important as this enables the synthesis of partially reconfigurable modules that can also be relocated to different parts of an FPGA than originally implemented.

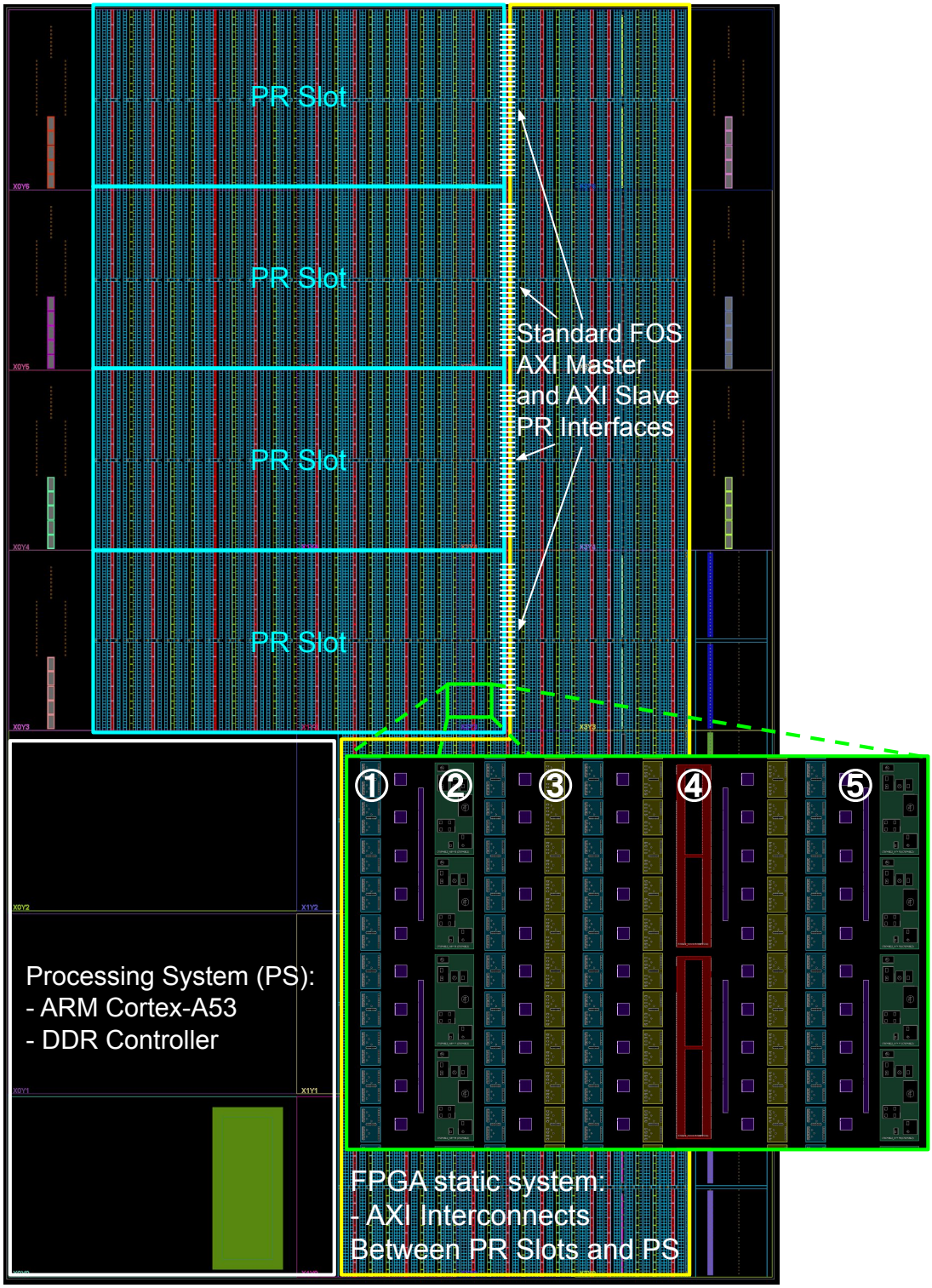

Figure 2.1: FOS supports four PR slots on ZCU102. The static system implements AXI interconnects between the PS and PR slots. FPGA resources are equal in these four slots. FPGA resources within any column are identical. Most column types are: ① - LUTM, ② - DSP, ③ - LUT, ④ - BlockRAM, and ⑤ - Interconnect.

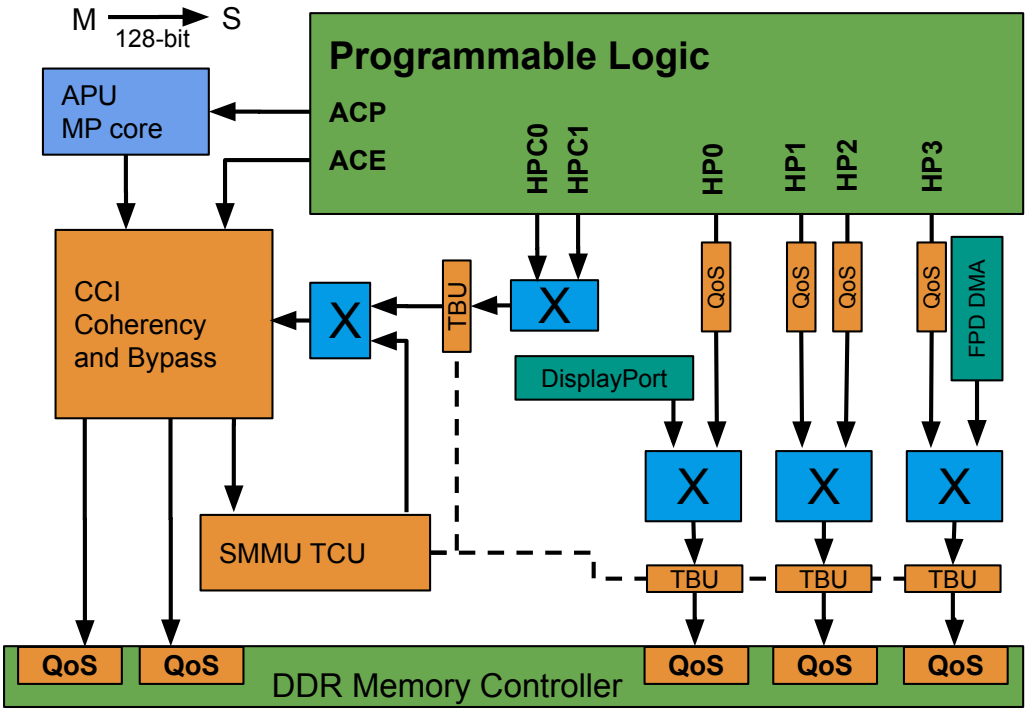

Figure 2.2: Xilinx Zynq UltraScale+ memory subsystem uses AXI interconnects between FPGA (PL) and hardwired memory controller [125, 5].

2.1.2 FPGA Operating System

The FPGA Operating System (FOS) proposes an Acceleration-as-a-Service (AaaS) approach that can utilize heterogeneous compute [115, 116, 117]. It currently supports two FPGA boards: Xilinx Ultra96 and Xilinx ZCU102. As Figure 2.1 shows, on ZCU102 FPGA, FOS implements 4 PR regions of one clock region height and identical size and resources. The system can also provide a combination of neighbouring PR regions as larger a single larger PR slot [114]. FOS originally targeted only High-Level Synthesis (HLS) to allow seamless programmability of the FPGA together with the CPU [90, 112], however, FOS also supports the acceleration using RTL accelerators [116]. FOS was used in this project as the foundation software layer for accelerator integration (see Section 4.2).

2.1.3 Xilinx Zynq UltraScale+

Xilinx Zynq Ultrascale+ systems are being deployed for exascale computing [67], edge hubs [59], and other embedded devices due to their tight and efficient integration of the

	ZCU102	Ultra96
FPGA	XCZU9EG-2	XCZU3EG-1
BlockRAM36K	912	216
DSP	2,520	360
Logic Slices	34,260	8,820
Processing System	Cortex-A53 + R5	Cortex-A53 + R5
APU Frequency	up to 1.5GHz	up to 1.5GHz
Level 1 Cache	32 KiB	32 KiB
Level 2 Cache	1 MiB	1 MiB
Bank Organisation	4 Bank Groups $\times$ 4 Banks each	2 Ranks $\times$ 8 Banks each
DDR Capacity	4GB	2GB
DDR throughput	2400 MT/s $\times$ 64-bit = 19.2 GB/s	1066 MT/s $\times$ 32-bit = 4.264 GB/s

Table 2.1: Platform specification for ZCU102 and Ultra 96 board.

host system with an FPGA fabric for acceleration. The Zynq UltraScale+ systems implement a hardwired DDR memory controller into the Processing System (PS) part of the die. The ARM system-on-chip, which consists of processors, AXI interconnects, and the memory controller is identical for all Zynq UltraScale+ devices [125]. Figure 2.2 shows a simplified view of the connections between programmable logic (PL), ARM Cortex-A53 (referred to as APU), and the memory controller. These connections are also the key point of interest to FPGA developers. Zynq US+ devices provide eight 128-bit AXI ports, each capable of up to 300 MHz. They are divided into four types, depending on their cache coherency:

High-Performance (HP) $\times 4$: no cache coherency

High-Performance Coherent (HPC) $\times 2$: I/O coherency

Accelerator Coherency Port (ACP) $\times 1$: I/O and L2 cache coherency

AXI Coherency Extension (ACE) $\times 1$: full coherency

In total, there are four types of memories available in Zynq Ultrascale+ systems: 1) in-fabric memory (which consists of block RAMs and distributed memory), 2) Cortex-A53's level 1 and level 2 caches, 3) on-chip scratchpad memory (256KB on-chip memory that is tightly coupled with the Cortex-R5 processor), and 4) off-chip DDR memory. This means that depending on the application and the memory used, state information is located inside the CPU, inside the scratchpad memory, inside the programmable logic, or in external RAM. There also exist other components between PL

and the DDR memory controller (see Fig 2.2): Translation Buffer Unit (TBU), Multiplexers, a Display-port, a Full Power Domain DMA engine (FPD DMA), and Quality of Service (QoS) buffers. These can potentially affect the behaviour and achievable memory throughput of hardware accelerators. However, these are often not considered by the developer as they exist inside the ARM SoC and Xilinx's Vivado does not provide direct control over these components.

Commonly used FPGAs are the Xilinx ZCU102 (XCZU9EG) due to the large interest for exascale deployment [3] and the Ultra96 (XCZU3EG) due to its wide availability and low cost. Table 2.1 lists the key characteristics of their memories.

2.1.4 Partial Runtime Reconfiguration

The concept of building a pipeline at runtime to tailor to the needs for a specific problem has existed for decades [119, 37]. Building custom pipelines at runtime requires the utilization of the partial reprogrammability capabilities of the FPGAs. Runtime reprogrammability has also led to the proposal of DPGAs, which are FPGAs that can hold multiple bitstreams at once and allow for very fast context switch [23, 24]. Although PR techniques and applications have been studied for so long, vendors have yet to provide solutions to many remaining challenges. For example, Xilinx supports tools for partial reconfiguration that split the design into fixed PR region and static part [128]. Every minor change in the static part (e.g., updating an IP) is followed by the requirement that all PR modules have to be resynthesized. Additionally, all PR modules cannot be relocated at runtime to other PR regions with the same resource footprint. This drives vendor tools unsuitable for our system.

This work utilizes GoAhead [13] together with a TCL library [118] to define PR interfaces and guide place & route tools to produce relocatable modules with predefined interfaces implemented as floating wires in a regular pattern. The module bitstreams get relocated using BitMan [89]. Using these tools allows for the definition of an interface to be used for dynamic module stitching at runtime.

2.2 Relational Databases

Relational databases have been proposed for the first time more than half a century ago [22] and are the most widely used organisation of data in databases since. Relational databases organize the data into a grid of columns (also called fields) and

rows (also called records, tuples). All rows in a table hold the same types of data and those types are defined by the columns of the table. The term relational expresses how multiple tables within the database can be retrieved in a linked manner. Tables in relational databases tend to have one column that holds the table's **primary key**, which is highly utilized in data storage, ordering, and row identification. Some tables also implement one or multiple columns storing **secondary keys**, which form relations to the primary keys of other tables (thus relational database).

Although database queries can be implemented using imperative languages [47], most relational Database Management Systems (DBMS) use the widely accepted Structured Query Language (SQL) standard [44] as an interface to the database. The language provides constructs to insert, retrieve, modify, and delete records as well as change the structures and relations of tables. However, the available systems that implement SQL do not fully follow the standard as they introduce custom query operators and/or do not support all definitions of the standard [44, 107, 82]. The large query complexity is apparent when observing the wide range of operators and constructs in the SQL standard [44] as well as the complex FSM required for its parsing [94]. However, our system can focus on accelerating the most used and compute-intensive operators in analytical subqueries, while other operations will be executed in software. Typically, mandatory operators in analytical subqueries on relational databases are restriction and join with other common operators being sort, aggregate, and arithmetic. The proposed system accelerates restrictions by researching a large utility filter module (see Section 4.3). In terms of join, we focus on accelerating equi-joins, since they are typically utilized in relational databases to join tables by their primary and secondary keys (see Section 4.5). Sorting is part of the SQL toolset and is often used in order to present the query results in a specific order. However, sorting is a vastly important operation in big-data query execution, since it is a required step in sort-merge-join which is more scalable than hash-join [110]. Additionally, grouped aggregations are suitable for sort-aggregate implementation on FPGAs [26]. Since sorting is a costly operation, but it is widely required in query acceleration, this work researches the field of large utility high-performance FPGA sorting (see Section 4.4). Finally, the use of arithmetic is present within result delivery as well as calculating filter conditions. Arithmetic modules are not complex when implemented in streaming systems as they are straightforward to be pipelined and have no feedback loops. However, the large utility methodology for implementing general operators such as arithmetic is also researched in this work (see Section 4.1.4).

For evaluation, we select the TPC-H decision support benchmark due to its broad industry-wide relevance, queries operate on large volumes of data, and are more complex than most OLTP transactions [108]. TPC-H is an online benchmark where systems may not optimize the DBMS for specific queries prior to their execution [79]. TPC-H is a widely accepted benchmark and the most notable argument against TPC-H is that it is a difficult benchmark for query optimizers due to its uniform data distribution [79]. However, this work does not look into query optimizers and, in terms of operator acceleration, the uniform data distribution can only give an advantage to hash-based operators, which we do not implement.

Our system currently lacks integration of parser and optimizer, thus we select a single query that we parse manually to be implemented in a dynamic stream processing system (see Section 6.3). TPC-H Query 19 is ideal to test the proposed system as it has medium to strong complexity for aggregation, join, and expression calculation. It also requires the cooperation of all identified major operators: filter, join, arithmetic, aggregate, and sort. At the same time, it is straightforward to parse with no major optimisations for subqueries [17], which does not put us in advantage or disadvantage due to our manual query parsing.

2.3 Database Acceleration

CPU

In general, traditional software implementations of DBMS can be compute-bound depending on query complexity. To solve this, researches have been exploring the utilization of Single Instruction Multiple Data (SIMD) co-processors of modern CPUs. SIMD accelerators provide a great increase in computational resources. Utilizing (4-16)-way SIMD co-processor has resulted in up to 3.3× higher performance than conventional CPUs in executing database queries [134, 21, 91]. However, these approaches optimise mostly for selected workloads and tend to put high stress on the CPU caches which can prevent their scalability to wider SIMD vectors.

Another approach to increase the performance of traditional CPUs is by utilizing manycore systems for database acceleration [49, 109]. Similar to SIMD, these algorithms are suitable for acceleration for a subset of database operators. Not all traditional optimizations can be applied to SIMD and manycore implementations because algorithms with large control overhead or shared global structures may not optimally map to parallel implementations (e.g., hash table generation).

GPU

GPUs provide greater memory bandwidth, parallelism, and compute resources than CPUs. However, similarly to CPU SIMD, GPUs suffer from selective adaptability of algorithms to implement. As accelerators with local memory, GPUs can suffer from large data movement between memory spaces. Bingsheng He proposed a GPU query executor that achieved up to 30% speedup on TPC-H over an optimized CPU implementation [38]. GPUs can also use the system main memory directly, but mostly fully streamed algorithms have to be used to maximize the benefits [50].

FPGA

The research on database acceleration using FPGAs looks back on a two-decade-long history [85, 99]. Traditionally never changing queries could be accelerated using FPGAs by synthesizing optimized hardware for a particular query. For instance, Glacier can compile queries into synthesizable VHDL code that can be used to generate a hardware accelerator for a given query [73, 74, 75]. Malazgirt et al. propose using high-level synthesis tools to compile SQL queries into FPGA bitstreams [60]. Xilinx also supports the Vitis Database Library, which allows the implementation of SQL queries as C++ code that then gets implemented using HLS [133]. Similar methods or handwritten RTL code can result in an energy-efficient and high-throughput design for a particular query. However since large designs take hours for synthesis, this approach is not applicable for the acceleration of queries known only at runtime.

Another approach is to implement static accelerators that can be initialized with parameters at runtime to accelerate common query operations. IBM's Netezza platform provides a commercial storage system, where FPGAs are integrated between mass storage devices and host servers and enable data restriction, (de)compression, and reordering operations [29]. Similarly, Swarm64 augments PostgreSQL [107] to utilize FPGAs for data permutation, (de)compression, and filtering [106]. These approaches have been very promising due to the limited I/O of mass storage. With the recent rollout of cheap and fast NVMe SSDs, however, the significantly larger FPGA costs are rendering this approach less economically viable. Xelera is another company that provides commercial acceleration in the field, but they focus only on database analytics and data hashing [56, 122]. Sukhwani et al. propose a CPU+FPGA query execution system, where the FPGA accelerates data (de)compression as well as restriction operations [104]. Ibex is a proposed storage engine that can accelerate restriction and

grouping operations before parsing the requested database tables to the software [121]. Casper et al. propose a system that implements acceleration for sort and merge join operations as well as field masking and reordering [19]. Salami et al. propose the AxleDB system which implements filter, sort, join, and aggregate modules forming a stream processing ring [98]. This approach for query acceleration can lead to good throughput and energy-efficiency results when the queries are fit for acceleration but usually tends to provide limited parameterization. Additionally, statically reserved resources lead to imbalances between reserved resources and required query operations at runtime. This approach also increases challenges with designing a system that can support most of the available SQL operators, as it requires statically placing and routing hardware for every supported SQL operation. In summary, static solutions commonly intend to incorporate an over-provisioning of resources that we can overcome using reconfigurable accelerators.

2.3.1 Dynamic Database Acceleration on FPGAs

The idea of building execution pipelines at runtime from dynamically implemented modules is relatively new [40]. The ReCoBus allowed the dynamic building of stream processing pipelines at runtime by placing encapsulated modules implementing the custom interface [53, 52]. It allowed the execution pipeline to be constructed using modules of varying number, size, and placement [81]. Techniques such as providing module footprint variants have been proposed to minimize the external fragmentation when dynamically building execution pipelines [120]. The GoAhead project took over in providing a toolchain to support implementing partially reconfigurable modules for dynamic pipeline building [13].

In 2012, Jensen proposed a stream processing system to accelerate database data filtering and string matching by building the execution pipeline at runtime from dynamic modules [48]. Meanwhile, Dennl et al. proposed a similar system for arithmetic, compare, and boolean operators [25]. Later, Dennl et al. added aggregations to the system [26]. Becher et al. introduced merge join, linear sort, and merge sort to the system and showcased the achieved high energy efficiency [11]. Becher et al. also proposed the use of a bloom filter to decrease the number of records prior to join operations [16, 12]. Later Ziener et al. proposed a full system that also adds hash joins and other support modules [135]. The complete system proposes merge sort, merge join, and hash join as part of the static system and four PR regions that can accommodate arithmetic, boolean, compare, aggregate, linear sort, and reorder PR modules.

Most recently, Vesper proposed another approach for a dynamic database acceleration system [118]. Vesper introduced means to encode virtual streams that allow the reuse of the PR datapath for time-multiplexing, thus can stream several different tables in the same run. The work is mostly focused on the partial reconfiguration aspects of such a system and does not provide designs and evaluations for most database aspects and operators.

2.3.2 Discussion

FPGAs have been shown to provide database acceleration. Dynamic stream processing is the state-of-art approach for executing runtime-known problems, while utilizing runtime-known resource pools. Despite the advances in dynamic stream processing, we identify three key areas that need improvement:

Dynamic stream processing interfacing: A significant constraint of the system proposed by Ziener et al. [135] is the passive stream interface between operators that is not able to support multiple stream sources simultaneously or PR modules with active operation (see Section 3.2.3). This forces the authors to implement merge sort, merge and hash joins in an overprovisioned manner as part of the static system. Since these key operators are part of the static system, the authors also have to implement multiple fragmented PR regions around them, instead of one large PR region. This allows the placed PR modules to work on either the inputs or outputs of the static accelerators. While Vesper [118] proposes a method of encoding streams and achieves multiple active modules, the work is still insufficient to support all main operators (e.g., it cannot support merge sort operator). We solve these challenges by proposing a new Dynamic Stream Processing Interface (DSPI, see Chapter 3). DSPI satisfies the requirements of all main operators (see Section 3.2.2). What is more, state-of-art work in dynamic database acceleration [135, 118] have not proposed means for straightforward software integration, while DSPI enables seamless memory-mapped register integration (see Section 3.3.4).

Module library: In the state-of-art system proposed by Ziener et al. [135], the methodology of implementing small operators as PR modules leads to large internal fragmentation, while often queries that suit FPGA acceleration are complex queries with large operator requirements. We propose improved accelerators with incorporated large utility methodology during design and implementation (see

Chapter 4). This methodology is suitable for general operators (see Section 4.1), but also for specialized operators such as our proposed DNF filter module (see Section 4.3), which lets us improve both resource requirements and achieved throughput (see Section 4.3.4). While the authors in related work [135] exemplify an 8-way static merge sorter, we enable a 64-way merge sorter (8× more sequences merged at once) implemented as a PR module in a part of a FOS ZCU102 PR slot (see Section 4.4). Additionally, they propose an one-to-many join module, and while this is sufficient for most real-world queries, this thesis proposes a many-to-many merge join operator (see Section 4.5).

Elastic operators: The concept for resource elastic operators has been briefly discussed by Vesper [118]. However, while a topology for resource elastic merge sort operator is proposed, it is ambiguous how to integrate it into the system using the proposed PR interface. What is more, the proposed merge sorter achieves small utility and is fragmented into three different PR modules (buffer, linear sort, and tree sort). State-of-art work on resource elasticity has been targeting monolithic problems with forced OpenCL workgroup parallelism [114, 117]. However, the presented approaches are not suitable for dynamic stream processing and especially for database acceleration with its heterogeneous operator and data requirements. This thesis enables the application of resource elastic techniques to dynamic stream processing exemplified on database acceleration (see Chapter 5). It discusses the approaches for achieving resource elasticity in stream processing (see Section 5.1) and the expected benefits (see Section 5.1.3). Finally, the work applies the techniques to three common database operators: 1) filter (see Section 5.2), 2) sort (see Section 5.3), and 3) join (see Section 5.4). Resource elasticity will then allow a runtime system to allocate available resources to the operators such that the system maximizes overall performance. For example, we can stitch three resource elastic 64-way merge sort operators in ZCU102 FOS slot and achieve 192-way merge sorting (see Section 5.3.3), while the system proposed by Ziener et al. [135] exemplifies only an 8-way static merge sorter (24× fewer sequences merged at once).

Chapter 3

Dynamic Stream Processing Interface

This chapter introduces a PR-capable dynamic interface to be used by processing elements for seamless operation and integration when stitched at runtime. First, we evaluate the hardware capabilities and accelerator requirements:

- FPGA acceleration is commonly bottlenecked by the memory subsystem. To maximize our understanding for efficient memory use and achieve large effective memory bandwidth, we study quantitatively the memory subsystem of our target FPGA family (see Section 3.1).

- Stream processing accelerators have a large range of interface requirements such as data rates and stream addressing capabilities. We study these requirements to aid the design of capable interface (see Section 3.2).

With the consideration of our findings, we propose our interface and protocol for dynamic stream processing in Section 3.3.

3.1 Memory Analysis of Modern FPGAs: Requirements and Expectations

Xilinx Zynq Ultrascale+ systems are being deployed for exascale computing [67], edge hubs [59], and other embedded devices due to their tight and efficient integration of the host system with acceleration fabric. With the extensive growth of the FPGA fabric in the recent generation of programmable chips, many acceleration problems have their computational needs saturated. On the other hand, most accelerated problems

nowadays suffer from a lack of sufficient memory throughput or substantially large latency [32, 95, 100, 63]. This challenge holds true especially for streaming applications where the achieved external throughput is key to acceleration. The main causes of data starvation in most systems are not necessarily the lack of corresponding hardware resources and capacity, but insufficient awareness of the underlying system components when designing for FPGA acceleration. Due to many abstraction layers including caches, naturally, the software community in computer sciences has grown to be less aware of the organisation and complexity of the underlying memory systems. However, this is not the case when utilizing hardware resources such as FPGAs, where developers tend to have direct access to softcoded or hardcoded implementations of DRAM controllers. Providing this control can often backfire due to the large complexity of these subsystems [28]. Due to the lack of relevant studies of the memory organisation, performance and best practices for the state-of-art FPGA systems (Xilinx Zynq UltraScale+), it was of necessity to research the field [63].

Overall, an accelerator developer usually expects the following behaviour from the memory system:

1. Same type of AXI ports should show the same performance behaviour.
2. Increasing burst size should improve both read and write performance until it saturates (a logarithmic-like relation).
3. Increasing the number of AXI ports to memory should increase total memory throughput linearly or sub-linearly.
4. Memory behaviour should be similar across different boards with the same memory controller except for the highest throughput achievable from the DDR memory available on a particular board.
5. Multiplexing in the PL and PS should exhibit similar throughput behaviour except for the higher latency overhead posed by the soft-logic AXI multiplexing.
6. Sequential memory access patterns should provide considerably higher performance than random access patterns.

3.1.1 Experimental Setup

In our experiments, we consider two widely available Zynq UltraScale+ boards - a ZCU102 Evaluation Board and an Ultra96. Our prime objective for evaluation is the

DDR memory alone as in most cases caches are useful only for the CPUs due to their small size. This is because using on-chip memory as a scratchpad memory in the programmable logic (PL) commonly provides better control, throughput and latency for accelerators than caches. Furthermore, the memory performance can become more unpredictable due to the interference caused by applications running on CPUs and the accelerators in PL. The goal is to examine memory effects that preliminary relate to accelerators located in the PL part of the system but that operate on DDR memory connected to the ARM SoC. We, therefore, do not examine any caching effects. The analysis of available bandwidth from the Cortex-A53 and Cortex-R5 cores have already been researched [8].

Appendix A.3 shows our detailed analysis of the memory subsystem. We describe the hardware and experimental setups. Then we run the experiments and collect and analyse the results.

3.1.2 Experimental Results: Conclusions and Best Practices

To extensively study individual parameters, our evaluation is based on synthetic and real-world applications with varying parameters such as the number of AXI ports, combinations of AXI ports, burst sizes, frequency, access patterns, address space organisation, multiplexing in PL vs PS, and Quality of Service. In general, our findings show that 1) 4 out of 6 common assumptions about memory behaviours do not hold and the remaining 2 do only in certain circumstances (see Section 3.1 and Appendix A.3), 2) the achievable peak throughput is 92.5% and 75% of the theoretical DDR throughput for Ultra96 and ZCU102 respectively, and 3) the default memory behaviour across boards, as well as the AXI ports of the same type on same board, can be very different. Moreover, a case study showed that just by performing our general conclusions (listed below) for memory optimization, an accelerator performance can be improved as much as 46% [63]. This shows that optimising the memory for the cloud and edge environments, as well as embedded systems in general, is of prime importance. In particular, the following general conclusions can be drawn for when working with Zynq UltraScale+ systems:

- Using all AXI HP ports does not always guarantee the highest read and write throughput. Some AXI combinations perform better than others.
- The same ARM DDR controller chip and AXI interface on different boards can lead to different memory behaviours based on the pre-programmed QoS and

priority settings. It is recommended to set these parameters explicitly for the applications.

- Read operations are prioritised heavily over write operations. While this does not necessarily hurt performance, having read-bound accelerators with write-bound accelerators will put the latter at a major disadvantage.

- On average, 128 and 192 Byte bursts often provide near peak throughput.

- Multiplexing AXI in programmable logic (PL) with AXI SmartConnect IP can provide better performance distribution than multiplexing in the ARM SoC at the cost of higher latency and use of FPGA logic (28% for Ultra96).

- Using large burst sizes reduces the throughput overhead of multiplexing AXIs in PL to almost negligible.

- Using higher frequency in programmable logic allows better utilisation of memory throughput but the benefits depend on AXI combinations and DDR memory characteristics.

- It is desirable to use large burst sizes for accelerators in multi-tenant environments for minimal throughput overhead due to rapidly changing access patterns at the DDR controller.

Considering these results, the designers of multi-tenant FPGA shells are forced to take action about careful management of not only DRAM allocation, but also allocation and management of the available memory bandwidth. We show that it is relatively easy to implement an accelerator that steals or corrupts the available performance for other users in an unmanaged environment. We use these results for the implementation of accelerators and efficient packing of the database data into memory transactions. The results have also contributed towards improving the FPGA Operating System (FOS) [116, 117], which our system targets.

The complete set of results (4800 data points of 50 seconds runtime each) and the released benchmark are publicly available[1] in order to allow the research community to exploit our observation for, in many cases, free performance tuning in terms of FPGA logic.

[1] Graphs, data, hardware, and tutorial located at github.com/kmanev/ZynqUSp-AXI-Speedtest

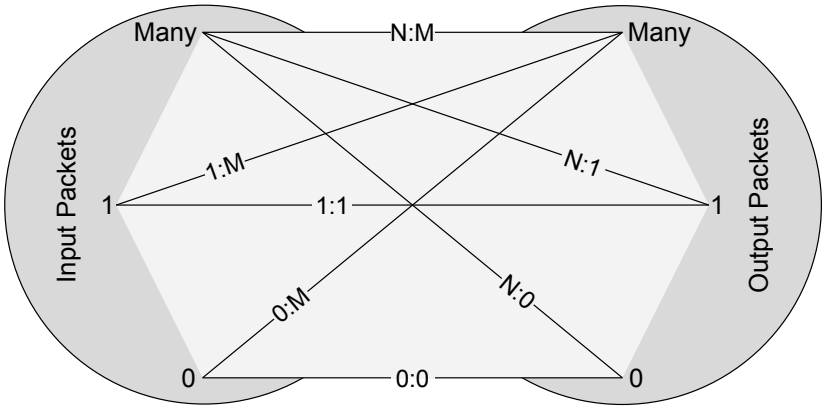

Figure 3.1: Due to high diversity in operations, there are stream modules that can generate any of {*0, 1, Many(M)*} output packets from {*0, 1, Many(N)*} input packets.

3.2 Stream Processing Elements I/O Requirements

Processing Elements (PEs) that operate on streamed data can have a range of interface requirements. Key considerations need to be taken in order to accommodate accelerators with varying consumer and producer rates of records and specific data requirements. Evaluation of possible operators based on effective data rates and stream capability requirements is fundamental for the definition of an efficient and effective dynamic interface.

3.2.1 Rates of Produced and Consumed Data

One method to classify modules is by observing their data consumption and generation rates in the system. Considering the case of provisioning an equal throughput capacity on the input and output sides of each module, data-bound modules can be input-bound, output-bound, or balanced (equally bound on both input and output interfaces). Additionally there exist operators that are fully data-dependent and their behaviour can be approximated prior to execution mostly by using profiling data. Common groups of modules with respect to their I/O ratio are shown in Figure 3.1. The figure expresses classification based on how many output records are generated per amount of input records.

There are seven identified input-to-output ratios:

Zero-to-Zero (0:0) Modules that do not consume or produce data packets. Instead, their functionality is to observe all packets passing through.

Example: Performance Monitoring Unit (PMU), statistics collection

One-to-One (1:1) : Modules that require stream inputs, apply computation to the streamed data and pass the data to the output. Some of these operators (filter, arithmetic, permutation) can stream the data through while applying their corresponding computation by implementing static pipeline stages. Their integration into a streaming system is rather trivial due to the lack of requirements for complex internal data management. Others (sorting) also do have a constant ratio between consumed and produced data but might require more complicated internal data structures and computation. This is caused by the blocking nature of many sort operators.

Example: arithmetic, filter, permutation, sorting

Zero-to-Many (0:M) : Modules that generate data based on initialization parameters. They do not require stream inputs during runtime. Such modules can have arbitrary complexity depending on their functionality.

Example: random data generation, prime number sequences

One-to-Many (1:M) Modules that generate 1 or more output packets per input packet. Such modules are output-bound. Generally, these modules need to implement internal buffering mechanisms for incoming input packets to avoid data hazards in scenarios of output congestion.

Example: SQL UNION ALL

Many-to-Zero (N:0) Modules that consume all data, compute and then the data is not needed to be forwarded to the output. At the end of execution, the host CPU retrieves the results by reading the module's memory-mapped registers.

Example: Global SQL aggregations: count, min, max, average

Many-to-One (N:1) Modules that consume 1 or more input data packets, compute and output a single data packet. Such modules are input bound and need to buffer input sequences only if the module is compute-bound.

Example: Group by SQL aggregations: count, min, max, average

Many-to-Many (N:M) Modules with highly unpredictable ratios. The I/O ratios of these modules are data-dependent. Generally, they will need to buffer input packets to avoid data loss in cases of output congestion.

Example: SQL Join

3.2.2 Amount of Tables Addressed by Accelerator Modules

On the contrary of how much data do modules produce or consume, we can sort them by how many tables do they produce or consume. It is not of any significance for this classification whether these tables are the input tables, result tables, intermediate tables stored in volatile memory, or intermediate tables that only exist as such in the form of direct streams between different computations.

Modules that do not produce data naturally work on zero output tables and similarly if they do not consume data they work on zero input tables. These cases are not of interest in the current grouping as they do not pose a large implementation challenge.

One-to-One (1:1) Almost all modules fall into this group. Naturally, even modules that have strictly data input- or output-bound behaviour usually still work on only one table.

Example: aggregations, arithmetic

Two-to-One (2:1) A very prominent special case is the combination of two input tables into one output table.

Example: merge join

One-to-Two (1:2) Another special case can be observed for splitting a table into two. This can be observed in the filter operation. The reason why the filter allows the splitting of the input table in two is that this enables a filter module to act as a selector for the application of further operations, rather than simply a data-omitting processing element.

Example: filter

Many-to-One (M:1) One of the more difficult cases is having a large number of tables that a module operates on. Very wide merging can be observed in large utility merge sorting and can stretch to thousands of tables [61].

Example: merge sort

One-to-Many (1:M) This scenario poses difficulty similar to the previous one. Very wide splitting can be observed in the hashing phase of streamed hash joins and can also reach thousands of hash collision tables. Additionally, it is observed in widely used algorithms such as k-means clustering.

Example: hash join (hashing phase), k-means clustering

As observed in this classification, there are special cases that require careful considerations to handle. More prominently the M:1 and 1:M cases can reach up to thousands of intermediate tables. The interface definition needs to consider all special cases and enable their seamless operation and integration.

3.2.3 Active and Passive Stream Modules

When observing the behaviour of processing elements, a pattern for data consuming and producing emerges. We can quantify operators into two distinct classifications: 1) active modules that do produce and/or consume data streams, and 2) passive modules that do not drive the data stream.

Active Modules

Active modules implement data-dependent functions. These modules need to be aware and keep track of aspects of their corresponding streams of interest and control the smooth dataflow by managing the provided control signals. Many database operators fall in this group: sort, join, aggregate. One key aspect in the implementation of these operators is that they require local buffers to hold a certain amount of input data before their algorithms produce an output. These modules are considered to be the main drivers of the data movement in a streaming region. Another characteristic of active modules is that their operation can be compute-bound, which requires dataflow control between communicating modules to accommodate for the computation speeds.

Passive Modules

These modules execute their operations at the interface speed. Examples include arithmetic, filter, permutation. Designing these operators as passive allows for many resource optimisations. Usually, these modules do not need to be aware of details about the dataflow, rather they simply apply their operation on their stream(s) of interest.

3.3 Proposed Dynamic Stream Processing Interface

The definition of the Dynamic Stream Processing Interface (DSPI) is key to enabling a variety of operators from the database query execution field to be mapped into partial reconfiguration streaming regions. Moreover, the general concepts behind the structure of the DSPI should allow for mapping virtually any other application that is suitable

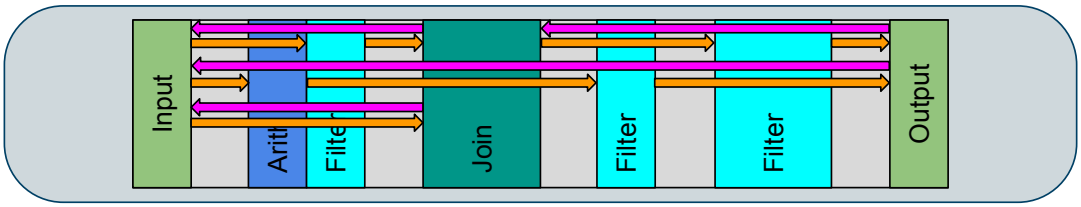

Figure 3.2: DSPI dataflow topology: credit allocation instructions move in the direction output-to-input (right to left), while streamed data moves in the opposite direction (left to right). Purple represents credit allocation links, while orange represents streamed data links.

for stream processing. To solve the problem, many fundamental requirements need to be considered in designing the interface as well as the modules: 1) data rates (see Section 3.2.1), 2) table addressing needs (see Section 3.2.2), 3) general behaviour and concepts of memory subsystems (see Section 3.1.2 and Appendix A.3), and 4) uncomplicated software integration. Ziener et al. proposed and implemented a dynamic interface for database acceleration [135], however, the dataflow management is fully external to the modules, thus it allows only for passive partially reconfigurable modules (see Section 3.2.3). Vesper proposed a concept for such an interface where the partially reconfigurable modules can control the dataflow using stall signals [118]. This allows for active modules to be implemented. However, precise management of stall signals has to be considered in order to avoid data hazards due to a module stalling while incoming data is in flight. Moreover, this approach does not fit all requirements that modules can have (see Section 3.2.2) as it does not support operators that can address many tables (most notably merge sorting).

3.3.1 Dataflow Topology

The proposed interface uses the concept of virtual streams where the same physical wires are used to transfer multiple logical data streams. The proposed interface implements a shared datapath that enables many data streams to be mapped using virtual streams. At runtime, the modules can be programmed to operate only on specific streams while passing through all other data to the succeeding part of the PR region. The dataflow model is similar to a Kahn process network [31]: consuming operators have to report any free space in their input buffers and producing operators listen to these reportings and produce output data accordingly. This is composed using a streaming topology that requires the active modules (see Section 3.2.3) to request data

to the preceding active modules in the system on the same stream. The requesting is done through an *Instruction* (described in Section 3.3.6) sub-interface that is routed in the opposite direction of the main datapath. This behaviour is shown in Figure 3.2, which implements the same example as from Figure 1.2. The request signals propagate unchanged through all modules on the way to their targeted module (which can include the input from DDR).

3.3.2 DSPI Signal Definition

Table 3.1 lists the physical definition of DSPI. A reset signal propagates only in the direction from the *DMA* module towards the other accelerator modules. Data transaction signals propagate from the DMA module towards the module that reverses the direction and back to the DMA module. The four *Instruction* signals propagate the same route and opposite direction as data.

Transactions

The transaction type (depicted as *Type* in Table 3.1) signal is a two-bit bus that implements a 1-hot encoding (for 0-cost decoding) selection between data and control packets. The definition of the bus signals changes based on whether data or control transaction is selected. Additionally, control packets can have two ways of addressing a target module: relative and absolute. Absolute addressing uses the same methods for module identification as normal data transactions. It is used, for example, to mark the end of stream (EOS) whenever all data of a stream has been processed. Additional use is to provide direct memory access to partially reconfigured modules in the stream pipeline (see Section 3.3.5). The challenge is how are modules initialized with parameters after bitstream reconfiguration. Different bitstreams could have different codes hardcoded in order to distinguish the target for control transactions, but this would not work if the same bitstream is used multiple times in a resource elastic fashion. For this reason, DSPI allows relative addressing in control transactions. Relative addressed packets address the modules depending on their physical position in the pipeline (see Section 3.3.4). Independently of the data and control transactions, the system implements *Instruction* transactions in the opposite of the stream's direction (see Section 3.3.6).

Signal name	Width, bits	Example width, bits	Definition & encoding
clk	1		Operational clock
rstn	1		Cascading reset
Data	$X \times 32$	512	Data
Last	1		Indicates last clock cycle of packet transaction
Type	2		Indicates packet type: 00 - Idle 01 - Data transaction 10 - Control transaction
StreamID	Any	4	Enumerates the virtual streams
ChunkID	Any	5	Enumerates the data transfer cycles within a packet
ChannelID	Any	10	Secondary enumeration of streams solving for many-stream problems
State	Any	32	State encoding between modules. Suitable for transferring intermediate states in resource elastic chains.
Instruction Type	3		Indicates control type: 000 - Idle 010 - Request 011 - Skip 101 - Repeat 110 - Restart 111 - Finish
Instruction StreamID	Same as StreamID		Targeted virtual stream
Instruction ChannelID	Same as ChannelID		Targeted virtual channel
Instruction Parameter	Any	16	Parameter of the instruction

Table 3.1: Signal definition of our Dynamic Stream Processing Interface (DSPI). The interface definition and usage examples can also be found at `https://github.com/kmanev/DSPI`.

3.3.3 Data Transactions

Data transactions carry the streams' contents through the operating modules (e.g., stream table records in database systems). When *Type* selects data transaction, there are six signals that encode the data, routing, and state information:

Data : The main datapath that holds the transferred contents is a multiple of 32 bits in order to provide easy integration with software and also with hardware resources (e.g., a Xilinx BRAM36K can be arranged as 1024x32 memory).

Last : A single bit denoting the last clock cycle of a multi-clock-cycle packet transfer.

StreamID : The data bus is shared for all operators placed in the PR slot (orange arrows in Figure 3.2 represent different data virtual streams sharing the datapath). The StreamID is used by modules to identify the currently transferred packet and determine whether they apply their operation to the streamed data (which can also mean consuming the packet). Thus it is used to label: 1) which module consumes the packet, 2) which module produced the packet, 3) what are the data types in the packet, and 4) which modules apply their operation to the data.

ChunkID : Considering applications such as databases, most real-world problems work on records that are significantly larger than any achievable datapath width. Thus, it is needed to utilize multiple clock cycles when streaming such data. ChunkID is used to enumerate the different chunks of data transferred. The modules can use that to selectively apply their operations in different chunks of the transferred record. Additionally, even if modules apply their operations to all chunks in order to increase utility, they can use ChunkID in order to select between various operation parameters.

ChannelID : In most systems, the number of utilized virtual streams will typically be smaller than what StreamID can encode. However, stream operators can have requirements for the data being split into hundreds or thousands of streams (see Section 3.2.2). ChannelID extends StreamID by further providing enumeration for virtual streams/tables. While StreamID defines the type of data, origin, and destination, ChannelID enumerates hundreds or thousands of additional virtual streams/tables that share the same StreamID parameters.

State : This bus is used for sharing custom intermediate data states between composed modules in a resource elastic chain (see Chapter 5).

Algorithm 1 Relative address handling in operating modules

```
1:  if ChannelID != 0 then
2:      ChannelID ← (ChannelID - 1)
3:      Output packet
4:  else
5:      switch Operation do
6:          case Read 32-bit :
7:              Data ← registers[State]
8:              Output packet of type Read 32-bit Response
9:          case Write 32-bit :
10:             registers[State] ← Data
11:         case ...
```

3.3.4 Module Memory-Mapped Registers

Memory mapping of module registers is the most common solution to ease the software-hardware collaboration and integration. The proposed interface enables memory-mapped registers by utilizing *relative* addressing. To mark a control transaction as being relatively addressed, the most significant bit of ChunkID is set HIGH. The rest of ChunkID are used to further identify what is the sub-type of the transaction. Transaction examples include read a 32-bit register, write a 32-bit register, read a 64-bit register, write a 64-bit register. There are twelve more encodings left unused that can be mapped to any other custom relatively-addressed control transaction needed by the target system. In this type of control packets, the Data and StreamID signals can be used to transmit the actual payload.

The addressing is done using the ChannelID and State signals. ChannelID encodes which module is addressed, while State provides the register address within the module. When the DMA module converts a read/write request from the host CPU, ChannelID encodes the position of the target module in the PR slot (0,1,2 etc.). Operating modules then use that as a counter and decrement it every time such a packet passes through until the value becomes zero (see Algorithm 1). In the case of memory-mapped registers read operation, the module returns the data in a read response control packet that has absolute addressing towards the DMA, while write operations require no acknowledgement.

To further optimise modules we omit any multiplexing on the wide datapath and vertical routing in these operations. This is done by replicating the transferred data along the full height of the interface (i.e. for a 32-bit write operation, we set the 512-bit datapath to have 16 copies of the write data). This way only the DMA module

would need to master the full datapath, while all other modules can selectively choose any 32/64-bit integer. Of course, while this solves the issue for register write operation, it would not solve the same problem in read responses. To solve the problem for read responses, we use ChannelID to encode which 32/64-bit integer of the datapath holds the actual data response, and the DMA implements a 32/64-bit 16/8:1 multiplexer. This enables small operators to mitigate any data positioning costs by using any data field on the datapath for both read and write register operations.

3.3.5 Module Direct Memory Access

Direct memory access may be required by many operators that need large amounts of intermediate storage. For example, hash join operators might build hash tables in DDR memory. The proposed interface enables memory-mapped registers by utilizing *absolute* addressing. To mark a control transaction as being absolute addressed, the most significant bit of ChunkID is set to LOW. The rest of ChunkID are used to further identify what is the sub-type of transaction. Example transactions include read 32-bit register response, read 64-bit register response, End of Stream (EOS), 512-bit DMA read, 512-bit DMA write, 512-bit DMA read response. There are ten more encodings left unused that can be mapped to any other custom transactions needed by the target system. In this type of control packets, the Data, StreamID signals can be used to transmit the actual payload.

The direct memory read/write control packets use the Data bus to hold the bulk of the data. The optimisation described in Section 3.3.4 can be reused in direct memory access as well. For read operations, the module marks StreamID that will later be used in the read response packet generated by the DMA module. ChannelID is used for read length for operations that span across multiple cycles (i.e., memory read burst length). For write operations, StreamID is unused. ChannelID is potentially used as an input to the wide data field multiplexers in the DMA module. The length of write transactions is defined by asserting *Last* signal in the last clock cycle of the write data transfer. Additionally, the State signal is used to hold the target memory address.

3.3.6 Dataflow Instructions

DSPI enables systems where active modules (see Section 3.2.3) produce and consume credits based on Kahn Process Networks (KPN) semantics [31]. This credits based network is similar to PCIe protocol [57] and also the AXI protocol [5]. One major

difference over the AXI protocol is the lack of stall mechanisms in DSPI. In order to reduce interface overheads, there is no busy indicator in DSPI. Similarly to KPN, the modules request data only when they have sufficient storage available in input buffers or they have the ability to process the data at line speeds. Since this can have a latency impact compared to a more aggressive approach (like AXI) when accessing external memory, the DMA module needs to provide data prefetching which will eliminate large DDR access latencies. On the other hand, when modules allocate credits to other modules in the PR region, then the latencies are typically in the range of less than ten clock cycles and this poses no data starvation hazard.

The stream instructions are implemented by using only four signals (see Table 3.1). StreamID and ChannelID are used as metadata that enumerates the receiving module of the instruction. The instruction parameter signal provides size to the instructions (i.e. how many times is the instruction executed). It enables the combining of multiple instructions of the same type which greatly reduces the stress on the instruction sub-interface path. DSPI enables five instructions to control stream flow:

Request : This is the most commonly used instruction to control the flow. It allocates credits to preceding modules on a particular virtual stream. Modules that operate at line speeds and have IO data rates that do not increase the size of output data compared to input data (*1:1*, *N:0*, and *N:1* in Section 3.2.1) can allocate credits freely based only on the credits they have (example of such module is aggregation). Other modules would typically need to ensure sufficient free input buffer space before allocating credits. Request instructions use the instruction parameter to announce the number of credits allocated.

Skip : This instruction is used to move ahead in the stream without streaming the data itself. It does not interfere with credit allocation, rather it moves data pointers such that they skip ahead. A stream is a continuous flow of data and a module can choose to skip certain elements and continue processing with the proceeding data. Skip instructions use the instruction parameter to announce the amount of skipped data records.

Repeat : Similarly to the skip instruction, repeat does not conflict with the credit system. This instruction is used to announce the need to restream the last data elements. A module can choose to rewind a certain number of elements and continue processing with the restreamed data. Repeat instructions use the instruction parameter to announce the number of repeated data records.

Restart : This instruction announces the need to repeat the streamed data from the beginning of the data sequence. It does not utilize the instruction parameter. Modules can have the need to operate on a data stream in multiple runs, thus they can announce this instruction to restart the corresponding stream.

Finish : This instruction announces that a module has completed computation on a particular virtual stream and its streaming can be dropped. Stopping the data flow on a particular virtual stream could also be implemented by stopping credits allocation. However, announcing the finish instruction allows for full awareness in other modules of the global state, possibly enabling relevant functionality or optimisations. Similarly to restart, this instruction does not utilize the instruction parameter.

The need for instructions

The implemented instructions in DSPI allow for full flexibility over moving stream pointers and allocation of credits. This allows DSPI to be used for general stream processing applications beyond database acceleration. The database acceleration example showcases the use of the available instructions. Since the *request* instruction is used for credit allocation, all active modules need to implement its usage. The other instructions have more limited usage but are key nevertheless.

The *skip* instruction can be utilized when accelerating queries that have *SQL OFFSET*, which is an SQL command to skip a certain number of records. This SQL command is largely used when organizing query results into multiple pages and the SQL query is re-executed with a different OFFSET value for each requested page.

The *repeat* instruction is key when implementing merge join operator (see Section 4.5.1). Due to the module's limited buffer sizes, the repeat command is used to allow the module to join streams that are highly skewed with large amounts of repeating keys. When the module detects the overwriting of skewed keys in its buffers, it can use the repeat command to reacquire the overwritten keys for joining proceeding records (see Section 4.5.1.

The *restart* instruction is key when implementing the merging phase of hash join operator (see Section 4.5.2). Due to the nature of the operator, it requires to stream the whole temporary table of hash collisions for a certain hash value in order to join them with the other stream. For this purpose, the collision tables would need to be fully streamed multiple times and this requires that they are properly restarted each time.

The *finish* instruction is key when accelerating queries that have *SQL LIMIT*, which is an SQL command to output only a certain number of records. Additionally, modules can announce that they have finished computation on a certain stream. For example, when the merge join module already receives an end-of-stream packet on one of the two merged streams, it can announce on the other stream that data is no longer needed to be streamed.

3.3.7 Evaluation

The interface is a key component that is often overlooked. The optimal design needs to accommodate all possible use cases while maintaining minimal resource requirements. Table 3.2 shows an overview of key features, capacities, and costs of DSPI compared to two other interfaces proposed in the state-of-art related works.

Features

All stream interfaces allow for passive modules as they require no special control. However, active modules have certain requirements to be allowed to control the dataflow. Ziener et al. utilize a PR region that has external dataflow control, which prevents using active modules [135]. Thus this forces modules such as sort, join to be implemented in the FPGA static system leading to overprovisioning or underutilization issues. On the other hand, Vesper proposes flow control using stall signals [118]. This allows for each module to announce its readiness to receive data. However, utilizing stall wires limits the maximal number of possible virtual streams. DSPI mitigates this issue by adopting the widely used credit-based system, providing instrumentation for module-driven flow control. Provided ability for bulk allocation minimizes the stress on the credit system.

Modules require assigning of execution parameters at runtime. Vesper [118] proposes a hardware mechanism to stream in module parameters in a special initialization phase. This feature does enable runtime setting of parameters, but does not allow for result extraction and requires an additional layer of drivers for software integration. On the other hand, DSPI provides a mechanism that allows for memory-mapped registers inside the PR modules. This eases the integration process, allows for register readback, and even allows for register operations during the actual stream processing runtime.

	Ziener et al. [135]	Vesper [118]	DSPI
PR-ready	✔	✔	✔
Passive modules	✔	✔	✔
Active modules	✗	✔	✔
Multiple streams	✗	✔	✔
Direct memory access from modules	✗	✔	✔
Maximum number of streams	—	16	16×1024
Memory mapped registers in modules	✗	✗	✔
Proposed/achieved datapath width, bits	128	256	512
Control overhead, bits	—	99	87
Data-to-control ratio	—	2.6	5.9

Table 3.2: Comparison between proposed DSPI and the proposed interfaces in related work. Active modules can regulate data movement, while passive modules cannot (see Section 3.2.3). Requirements for the number of streams are of high complexity for some operators (see Section 3.2.2).

Wire cost

The proposed stall mechanisms in the state-of-art related work [118] do not scale due to the requirement of one additional wire per additional virtual stream. On the other hand, DSPI has logarithmic wire complexity, thus the proposed example implementation allows for 1024× more virtual streams while maintaining smaller wire overhead. Additionally, the proposed interface can be easily extended with minimal overhead increase. Moreover, due to the optimised nature of the interface and modules, DSPI manages a 2.26× better data-to-control wire ratio.

Logic cost

The stall mechanisms in related work [118] require early stalling to prevent losing data items that are in flight in pipeline stages from source to destination modules. This early stalling will result in the underutilization of the available module buffers. Alternatively, the modules could implement hazard avoidance by providing a banking mechanism of in-flight data whenever the destination module decides to flag the corresponding stall signal. The proposed credit-based system in DSPI omits such stall mechanisms and their associated costs and limitations.

Since the interface will be used in each module, its implementation cost should be minimal. The explicit mitigation of any handshaking protocols in DSPI enables modules to pass data/control packets freely at minimal cost while utilizing fully the provided input buffers of active modules. Thus in DSPI all modules can selectively implement or omit interface functionality and the corresponding logic overhead. The only mandatory implementation is lines 1-3 from Algorithm 1 to allow relative addressing of modules. All other presented control packet requirements and encodings can be implemented as needed. Thus if features such as direct memory access are not used by a module, the module can simply omit all associated logic for producing and consuming the direct memory access control packets. This does not affect the other modules since if an operation is not implemented to be recognised then the module can simply pass through data unchanged to the module's output side.

Another consideration in DSPI is the simplicity of decoding transactions in the FPGA logic. For example, the *type* signal in the interface definition only uses a one-hot encoding that requires only a single LUT input for enabling operation.

3.4 Conclusion

DSPI is an efficient and versatile protocol for dynamic stream processing modules that can be stitched at runtime. It implements a credit system methodology where PR modules allocate tokens, which enables precise dataflow control leading to maximizing buffer usage and efficiency. The interface is adaptive and most functionalities are selectively omitted by every module to minimize overheads, with only one feature being mandatory (see Section 3.3.4). While state-of-art system proposed by Ziener et al. [135] does not support active modules instantiation in the PR region and requires the presence of static accelerators, DSPI utilizes virtual stream encoding and enables all operators to be implemented in the PR region. DSPI also achieves 13% fewer control wires when compared to state-of-art stream processing interface [118]. While minimizing the wire and logic overhead costs of DSPI, it achieves the addressing of 1024× more virtual streams (see Section 3.3.7). It also enables key functionality such as easing software integration when using memory-mapped registers inside the PR modules.

We use DSPI to implement our stitchable accelerator modules (see Chapter 4). This interface allows us to implement a dynamic merge sort module (see Section 4.4), which would not have been possible when using existing interfaces. Moreover, we utilize DSPI to enable resource elastic modules (see Chapter 5). For example, composable filter module utilizes the DSPI *State* bus to pass intermediate data between modules (see Section 5.2). A composable merge sort module utilizes the DSPI *ChannelID* bus in a cascading fashion to merge more sorted sequences at once (see Section 5.3). Additionally, the deliberate avoidance of asynchronous signals in DSPI allows for seamless partial reconfiguration boundary crossing (see Section 6.2).

Chapter 4

Module Library

This chapter introduces the key design factors for implementing scalable streaming accelerators with large utility. To minimize resource requirements and maximize throughput, we implement modules in a thin and tall form factor (see Section 4.1). We also provide efficient implementations with large utility for the key database operators:

- We implement a filter module that utilizes standard Boolean normal form to evaluate Boolean expressions (see Section 4.3).
- We implement a dual-phase sorting strategy of using a linear sort module to generate sorted sequences, followed by one or multiple stages of merge sorting to produce the final sorted result (see Section 4.4).
- We implement a merge join module that can perform many-to-many join operations (see Section 4.5).

Considering their key role in FPGA acceleration, we implement these modules with considerations for resource elasticity, which we describe in Chapter 5.

The proposed system is exemplified on the ZCU102 FOS platform and our custom streaming protocol is adapted to the standard FOS AXI interface through our *DMA module* (see Section 4.2). The system was also designed with consideration of Xilinx Virtex FPGAs of the current and next generations that provide more FPGA resources and higher memory bandwidth through DDR4 memory channels. By utilizing a PR region of two clock regions height and targeting 512-bit datapath and 300 MHz operating frequency, this system provides sufficient high speeds to utilize most FPGA memories and exemplifies the high scalability of the design. The described modules are implemented in our system prototype and evaluated against a TPC-H case study in Chapter 6.

4.1 Design Factors

While FPGAs can greatly accelerate many algorithms, FPGA operation is also associated with overheads. When the FPGA and the host CPU share the same address space, any data movement overheads are omitted. This is, of course, limited to when the target data is already available in RAM. The overheads imposed from external storage have the same impact when accessed for CPU, GPU, or FPGA acceleration. Thus, the overhead of any external data movement is not key in designing accelerators and must be handled rather at a system-wide level.

Compute capacity

FPGAs suffer from configuration overheads, which can render dynamically reconfigurable systems inferior. In order to instantiate an FPGA design to accelerate a problem, the host system needs to ensure that the provided FPGA benefits will overcome the associated overheads. Ideal targets for FPGA acceleration consist of operations that will result in a compute-bound software execution. In such cases, modules with low computational capabilities tend to need multiple instances in order to satisfy the larger problem requirements, thus increasing the overall resource cost. We anticipate that it is beneficial to provide module designs that enable more complex behaviour (e.g., filter module that can compare multiple fields in parallel, rather than comparing only a single field).

Module scalability

One of the key benefits of FPGA acceleration is the ability to implement dataflow accelerators with deep, complex, and customizable processing pipelines. However, for achieving FPGA acceleration, we also need to provide high streaming throughput. Accomplishing a high throughput requires both high operational frequency and a wide datapath. Designs, even in modern FPGAs, achieve about 100-300 MHz, which is a low frequency compared to modern CPUs' 4-5 GHz. To compensate for the relatively low frequencies, the designs need to implement a wide datapath (several hundred to thousands of wires) to show performance improvements. Often designing scalable modules can be non-trivial and its achievability varies depending on the target algorithm and the design approach. We target the implementation of scalable modules, meaning that our modules are able to operate on such wide datapaths, while also meeting our high operating frequency target (300MHz in our prototype system).

Utility

Many large problems require several runs through the FPGA chip (e.g., sorting). Completely independently of throughput, the total execution time scales linearly with the number of data runs through the chip. Additionally, often external throughput is a limiting factor for FPGA acceleration. Therefore minimizing the number of runs is of utter importance to increasing the effective speedup. Stitching more operators in one run will benefit the minimization of the total number of runs, however, that is limited by the amount of available resources. On the other hand, increasing the amount of compute capacity per amount of resource requirements will allow for higher compute per FPGA run and minimize the total number of runs. In this thesis, we define *utility* as the amount of useful compute accomplished per unit I/O. Therefore, utility is a key measurement for designing and evaluating modules and their suitability for acceleration.

4.1.1 FPGA Resources

Understanding the organization of the FPGA resources and the resource footprint of the synthesized PR modules is beneficial to achieving high scalability and utility. As described in Section 2.1.1, the FPGA fabric provides diversity in available on-chip compute and memory resources. However, this diversity is highly organized (see Figure 2.1) as all resources within a column are equal. In fact, the fabric of Xilinx Zynq UltraScale+ FPGAs is organized into double-columns that consist of two columns (one CLB column and one CLB, BlockRAM, or DSP column) with an interconnect column inbetween (see ①⑤②, ④⑤③ etc. in Figure 2.1). Double-columns provide a more generalized view of the FPGA resources. Additionally, the switch matrices in Xilinx FPGA interconnect provide a general routing architecture with identical wires to other switch matrices, independently from the type of the surrounding FPGA resources. Using these regular identical wires is suitable for straightforward wire interface definition for PR modules.

All compute and memory FPGA resources of the ZCU102 can be described by three types of double columns: 1) LUTM+LUT (M), 2) LUTM+DSP (D), and 3) BlockRAM+LUT (B). Additionally, there are specialized columns that implement I/O pins (I/O) but considering their position and functionality, they are not of interest to the stream processing system. Figure 4.1 visualizes the available double-column resources in the ZCU102 board (featuring a Xilinx ZU9EG FPGA). The targeted PR slot by the

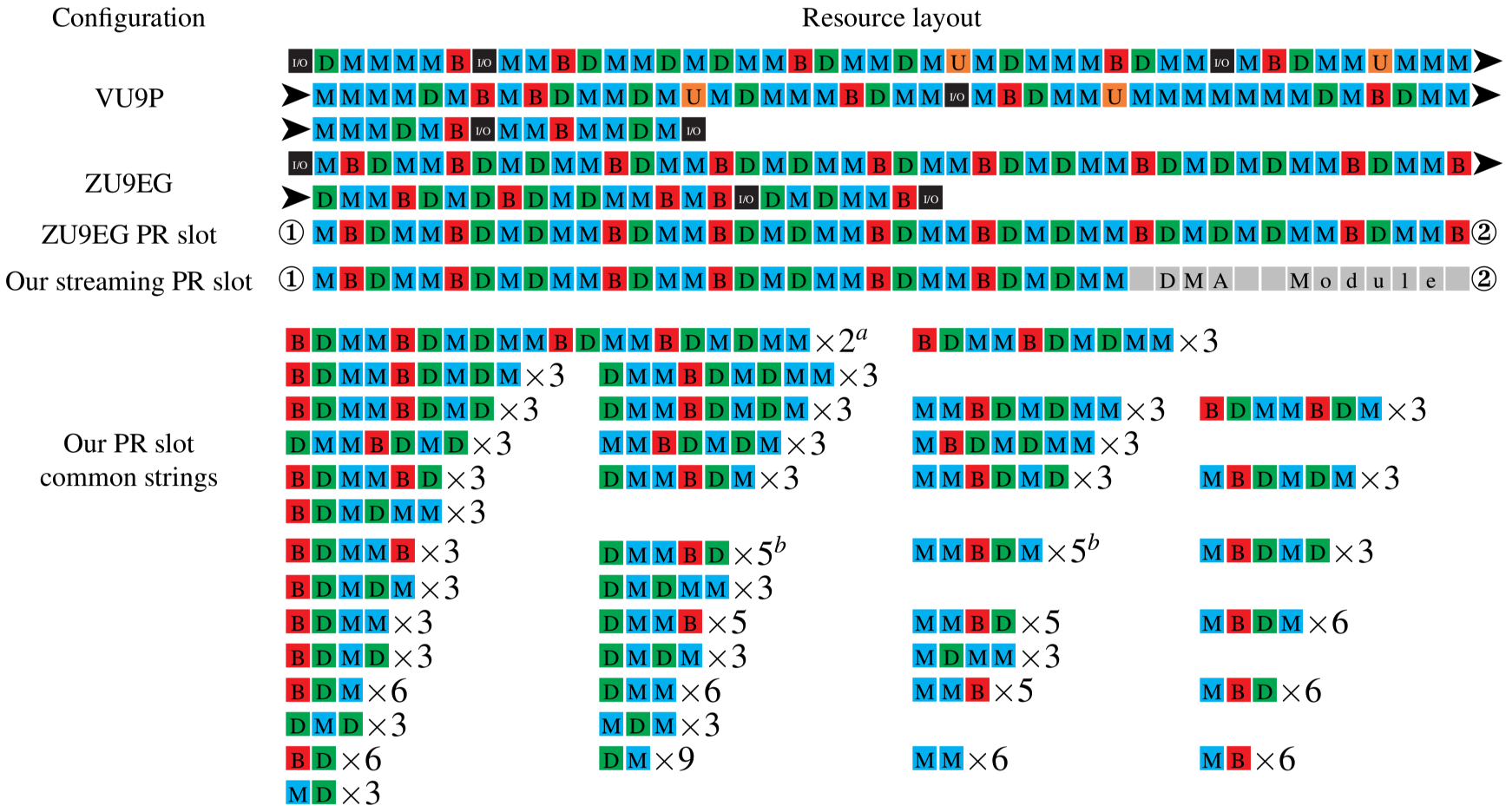

Figure 4.1: Column types: 1) I/O – I/O, 2) M – LUT+LUTM, 3) B – BlockRAM+LUT, 4) D – DSP+LUTM, and 5) U – UltraRAM. ZCU102 PR slot sides: ① — edge of PR slot (reversing stream direction), ② — standard FOS PR interface. Note, for simplicity all presented strings are substrings of BDMMBDMDMM. Other unique substrings in the PR slot are also available.

[a] BDMMBDMDMMBDMMBDMDMM can be placed in 2 positions that overlap each other.

[b] DMMBD and MMBDM can be placed in 5 positions some of which overlap.

database system consists of a DMA module that is located next to the standard FOS PR interface. This module provides interface translation between the AXI master and AXI slave ports that the FOS system implements and the custom DSPI interface on the other side where the actual accelerators are to be placed. The accelerator part of the PR slot implements three equal large resource footprint blocks (BDMMBDMDMM) with an additional column at the end that can be used for reversing the stream direction.

4.1.2 Shape, Resources, Positions for Proposed Modules

Figure 4.1 outlines module resource targets in the form of strings to be mapped as a string matching problem for module placement planning [34]. Resource targets have various possibilities for placement inside the PR region. Thus we study also how many place positions does each target module footprint have. Most available resources in our target device and PR slot map in three different placement locations, due to being a unique substring in the three large repeating patterns (BDMMBDMDMM). This regular pattern, however, does not necessarily correspond to other devices of the same family (like Xilinx VU9P) or future FPGA fabrics, thus it is important to generate resource footprint graphs for the target family and study the available module footprints and their placement options. Xilinx VU9P FPGA is widely utilized in datacenters (e.g., in Amazon AWS F1 instances [2]). VU9P devices provide more diverse and targeted resource footprints. For example, the largest resource footprint with no dedicated blockRAMs on ZCU102 is DMDMM, while on VU9P it is MMMMMMMDM . Future prototypes of certain large modules, such as our filter module (see Section 4.3), on datacentre FPGAs will not underutilize the BlockRAM resources (reserve BlockRAM resources without utilizing them for compute).

Resources

The available resources of the proposed slim modules are rather scarce, especially when compared to the total amount of resources available in the large modern FPGAs. Table 4.1 shows the primitives available in the three available basic double columns. The resources available are scarce when taking into account the width of the targeted datapath. Providing only a single pipeline stage on both stream directions requires 1,198 Flip-Flops, which is a considerable amount.

Pattern	Positions	LUT	LUTM	Flip-Flop	BRAM	DSP
M	15	1,920	960	3,840	0	0
D	9	960	0	1,920	24	0
B	6	960	960	1,920	0	48
BD	6	1,920	960	3,840	24	48
DM	9	2,880	1,920	5,760	0	48
MM	6	3,840	1,920	7,680	0	0
MB	6	2,880	960	5,760	24	0
MD	3	2,880	1,920	5,760	0	48
BDMMBDMDMM	3	14,400	7,680	28,800	48	144

Table 4.1: Available FPGA resources in common module layouts.

Module shape

Ideally, small operators have a small resource impact. While data streams horizontally through the two clock regions height of the targeted PR slot, the width of the modules is considerably smaller than their height. Figure 4.2 illustrates the available module shapes depending on the resources utilized. That figure preserves the visual ratios of the resource representation as shown in Xilinx Vivado. The clear pattern of slim tall modules restricts the possibilities of vertical data movement in the modules with a smaller resource footprint. This requires additional consideration when implementing modules that target small resource impacts. State-of-art work implemented simple modules (such as arithmetic, comparison, boolean evaluation) using vertical multiplexers for both the input fields and the output field [135]. This design was feasible for the targeted system, which consists of only four 32-bit data fields in the PR region as their multiplexing was more relaxed. However, the design does not scale well and vertical multiplexing has to be avoided as much as possible when targeting wide datapaths.

4.1.3 Module Resource Footprint Variants

Building the PR region using presynthesized modules also results in a packing problem. A bad solution to this packing problem results in a fragmented PR region that is underutilized because of resources left unused between modules.

There exist multiple sources for module placement constraints. Depending on the executed subquery, the most notable constraint is for preserving the order of chained operations. However, in cases of multiple stream chains (e.g., two streams operated by chains of modules before a join operator) the module placement can be more relaxed as

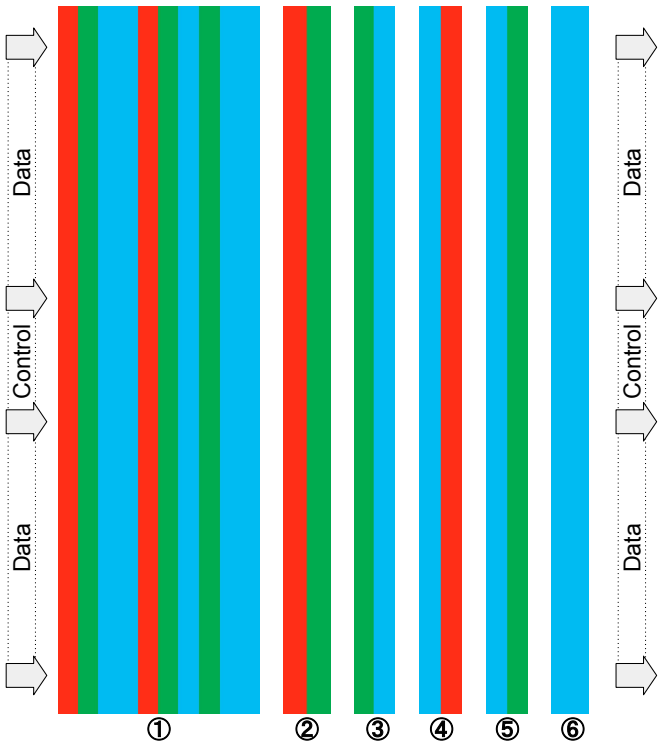

Figure 4.2: Resource shapes of common strings in their original aspect ratio in Xilinx Vivado device view: ① - BDMMBDMDMM, ② - BD, ③ - DM, ④ - MB, ⑤ - MD, ⑥ - MM. Modules are thin and high. Data streams are horizontal. The vertical movement of data should be minimized to prevent congestion of vertical routing resources.

there are no ordering constraints between the modules from the different stream chains. Since most resource strings can have rather limited placement positions, in order to enable optimized module packing, the system might require the implementation of modules with the same utility into multiple different resource strings (i.e. physically implemented modules offering the same function but with a different resource column layout). Increasing the implemented number of resource strings that are targeted for a module configuration, increases the number of possible module placement positions. This can lead to a reduction in fragmentation and ultimately to elimination of data runs through the FPGA (if more FPGA resources can contribute to an acceleration task) which effectively increases performance.

The optimal choice of resource footprint variants of a module is a process that is dependent on factors such as the module resource requirements. We can utilize a Pareto-Front approach by identifying the nonoptimal module footprints. If a module can be placed and routed for a certain resource placement string **X**, all possible resource placements where **X** is a substring, are not on the Pareto-Front and can be omitted.

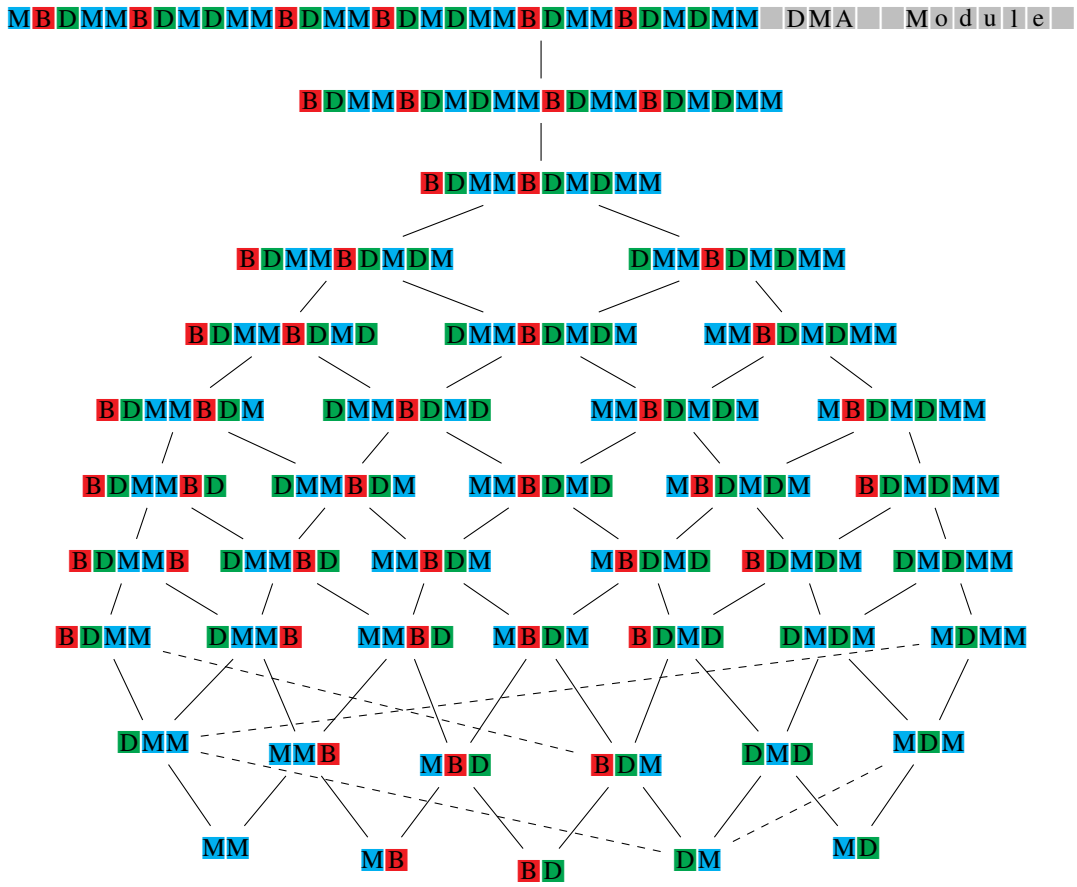

Figure 4.3: Resource graph in targeted ZU9G PR region.

This can be looked up in a graph that implements a resource tree of the available substrings in a PR region. When finding a successful placement, all ancestor nodes in the tree are suboptimal. Implementing all Pareto-Front placements of a module can be achieved with Algorithm 2. Figure 4.3 implements the resource tree of the ZU9G PR region (ZCU102) that visualises the resource substrings of module footprints. The same tree would have to be created when targeting PR regions in other FPGA devices.

4.1.4 Designing for Optimal Operators

Most compute modules in such systems have little resource requirements relative to the capabilities of modern FPGAs and implement into slim module footprints. Ziener et al. propose arithmetic, compare, aggregate, and boolean modules that require only two double columns to place [135]. This is achieved due to the narrow datapath of only four integers, where the ALU is easily accessed by all four integers. Naive or cascaded

Algorithm 2 Function to search all Pareto-Front placement positions (**R**) for a module configuration **S**

```
1:  function Place (S)
2:      R ← NULL
3:      Sort all resource strings by length
4:      Push all resource strings in Q
5:      while(not Q.empty())
6:          X ← Q.pop_front()
7:          B ← Place&Route(S,X)
8:          if B placed successfully then
9:              R.push(B)
10:             foreach Y in Q
11:                 if Y is ancestor of X then
12:                     Q.pop(Y)
13:     return R
```

multiplexer implementations (see Figure 4.4 a-b) are feasible with such limited datapath width. However, as the datapath width scales up to achieve higher throughput, the naive design results in vertical wiring congestion which requires making the module wider (and underutilizing logic resources) to accommodate the routing. Cascading the multiplexing of the input fields results in a scalable solution. However, it still results in an increased footprint when increasing the datapath width until it saturates.

Distributed computing modules

Although the small module footprints' lack of vertical data movement results in a lack of flexibility, the number of available FPGA primitives is still in the range of thousands. A module with a resource footprint MM still provides 3.8K LUTs and 7.6K FFs, while many 32-bit operations require only tens of LUTs to implement. Thus implementing a single 32-bit operation in a module using two double columns results in only 1% of the resources to be used for the targeted compute functionality.

We anticipate that the use of modules implementing distributed computing along the wide datapath is well suited for reconfigurable stream processing. Figure 4.4 c-d visualise the concept of distributing the compute logic inside a slim module. Adopting this methodology results in multiple benefits:

No vertical wiring : This approach moves the compute to the data, which removes the need for vertical wiring. As already described and visualized, this leads to a decreased module resource footprint.

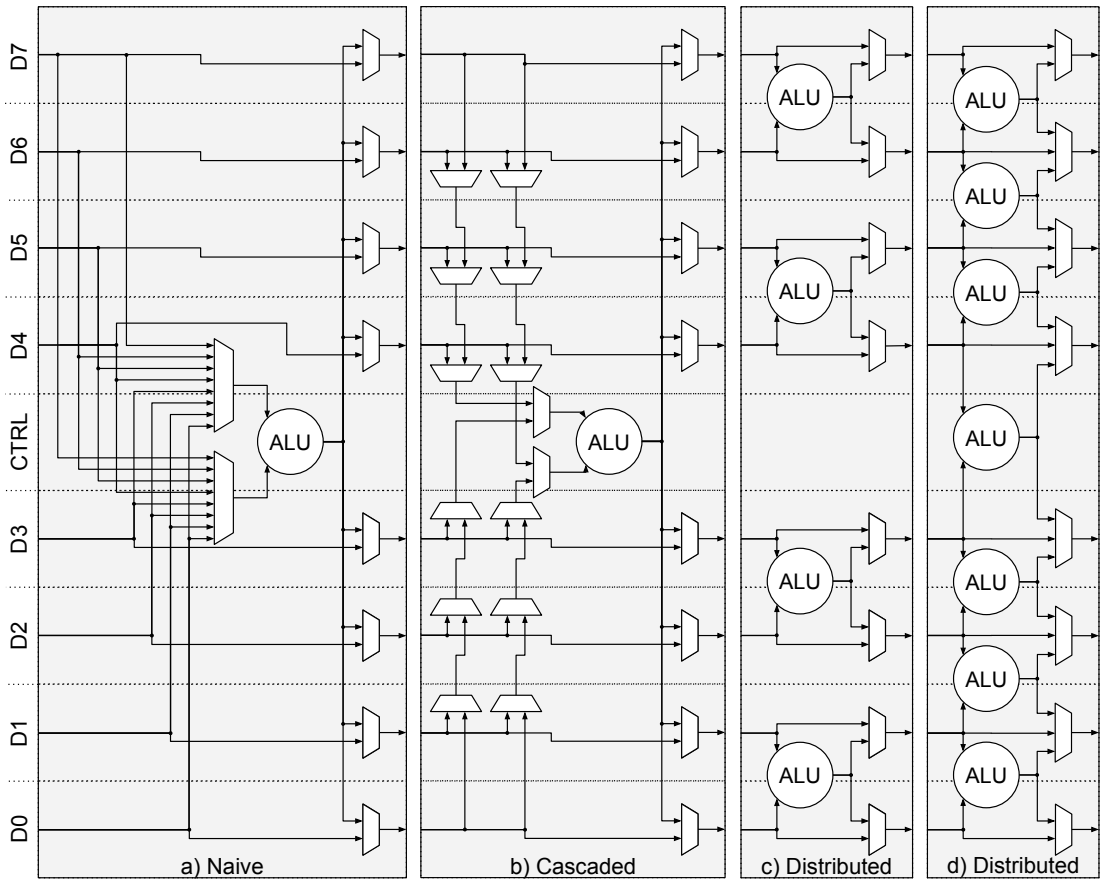

Figure 4.4: Examples for alternatives for designing compute modules with datapath width of 8 integers: a) naive: unscalable due to high vertical wiring, b) cascaded multiplexing: medium vertical wiring, c-d) distributed computing: no vertical wiring.

Increased utility : By replicating the ALU, the module enables multiple data fields to receive operation simultaneously and thus increasing the module's utility. Replicating the ALU has no negative resource impact for most operations. For example, considering 16 32-bit data fields (for 512-bit datapath) and 15 addition operators of 32 LUTs each results in only 12.5 % LUT utilisation for the compute and 13.5 % LUT utilisation for the output multiplexing for a total 26% utilisation for a MM resource footprint.

Additionally, we can decrease the module library size. Implementing a module for every possible arithmetic, compare, and boolean operation results in a major increase in the number of bitstreams in the module library. This is worsened when we anticipate the module place and route for multiple resource footprints (see Section 4.1.3).

However, due to the small nature of the core compute elements, packing them in multi-functional ALUs is also feasible. This has also the benefit of further increasing the utility by allowing simultaneously applying different operations on the different fields. Xilinx DSP blocks implement ALUs with addition, subtraction, multiplication, and logical operators where the operation can be selected at runtime [7]. Furthermore, Xilinx DSP blocks are spread along the vertical direction of the resource column, thus all data fields have access to different DSP blocks with no vertical routing. This allows the implementation of an arithmetic module targeting footprints with DSP columns (D) to achieve efficient, programmable operation with large utility.

The methodologies for distributing the core computation over the datapath do not apply for complex algorithms (e.g., sort, join) or complex computations (e.g., division). These special cases should be handled by proprietary modules where corresponding optimisations and design concepts are to be handled on a per-module basis.

4.2 DMA Module

While the proposed system requires support from a custom dynamic interface (see Chapter 3), it also aims to utilize standard slots in multi-tenant systems (e.g., FOS). This is achieved using our DMA module (see Figure 4.5). It is the first-most placed module, communicates with the external infrastructure, and provides abstractions of multiple details:

Addressing : Streaming operators require the abstraction of data positioning in main memory or within mass storage devices. To do this, the DMA module needs to convert credit allocation commands (see Section 3.3.6) into generated stream packets onto the PR datapath.

Stream states : The DMA abstracts the state of the tables in memory. Such an example is keeping track of the remaining records in a stream and issuing correctly control packets such as End-of-Stream (EoS). Additionally, DMA implements the DSPI-defined instructions for controlling the data streaming such as the *Finish* command which causes an immediate EoS.

Memory mapped registers : DSPI supports relative addressing to enable memory mapping of registers located inside the PR modules (see Section 3.3.4) and the DMA module needs to implement the translation of AXI accesses on its slave

port to such relative addressed stream packets. What is more, the module provides cascaded multiplexing for 32/64-bit read response data incoming from the PR region as well as multicasting of the write data.

Direct memory access : Since DSPI also allows modules to selectively request direct access to external memory-mapped memories, the DMA module needs to handle and track those as well to satisfy the requirements of the master AXI protocol. Eventually, if narrow DMA operations are implemented, they can reuse the same multiplexing that is provided for the memory-mapped register operations.

4.2.1 Data ordering

Additionally to abstractions, the DMA provides important data organisation functions. The utilized accelerators can have a wide range of data positioning requirements:

Data positioning : Modules implementing certain complex operators such as sort/join would often require specific positioning of keys within the wide datapath to omit global routing thus providing more optimal resource utilization. However, chances are that the incoming data from off-chip memory does not have the data positioned as required, thus the data needs to be reordered.

Data duplication : Certain modules might require data duplication. For example, if a query has a complex filter requirement that implements multiple data compares over a certain field, then this field might need to be duplicated to accommodate for distributed compare PEs along the datapath. Additionally, data might need to be duplicated in cases where it will be used and overwritten (e.g., by an arithmetic module).

Data projection : Often queries will be targeting only a small subset of the fields of a table. To remove the overheads of moving excess data that is unused, it needs to be omitted prior to 1) entering the PR region, or 2) writing back of intermediate tables in DDR.

Vesper proposes the use of a Benes network to solve this problem [118, 14, 103] as it achieves a significant reduction in vertical wiring compared to fully populated crossbars. However, while this approach provides for perfect data reordering, it is not clear whether it can satisfy data duplication requirements together with data repositioning in time and space. Also, it imposes extra difficulty for software to generate routing

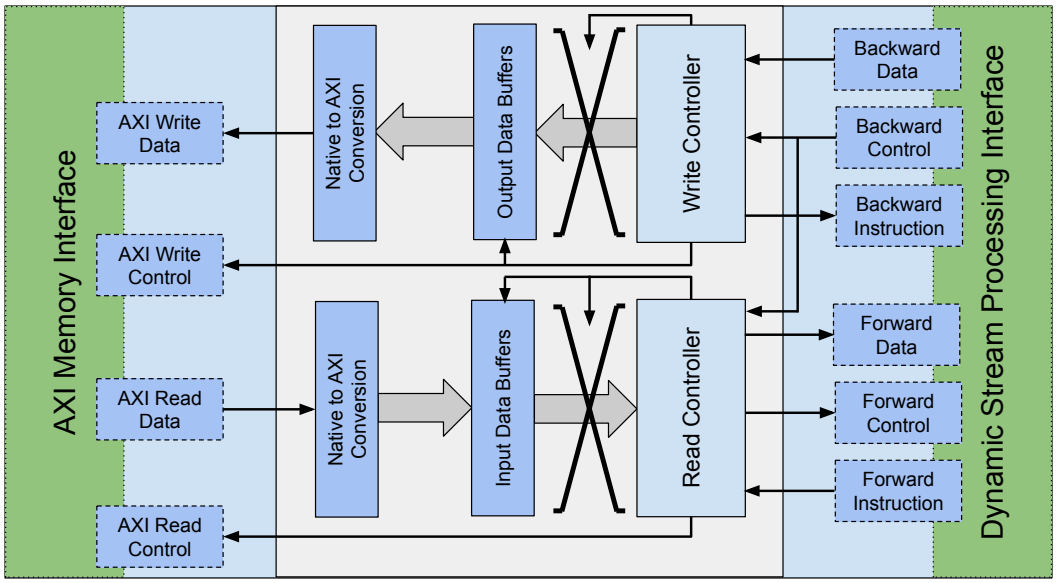

Figure 4.5: DMA module converts between AXI and DSPI protocols and provides the abstraction of physical data to the accelerator modules. Multiplexing in space is achieved in both input and output directions through a crossbar. Multiplexing in time is implemented through fine-grained runtime programmable pointers for reads/writes to the data buffers. Note: this is a logical representation and physically the data busses spread throughout the full vertical height of the module.

parameters. On the other hand, Ziener et al. implement two fully populated crossbars on both sides of a BlockRAM column [135]. This approach requires larger vertical routing overhead but provides full flexibility of data reordering and duplication. At the moment, the proposed DMA module implements a single fully populated crossbar and a single BlockRAM column to multiplex in space and time for both read and write directions (see Figure 4.5). This solution does provide for all possible data reordering. It also satisfies all possible data reorderings when there is also duplication requirement, but sometimes requires additional chunks in the streamed record in the PR region, which can have a negative performance impact in certain corner cases. One major advantage is the ease at which the crossbar routing (see Figure 4.5) is programmed by the software. Each 32-bit data word is abstracted as a network packet with programmable buffer read offset and packet routing destination.

The crossbars hold their route data into BlockRAMs to also enable distinct multiplexing information for each supported stream and chunk ($16 \times 32 = 512$ unique crossbar configurations in the proposed example).

Data prefetching

Modern DDR technologies provide relatively low access latencies (see Section A.3), however, they still pose a performance hazard if not handled correctly. As already described, the DMA module utilizes BlockRAMs to enable time multiplexing (see Figure 4.5). It also uses these BlockRAMs to fire multiple DDR read operations simultaneously. This results in the filling of the pipelines between the DMA module and the DDR memory controller, thus fully utilizing the capabilities of the controller without creating idle cycles. Additionally, since the system implements a streaming application where the forthcoming data is at known positions, the DMA module also provides and handles data prefetching. This majorly minimizes the serve latencies for data requests from compute modules in the PR region.

4.3 Filter

Despite restriction being obligatory operation in virtually any database system, its acceleration using FPGAs is seldomly examined. This work proposes an efficient solution for database restriction that provides high utility and scalability by utilizing Disjunctive Normal Form (DNF) solvers for Boolean expression evaluation [92, 64]. The proposed filter is also resource elastic (see Section 5.2).

4.3.1 Compare Operations

A stream processing module that applies restriction would need to compare or match the data elements to a set of pre-initialized reference values. Arithmetic compare requires the support of six main operations: $c \in \{<, \leq, =, \neq, \geq, >\}$. Previous works put forward modules that each used a single hardwired operation to execute. Although this suits well a low throughput system as proposed in [135], it would be impractical for application in our system, as we aim for larger datapath sizes and provide optional computation on every 32-bit data element in a record. We propose and implement two hardwired compares (see Figure 4.6 b) and use their results to evaluate a pre-initialized selection of compare operations using Algorithm 3.

We propose the use of distributed memory (using LUTM) to hold the reference values as well as the operations to be performed for each data element in each functional ID position (see Figure 4.6 a). Additionally, the module can implement multiple compare operations for every data element by replicating the compare elements.

Algorithm 3 Derive Boolean result of a 32-bit compare: X {OP} RefVal

1: $bLT \leftarrow (X < RefVal)$
2: $bEQ \leftarrow (X = RefVal)$
3: **switch** $Operation$ **do**
4: **case** $<$: $bRes \leftarrow bLT \wedge \neg bEQ$
5: **case** $\leq$: $bRes \leftarrow bLT \vee bEQ$
6: **case** $=$: $bRes \leftarrow bEQ$
7: **case** $\neq$: $bRes \leftarrow \neg bEQ$
8: **case** $\geq$: $bRes \leftarrow \neg bLT$
9: **case** $>$: $bRes \leftarrow \neg bLT \wedge \neg bEQ$

4.3.2 Boolean Expression Evaluation

The compare operations produce a true or false response, but in the cases of complex WHERE expressions using multiple compares, we have to evaluate multiple true/false responses through a Boolean expression. Different methods for Boolean evaluation in restriction operations have been proposed, such as using a look-up-table [121, 58, 105], a hardwired Boolean operator tree with programmable nodes [104], or providing dedicated modules for the Boolean operators [135]. These approaches are not designed to deal with many input literals (complex Boolean expressions) and/or literals that are evaluated over multiple clock cycles (as needed for larger records). For example, using big LUTs for evaluation is limited as it requires 2^N bits to store the look-up table and a method for dividing the problem into multiple smaller LUTs has not been discussed in [121]. A Boolean programmable tree of operations limits the flexibility of the enabled queries to be accelerated because it is unable to process Boolean results that are generated in different clock cycles (from data in different chunks of a record) [104].

As a solution to this problem, we propose a design that adopts Disjunctive Normal Form (DNF), as a representation of the Boolean expression [92]. DNF comprises clauses that are aggregated with an OR operation where each clause consists of AND-ed positive or negative propositional variables (literals) that result from the undertaken compare and match operations.

A literal in a clause has three possible programmable states for each of the clauses: positive literal, negative literal, or not existing in this clause. Thus this requires parameter storage of two bits per functional ID to encode the usage of a literal in a clause and then logic to evaluate the result from a compare PE. We propose the use of a single LUTM as a look-up table with inputs being the function selector (5 bits) and the resulting bit from the compare PE, and producing a 1-bit result that states whether the

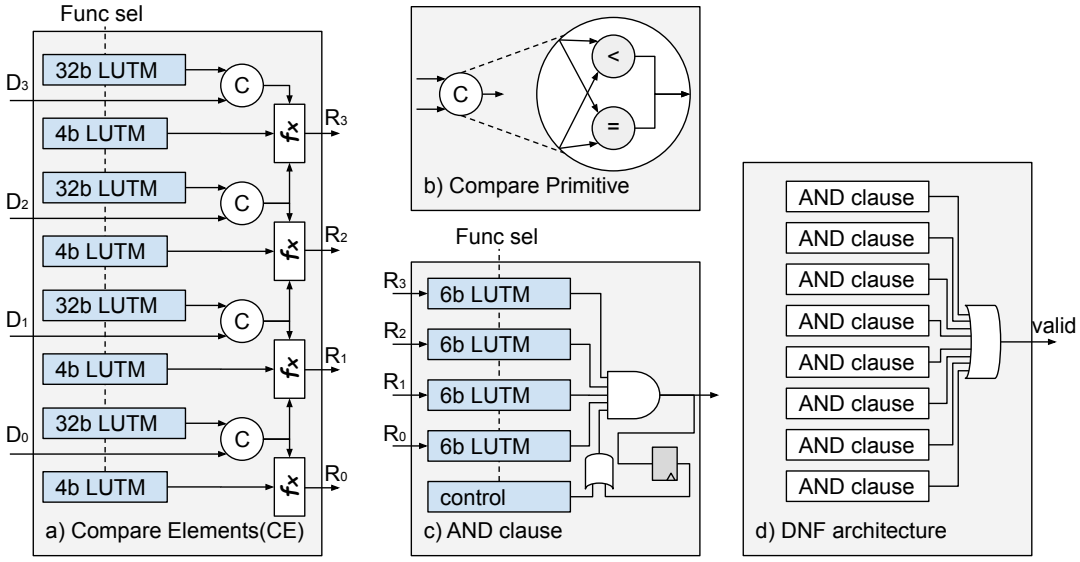

Figure 4.6: Architecture of the proposed module for 4 32-bit data elements and 8 DNF clauses. The design utilizes FPGA LUTMs (see Section 2.1.1) to hold reference data.

particular literal satisfies the particular clause as shown in Figure 4.6 c. These literals are then evaluated using static AND and OR trees as shown in Figure 4.6 c-d. When the literal does not exist in the particular DNF clause, it is initialized to produce a Boolean result of 1 for both result states from the compare. With these optimizations, we shrink the logic utilization of the DNF structure to requiring only 1 LUTM per literal and an optimized hardwired Boolean reduction tree.

4.3.3 Support for Large Data Types

The here proposed system is aimed at computing 32-bit data elements, but large databases in the big-data era have to compute with a large amount of 64-bit data variables as well (e.g., keys in large relational databases). Therefore we also propose native support for 64-bit data restriction operations by utilizing the 32-bit compares and passing their results to the corresponding operation evaluators of their neighbour data element where the most significant 32-bit value of the 64-bit variable is located. With this, we support 64-bit compares without increasing the amount of data computation compared to the 32-bit alternative, by introducing only slightly more complex Boolean logic to evaluate such 64-bit compares. Algorithm 4 shows the logic integrating both 32 and 64-bit support.

To support larger data types than 64 bits, we split the corresponding data types

Algorithm 4 Derive Boolean result of a 64-bit compare: X {OP} RefVal

1: $HbLT \leftarrow (X_{Higher} < RefVal_{Higher})$
2: $HbEQ \leftarrow (X_{Higher} = RefVal_{Higher})$
3: $LbLT \leftarrow (X_{Lower} < RefVal_{Lower})$
4: $LbEQ \leftarrow (X_{Lower} = RefVal_{Lower})$
5: **if** 32*bit Operation* **then**
6: **switch** *Operation* **do**
7: **case** $<$: $bRes \leftarrow HbLT \wedge \neg HbEQ$
8: **case** $\leq$: $bRes \leftarrow HbLT \vee HbEQ$
9: **case** $=$: $bRes \leftarrow HbEQ$
10: **case** $\neq$: $bRes \leftarrow \neg HbEQ$
11: **case** $\geq$: $bRes \leftarrow \neg HbLT$
12: **case** $>$: $bRes \leftarrow \neg HbLT \wedge \neg HbEQ$
13: **else** if 64*bit Operation* then
14: **switch** *Operation* **do**
15: **case** $<$: $bRes \leftarrow HbLT \vee (HbEQ \wedge LSbLT)$
16: **case** $\leq$: $bRes \leftarrow HbLT \vee (HbEQ \wedge (LbLT \vee LbEQ))$
17: **case** $=$: $bRes \leftarrow HbEQ \wedge LbEQ$
18: **case** $\neq$: $bRes \leftarrow \neg HbEQ \vee \neg LbEQ$
19: **case** $\geq$: $bRes \leftarrow HbEQ?(\neg LbLT) : (\neg HbLT)$
20: **case** $>$: $bRes \leftarrow HbEQ?(\neg LbLT \wedge \neg LbEQ) : (\neg HbLT)$

into multiple 64-bit data types before the generation of the DNF clauses and incorporating the relation of the sub-types into the logical expression. For example, 128-bit $X < Y$ can be implemented by using the 64-bit higher(H) and lower(L) parts of the values similar to the support of 64-bit values from Algorithm 4: $X < Y = (X\_H < Y\_H) \vee ((X\_H == Y\_H) \wedge (X\_L < Y\_L))$. Strings can be of very large sizes, but most operations would be comparing for an exact match, thus utilizing the $==$ operator, which implements a logical AND of the compared subparts (Line 17 in Algorithm 4). Consequently, AND Boolean operations result in a single DNF clause, and hence do not increase the complexity of the DNF logical evaluation.

4.3.4 Evaluation

Synthesizing the module to support 32 DNF clauses and 4 compare PEs results in a module footprint (see Section 4.1) of BDMMBDMDMM . This is the largest module configuration, providing sufficient filter capacity to execute the filtering requirements in all TPC-H queries. The filter also achieves resource elastic module alternatives that require significantly smaller resource footprints as described in Section 5.2.

Resources

Distributed memory, which is used to hold initializing data for the compares and the DNF propositional variables Boolean evaluation, requires most resources. Since in the filter design we instantiate directly the LUTMs that told reference data, we can model precisely the LUTM requirement for a specific filter configuration. For a module working on N 32-bit data elements, with J compare PEs per data element, and handling K DNF clauses, the LUTM cost scales as follows:

$$LUTM = InfrastructureOverhead + N \times J \times (K + 16 + 2)$$

where $(K+16+2)$ is a result of the K clause solvers, 16 LUTMs for compare reference values and 2 LUTMs for compare types. As expected, the *module utilization scales linearly with datapath width*.

Throughput

The module synthesizes successfully for the target of 512-bit datapath and 300 MHz operation frequency. The wide datapath showcases the design's ability to scale. This leads to an effective streaming throughput of 19.2 GB/s. The module sustains *one processed chunk per clock cycle*. Considering that any off-chip memory will be required to support this throughput for both read and write operations, this leads to aggregated I/O requirement of 38.4 GB/s. This throughput is sufficient to fully utilize most modern FPGA systems. In cases of higher throughput targets, increasing the datapath width will result in a linear increase in throughput. Moreover, the minimized critical path allows for the introduction of additional pipeline stages in most computational submodules, thus allowing for an increased frequency target, which also scales linearly with throughput.

Utility

The more operations executed per module the better the utility for the given module (see Section 4.1). The proposed filter module can be placed alone to execute the restriction and Boolean evaluation operations in all queries from the TPC-H benchmark including Q19 [108, 68]. Q19 is the most complex in terms of compare operations and Boolean evaluation. It implements 28 different compare operations, including a text

field that is compared to 12 different reference values. The approach of implementing a single field operation per module (as presented in state-of-art work [135]) can implement only a single 32-bit compare or 2:1 Boolean reduction operations per module. This leads to the requirement of at least 74 compare modules to implement the 28 integer and string compare operations (some string comparisons require multiple comparator modules). Using the 2:1 Boolean reduction modules, the system would need at least 73 modules to gather the results of the 74 compare elements. This requires at least 147 modules in total, each of which utilizes two CLB columns (1,280 6-bit LUTs). In total, 188,160 6-bit LUTs will be needed to place all modules to implement that filter, which most probably will result in the need for multiple major data runs through the chip. Since the number of data runs scales linearly with runtime, this significantly reduces effective performance. When utilizing our large utility approach, we can place a single large filter module that accelerates the full query filtering. Our largest module uses 7,675 6-bit LUTs (24.5× less than related work) in practice and its module resource footprint (BDMMBDMDMM) reserves 14,400 6-bit LUTs (13.06× less than related work [135]). Other than the 13× improvement in resource cost, our module implements the filter operation with only one pass of the data through the chip, maximizing the effective performance. By all means, a module reserving 14,400 LUTs is relatively large and not all queries have complex filter requirements, thus we also synthesize smaller filter modules with smaller utility and resource cost (see Section 5.2).

The Boolean expression of Q19 simplifies to 24 DNF clauses. A single module currently supports up to 4 compares per data element and up to 32 DNF clauses evaluation. The capacity of 32 clauses is sufficient to implement Q19, but the issue occurs from the 12 consecutive compares with different reference values over one column (P_CONTAINER). This can be solved by utilizing the duplication capabilities of the DMA module (see Section 4.2) and replicating that particular field by placing it two more times in the streamed packets. This results in the field being present a total of 3 times in the streamed packets and the filter module supports up to 4 unique compares on each replica, thus achieving the total 12 required compares for the Q19 filter. However, replicating the field can incur a loss in effective throughput. Additionally, adapting DNF solvers is naturally suitable for resource elastic integration. Using the module in resource elastic fashion enables implementing Q19 without data replication required. This is described in Section 5.2.

4.4 Sorting

Sorting is one of the most widely studied and applied algorithms in the history of computers [51]. The large attention on the sorting problem has led to many works even including works that try to achieve the slowest sorting speed [18]. Naturally, sorting has also been targeted for hardware acceleration for more than half a century [10, 9, 78, 76].

4.4.1 Sorting for Database Acceleration

Sorting is a rather costly operation. Database systems in software deploy optimizations as often as possible to avoid sorting. For example, software-based database query executors tend to avoid the usage of merge-join and merge-group-by operations by utilizing hash-based approaches. The hash implementations are suitable for CPU execution due to multiple factors:

Caches : The caches in modern CPUs are optimized to hide large latencies from semi-random accesses such as those when looking up hash collisions.

Large control-flow throughput : The high operational frequency (several GHz) of CPUs enables them to work with rather small data types at a time while still evaluating to a significant throughput achieved. This also enables CPUs to benefit from tailored optimizations utilizing their flexibility (e.g., quick address evaluation and access).

Multicore : Whilst the generation of hash tables requires thread synchronized accesses, the hash table inference is easily implemented using manycore solutions. The high level of parallelism benefits this approach for CPUs.

In general, modern CPUs are highly optimized for small random memory accesses and do provide high throughput when utilizing parallelism.

However, hash tables are not ideally suited for FPGA implementation. Whilst FPGAs do provide high throughput computation and memory accesses, they have to be utilized in a different manner than CPUs to achieve the best efficiency in accelerator designs. As described in Section 3.1, FPGA memory subsystems require significant burst sizes to accommodate for the lack of caches and achieve high throughput. This does not apply well to the small random accesses that are needed for the generation and inference of hash tables. Additionally, the most optimized FPGA accelerators

still work at an order of magnitude lower operational frequency than CPUs and accommodating for high throughput requires a rather wide datapath. This results in less flexibility for semi-random access approaches inside algorithm implementations.

Thus, on FPGAs, algorithms such as join and group-by tend to be more suitable utilizing merge approaches. Using high-throughput hardware sorting followed by merge join or group-by algorithms avoids small random memory accesses, keeping all data processing in a stream fashion, also making the approach suitable for future stream throughput increase.

4.4.2 High-Throughput Sorting on FPGAs

In recent years, the topic of high-throughput sorting has emerged again with the publishing of the Parallel Hardware Merge Sorter by Song et al. [102]. It was then extended by, Mashimo et al. proposing the High-Performance Hardware Merge Sorter [65], followed by Saitoh et al. proposing the Massive Merge Sorter (MMS) [96] and the Very Massive Sorter [97]. Most recently, Papaphilippou et al. proposed the Fast Lightweight Merge Sorter (FLiMS) [83, 84].

These sorters achieve very high sorting throughputs that are significantly larger than current memory technology can sustain. When targeting a particular platform, the need to increase utility at the cost of throughput is rather obvious as long as the throughput of sorting can sustain the memory throughput.

4.4.3 Algorithm for Sorting on FPGAs

Sorting of data that is larger than the available on-chip memory cannot be done with a single run of the data through the chip. This is by definition, due to the need for any algorithm to have seen all values at least once before any sorted output can be produced. The state-of-art approach for FPGA sorting is to split the runs into a single linear-sort run followed by merge-sort runs [54].

Linear phase

The purpose of the initial sorting run is to convert unsorted data into sorted sequences (L-way Linear Sort in Figure 4.7). The utility of linear sorters is depicted by the size of the sequences they can produce. The number of output sorted sequences is the total

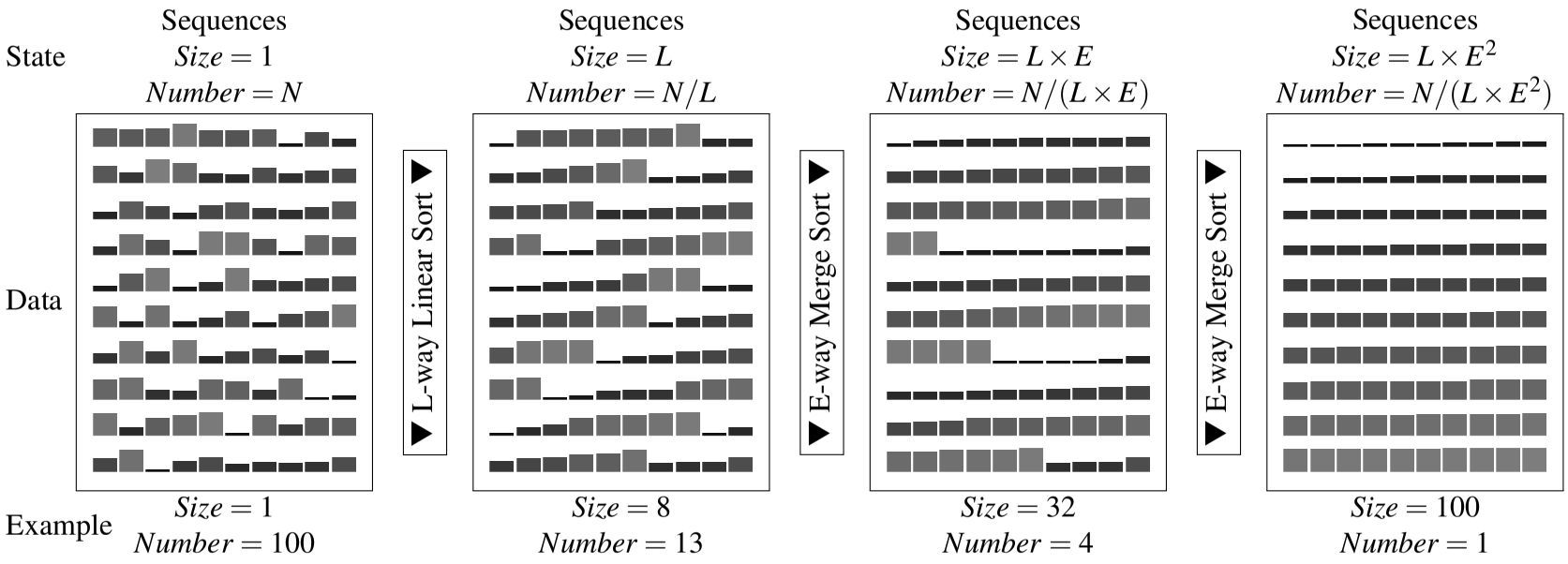

Figure 4.7: Data states in Linear-Merge sort sequences. The example shows multiple stages of sorting random data using L-way Linearsort (L=8) and E-way Mergesort (E=4).

number of elements (N) divided by the utility of the linear sorter (L):

$$Number = \lceil N/L \rceil ; Size = L$$

Larger sequence sizes result in a smaller number of sequences, which consecutively results in fewer merge stages. The utility of the linear sorters is mostly bound by the available on-chip memory allocated to the linear-sort module. Utility scales linearly with BlockRAM requirement.

Merge phase

Following the phase of linear generation of sorted sequences, the hardware utilizes merge sorting to merge them into larger sequences (E-way Merge Sort in Figure 4.7) The utility of merge sorters is depicted by the number of sorted streams they can merge at once. The number of required FPGA runs scales logarithmically with the number of sorted sequences: $number\_of\_runs = \log_E Sorted\_sequences$. *More sequences merged at once results in a smaller number of runs which increases effective performance.*

Sorting complexity

Our algorithm can map the sort problem in space (through the utility of the hardware modules) and time (through the number of hardware runs). This paragraph considers the cooperation of a L-way linear sorter and an E-way merge sorter. Considering the following example: sorting of N elements of size P each using hardware running at frequency F and sorting R elements per clock cycle. In every linear and merge stages the whole data has to be streamed through, thus when they operate with the same rates, they have the same runtime:

$$T_{linear} = T_{merge} = \frac{N \cdot P}{F \cdot R \cdot P} \tag{4.1}$$

The total execution runtime of a sorting sequence comprising of a linear sort phase and multiple merge sort phases is equal to:

$$T_{total} = T_{linear} + \sum_{i=1}^{\log_E \frac{N}{L}} T_{merge} \tag{4.2}$$

Combining 4.1, and 4.2 we can calculate the total sorting execution time as follows:

$$T_{total} = \frac{N \cdot P}{F \cdot R \cdot P} + \lceil \log_E \frac{N}{L} \rceil \cdot \frac{N \cdot P}{F \cdot R \cdot P} = \lceil 1 + \log_E \frac{N}{L} \rceil \cdot \frac{N \cdot P}{F \cdot R \cdot P} \tag{4.3}$$

The sorting throughput is bound by the equation of constants $F \cdot R \cdot P$, which requires $2 \cdot F \cdot R \cdot P$ external memory throughput to fully saturate the sorters' input and output ports. Otherwise, the problem becomes memory-bound and is limited by the available external throughput T:

$$T_{total} \approx \frac{N \cdot P}{T} \cdot \lceil 1 + \log_E \frac{N}{L} \rceil \tag{4.4}$$

Achieving a time complexity:

$$O(N \cdot \log_E \frac{N}{L}) \tag{4.5}$$

Although, in theory, the time complexity evaluates to $O(N \log N)$ (in Bachmann–Landau notation [6, 55]), in practice, the runtime can be greatly optimized by increasing the utilities (L and E) of the linear and merge sorters which will each have a logarithmic performance impact.

4.4.4 Merge Sorting on FPGAs

Traditional FPGA Merge Sorting

Most software algorithms for sorting are not easily applicable in hardware as they often require much conditional execution and random memory accesses. Merge sort, on the other hand, is a very promising approach for sorting in hardware. This holds in particular for *external sorting* where the problem is too large to fit into RAM and where the problem and even temporary data is commonly stored in mass storage (e.g., SSDs). The naive implementation of a hardware sequence merger utilizes a balanced binary tree structure with almost no control logic (see Figure 4.8 a). The comparing cells are implemented with processing elements that can sort when meeting two conditions: the output FIFO is not full and both input FIFOs hold data to sort. If these conditions are met, sorting takes place from the input FIFOs and (lets say) the smaller value will be pushed into the output FIFO. The FIFO buffer sizes do not have to be large because they basically decouple operation from control such that no global control is needed and each sorting cell is self-timed and controlled by the output and input FIFO fill levels. Each stage k of such a tree consists of 2^k FIFOs and 2^{k-1} compare units. Thus, an E-way merge sorting tree consists of $2E-2$ FIFOs and $E-1$ comparators.

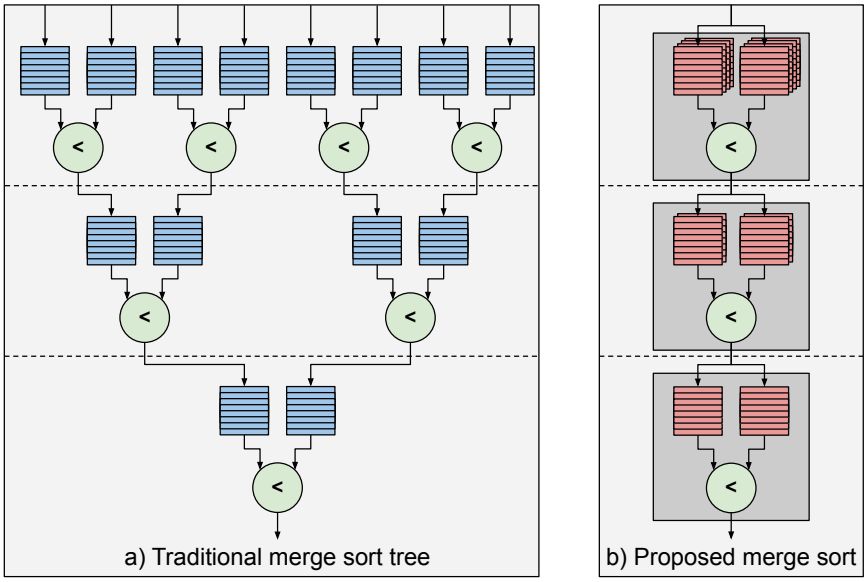

Figure 4.8: Traditional merge sorters (a) implement a tree of cascaded FIFOs and comparators. Proposed sorter (b) shares the compare elements in each stage (level of the tree) as well as compacts the FIFOs into larger blocks with shared read/write ports.

Traditional Merge Sorter Advantages

- This basic approach uses only little control logic and is easy to implement.
- Only fast local communication between processing elements is needed because every sorting cell is connected to at most three other cells in the binary tree.
- The critical path includes essentially only the key comparison and a 2:1 MUX.

Traditional Merge Sorter Disadvantages

- The linear complexity growth of the proposed binary tree makes it unfeasible for implementing a design that merges a large number of sequences.
- For FPGA implementations, this design intends to under-utilize the FIFOs between the stages that would either be implemented using BRAM or distributed memory primitives (e.g., SRL16 primitives in the case of Xilinx FPGAs) that could hold much more entries than actually needed.
- When starting the sorting process, all buffers at the input side will request data at the same time which requires a large amount of resources for the input buffers.

Optimisation

When analyzing a traditional E-way merge sorter, we can observe the following characteristics which we will exploit for an improved implementation:

- In the steady state where all FIFOs are full, then whenever one output record is read, only one empty token will be replaced from each of the tree levels.
- Consequently in each level of the tree, only one compare unit is active as well as one (input) FIFO pull request and one (output) FIFO push request. This means that it must be possible to implement the sorter cell and the FIFOs using shared compute and memory resources.
- Additionally, a load unit that would be in charge to fill the input FIFO buffers will at most receive one refill request per cycle which implies that we don't need any complex multi-channel arbitration scheme.

Considering these observations leads to an efficient merge sorter design that shares compute and memory resources within sorter tree levels [46, 111, 61].

It should be mentioned that the observations apply to decision trees in general and that much of the work presented in this thesis can serve as design patterns for implementing such trees efficiently on FPGAs.

4.4.5 Utility of Merge Sorting

Although this work is exemplified for an in-memory database, future works might target databases of sizes significantly larger than available DDR memory or large problems that are not in the database field. Thus this section presents the major importance of utility for general problems of very large data sizes.

Because external sorting has to work on data from mass storage devices where access (throughput and latency) is relatively expensive (as compared to memory or even on-FPGA data access), it is of paramount importance to minimize the number of major runs. We use the term *major run* to express a major sorting step where the entire problem is read from mass storage, then processed and finally written again (to the same or another) mass storage device. A major run may include one or more *minor runs* through local memory (e.g., DDR memory) that work on smaller problems. For example, unsorted data may be read from disk, then passed through a linear sorter that works entirely with on-FPGA memory, followed by an intermediate small merge

sorter step through DDR memory which produces a temporary merged sequence that is merged again by another run through the chip before writing the generated sequences finally to disk.

If we assume for the previous example a system that is providing an aggregated disk throughput that allows reading or writing the entire problem in time T, then we need for sorting at least the time $2 \times T \times number\_of\_runs$. Similarly, the energy needed for sorting is mostly depending on the problem size and the number of major runs. Because sorting cannot deliver a result before all input records had been seen by the sorter at least once, external large problem sorting cannot be accomplished in a single major run. Consequently, most external sorting approaches aim for two major runs where in a first run large sorted sequences are generated that are merged (ideally) in just one final major run. Therefore, external sorting through mass storage devices takes at least the time $4 \times T$ (for two major runs requiring disk read and disk write).

In order to perform large problem sorting in just two runs (or a few runs), the goal is not that much to deliver high throughput per run (as long as we can saturate the mass storage or memory throughput), but on the *utility* per run.

Because after each major and minor run the produced sorted sequence gets longer, in particular, the final merge step needs attention as this sort step is normally to be accomplished with orders of magnitude less memory than needed to store the entire problem. As a practical example, let us assume that we want to sort a problem that is 1 TB in size and that the first major run generated 1024 sequences that are each 1 GB long (on disk). When using a high utility merge sorter able of merging those 1024 streams in a single run, RAM is only needed to buffer mass storage access. However, when considering a minor temporary run using a 32-way sorter cascaded with another 32-way sorter (for effectively sorting $32 \times 32 = 1024$ streams), this run would require memory to store at least 1 TB / 32 = 32 GB in addition to the memory needed for buffering mass storage access, which would be more DDR memory capacity than available on most FPGA boards.

For big data sorting on FPGAs, merge sorters with a large number of merged sequences are highly beneficial for the subsequent runs. Even Intel's and Xilinx's new FPGA devices with embedded fast HBM memory [27, 131, 130] would not help here because for external sorting: 1) we are I/O bound by the mass storage devices and 2) the limited (HBM) memory sizes available cannot fit the problem. For example, as discussed above, to sort 1 TB using 32-way sorter, we need 32 GB of DRAM only for intermediate data storage which is more than all current HBM sizes available.

4.4.6 Large Utility Merge Sorter

Top Design

The proposed design of an E-way merge sorter consists of $\log_2 E$ sorter cells. Each of these sorter modules is in charge of all the sorting in one of the stages of a traditional hardware merge sorter implementation. The cells are arranged in a linear fashion, meaning that cell k (implementing stage k) communicates only with the adjacent cells $k+1$ (referred to as the previous cell) and $k-1$ (referred to as the next cell).

Sorting Cell

Assuming steady state, every stage k of the sorter consists of a single sorting cell that represents the 2^{k-1} compare units and 2^k FIFOs. While a single combined compare unit can be used for the sorter cells, we need at least two FIFO elements, because it requires the content of two different FiFOs for the compare. This imposes an implementation requiring at least two random access memories for mapping the buffered data as shown in Figure 4.9.

Logical FIFOs

The logical FIFOs provide an abstraction layer that implements a first-in-first-out memory with parameterized depth, width, and the number of channels. The module maps multiple channels to a single shared (simple dual-port) memory. This uses the observation that for all left and right FIFOs in the traditional merge tree there is only a single aggregated read per input branch and a single aggregated write operation needed per clock cycle. To implement the actual logical FIFOs, we use two additional memories (i.e. essentially register files) for holding the read and write pointers per channel. The dual-ported memory buffer uses then a concatenation of the channel identifier and the corresponding pointer for addressing (as we use powers of two for logical FIFO sizes). With this, a push writes data into the corresponding buffer followed by an increment and update of the corresponding write pointer register file entry. Respectively, a pop command results in an increment of a read pointer (given by the requested virtual channel).

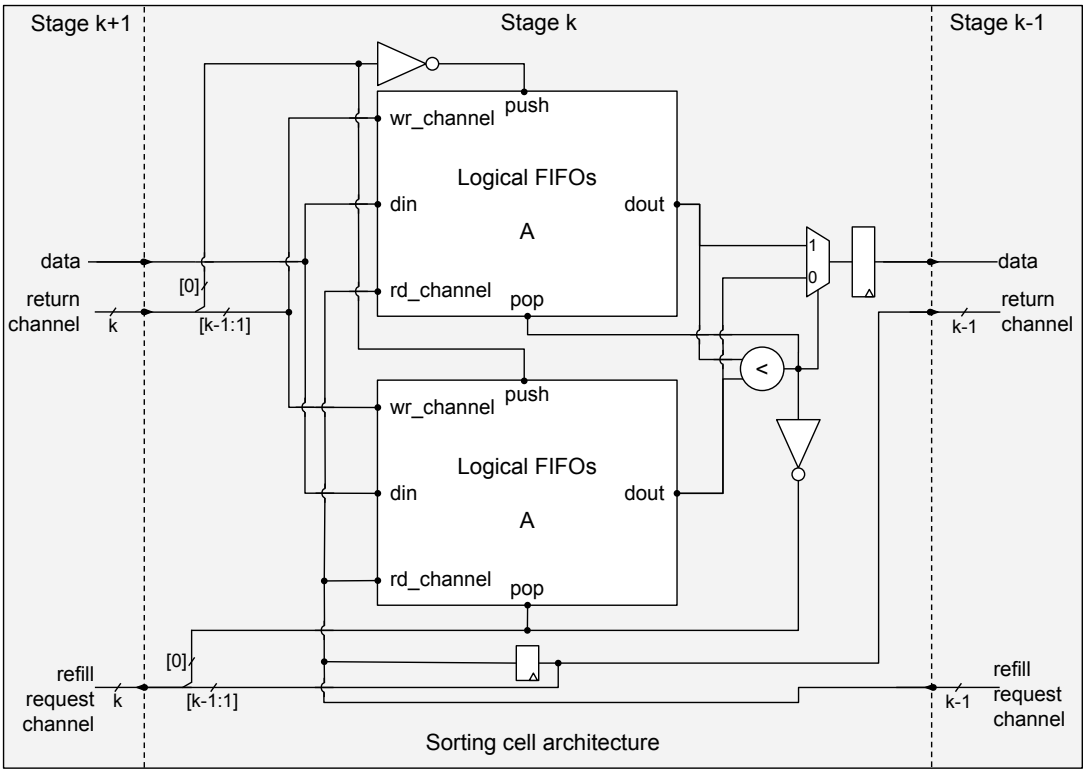

Figure 4.9: The sorting cells of the large utility sorter use two virtual banks of FIFOs.

Synchronization

In the traditional tree design, each compare cell checks for space in its output buffer. However, continuous polling on all channels is not easily possible when sharing one buffer for multiple mapped FIFOs. To overcome this, upon pop command, stages generate refill requests to their previous stage. This means that in each stage of the tree only one input FIFO channel will be pushed and that results in a corresponding refill request to the previous stage. For a refill request, the corresponding FIFO identifier is passed to the previous cell.

Initial and Steady States

The previously described scheme demands that the data buffers are already full. We, therefore, incorporated an initializing phase that fills all sort stages in order before changing to full streaming mode.

Flags

The implementation of the proposed design adds one flag bit to each record (which can most of the time fit into the parity bit of the used BlockRAM buffer). The flags indicate if a record is finished and not valid. With this, the sorter terminates if all sequences signal being finished. This also allows the sorter to have channels that stay empty and allows us to disable individual channels if not all can be used simultaneously (e.g., using only 1000 channels from 1024-way merge sorter), hence making the sorter more generic.

Communication and Stalling

The proposed design does not use request buffers between sorting stages. The sorter cells themselves consume only relatively few resources (usually less than 800 LUTs), which allows them to be placed closely together and consequently with small wiring delays. Not considering request buffers inside the sorting cells results in constant request serve time. However, there are three stalling sources that are considered.

Firstly, the input side of the design may be stalled by the load unit (e.g., due to any disk or external memory congestion). The second stall source is respectively at the output side of the sorter. The design could also stall internally if any of the stages experience data starvation for a specific channel. The forward stalling is implemented using a propagating stall signal. Any cell can also backward stall the design by simply not issuing requests to the previous cells. The first two stalling sources cannot be bypassed, and are 'desired' in a sense that they mean that the design fully utilizes the available memory throughput of the used hardware. The third stalling source can be eliminated if buffer sizes in each stage are big enough [1] to handle fully skewed data without causing data starvation in any stage. If this is not ensured, the design needs additional logic to check whether a request can be served and otherwise stall the sorting in the hazardous stages.

4.4.7 Evaluation and Graysort Case Study

Evaluating the very large utilities requires ($E \geq 512$) more BlockRAMs than available in the targeted ZCU102 PR slot, thus we also evaluate the design on a Xilinx Virtex FPGA. Placing and routing the sorter for Xilinx VC709 board achieves utility (E) of

[1] 8 tuples per virtual channel per stage is sufficient. This results in a significantly large storage requirement in the larger stages (hundreds to thousands of virtual channels).

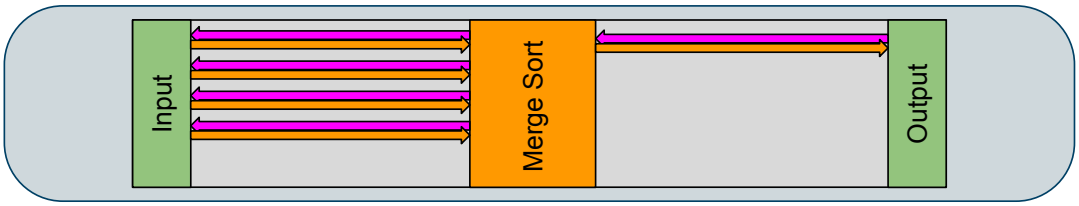

Figure 4.10: Merge sort integration with DSPI. The module inputs many (tens to thousands) virtual streams and merges them to one. It utilizes only one DSPI StreamID, but many ChannelIDs to enumerate virtual streams.

up to 8192 whilst maintaining slice resource requirement under 4K and a frequency above 200 MHz [61]. A case study sorting 4GB of graysort records [33] achieves 81% (75%) of peak dual DDR3 throughput while merging 1024 (2048) sequences in a single run [61]. For the DDR3 memories on the VC709, we need 2.8 KB memory bursts to achieve 81% of the theoretical DDR throughput. The VC709 design requires very large amounts ($E \times 2.8\ KB$) of on-chip memory to simply buffer the read operations. The modern ZCU102 DDR4 achieves 98.5% of maximal AXI performance requiring bursts of only 192 Bytes (see Section 3.1). This greatly reduces the required on-chip memory to buffer read operations and allows us to fit a sorter in the limited small PR modules.

4.4.8 Large Utility Merge Sort for DSPI

The ideal design for a merge sorter to be placed in a streaming system should omit direct memory access handling from the sort module. Thus the stream implementation of the proposed merge sorter utilizes the ChannelIDs of the proposed DSPI (see Figure 4.10 and Section 3.3.2). The DMA module handles the addresses, sizes etc. of the sorted sequences that are to be merged, providing a full abstraction for the merge sort module.

Building the merge sort module as a DSPI streaming module for $E = 64$ maps to a BDMMBDMDMM resource footprint (see Section 4.1). While the authors in state-of-art related work exemplify an 8-way merge sorter built as a static module [135], our proposed large utility sorter achieves 64-way merging (8× more sequences merged at once) as a relatively small PR module. In another database engine, AxleDB [98], a static 16-way merge sorter operating at 3.2 GB/s requires 33K LUTs, 34K Flip-Flops, and 33 BlockRAMs, while the proposed 64-way (4× better) sorter operating at 19.2 GB/s (6× better) requires only 6K LUTs (5× better), 7.5K Flip-Flops (4.5×

better), and 35 BlockRAMs. The module is bound by the available BlockRAMs in the targeted PR region, while the module utilizes only 41% and 25% of the LUTs and Flip-Flops respectively. The achieved resource footprint allows the module to be placed into three different positions. The sort module is suited for resource elastic operation as described in Section 5.3. Achieving $E = 64$ with such a small resource requirement is non-trivial, however, the key advantage of the implementation is its scalability with respect to the fabric memory allocated to buffer a larger amount of record data. This module would benefit from implementation to Xilinx Virtex devices [130], where the design would be able to utilize UltraRAMs [129] to buffer the inputs and achieve significant improvement in the number of merged sequences.

4.5 Join

Hash-joins are the preferred target approach for software due to the high memory access flexibility enabled by CPU caches. This is due to the random access pattern of the hash table generation and possibly hash table inference. On the other hand, sort-merge joins provide sequential accesses to external memory when joining, which is significantly more suitable for FPGA platforms. Thus merge joins are more suitable when paired with optimized sorting algorithms. Additionally, often the data is stored sorted by the relation keys and/or is requested to be delivered in a sorted manner, thus in these situations, there is no sorting overhead for implementing merge joins. On the other hand, if considering small tables where the hash tables can fit in the provided on-chip memories, then the random accesses to off-chip memory are eliminated, thus providing line-speed hash-join. This is of course seldom the case when targeting large database problems and in such cases, software query execution would likely outperform FPGA acceleration (due to the reconfiguration overheads). In general, hash-join tends to outperform sort-merge join with small problem sizes, but is less scalable than sort-merge join (due to handling of hash collisions) and sort-merge join achieves higher performance on large problem sizes [110]. The sort-merge join approach can be beneficial even in software execution when considering near memory execution [70].

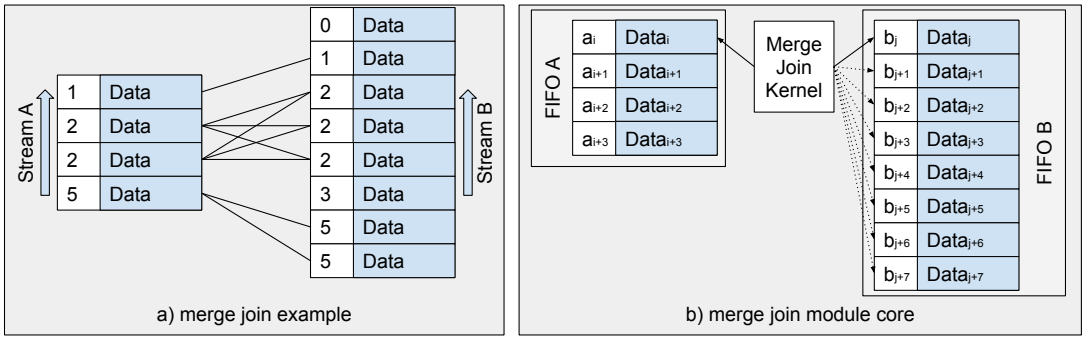

Figure 4.11: A many-to-many merge join has to be able to map multiple records from stream B per record from stream A. Stream A moves only in the forward direction, while stream B might need to use DSPI and roll back in order to recheck join keys.

4.5.1 Merge Join

Casper et al. proposed a merge-join network that evaluates in parallel 4 elements from each of the two streams thus producing up to 16 joined results per clock cycle [19]. Papaphilippou et al. adapted a high-throughput sorting algorithm to build a high-throughput merge-join with late materialization techniques [86]. Although these approaches achieve a very high join throughput, our system can satisfy off-chip throughput by producing only one joined record per clock cycle. Additionally, the implementation of algorithms that produce a variable amount of output records impose more stress on buffering and have significantly larger resource requirements for the core compute algorithms. Thus we propose an optimized merge-join module that consumes and produces one record per clock cycle.

Although one-to-one and one-to-many joins are the most widely used ones (e.g., in all queries in TPC-H [108]), for completeness, we implement a merge join that can also perform many-to-many joins. Figure 4.11 a depicts an example for a many-to-many merge join dataflow. In one-to-one/many merge joins, the records from both streams A and B are examined in a fully sequential order (no relations between records will cross in Figure 4.11 a), which negates the need for local scratchpad memories. In many-to-many joins, however, while stream A is accessed in a fully sequential manner, stream B forces a more complex record access pattern when dealing with key collisions. Most notably, in successive repeating keys from stream A, the lookup pointers in stream B have to be able to be reset to the start of the last unique sequence. Figure 4.11 b shows that we can always keep the current lookup pointer in FIFO A at the front. However, we need full access to the records in FIFO B as the record pull and record reads are

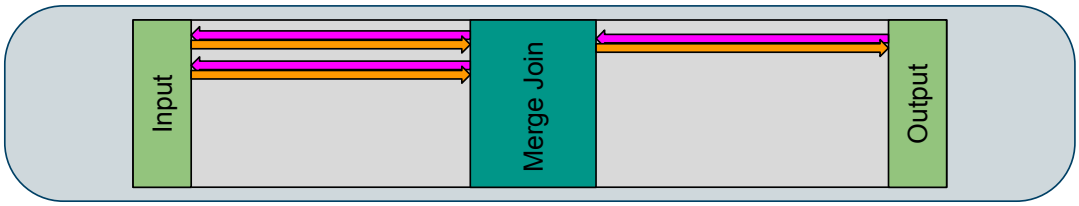

Figure 4.12: Merge join integration in DSPI. The module uses two different DSPI StreamIDs as inputs and merges them into one StreamID output.

completely data dependent. In cases of successive keys in B, we only move the current read pointer in case the data will be re-read to match for another key from A. Then the PE will only pull from FIFO B when $a_i > b_j$.

The challenge in managing many-to-many merge joins is an edge case when two or more consecutive records from stream A have the same key α and there are more records in stream B with key α than fit inside FIFO B. We handle this edge case by utilizing the *Repeat* instruction of DSPI (see Section 3.3.2). The module finishes all join matches with a_i by requesting all matching records from stream B. Then if the new a_i (the old a_{i+1}) key holds the same value, a *Repeat* instruction is fired for stream B. Then the module can proceed to request again the already streamed records from B and match them to the new key from A. Noting that repeat instructions at the moment are only used as a hazard avoidance mechanism, as they pose a large latency and a system stall. The DMA needs to recalculate DDR addresses, flush all prefetched records and re-request records from DDR. Calculating the repeat parameter requires additional consideration in certain module combinations, for example, if the merge join module operates on the output from a filter module. Our filter module can split a table into two virtual tables (for valid and invalid data) and the merge join module can count both.

Merge join in DSPI

The merge join module in our system uses two different DSPI StreamIDs as inputs and merges them into one StreamID output (see Figure 4.12). We build the design for our system with FIFO sizes for streams A and B of 512 and 1536 chunks respectively. The depth is measured in chunks rather than records since the module is runtime parameterizable for record sizes and data merge behaviour. Upon a successful join of two records, the module can reorder data fields in time (i.e., selectively move data fields in different ChunkIDs). This allows for more complex integration of the joined record and positioning data appropriately to meet the requirements of any following modules.

It also can omit data fields or even whole chunks from the output if they are no longer needed. Building the module results in BDMMBDMDMM resource footprint (see Section 4.1). Utilization analysis shows that it is bound by BlockRAMs, while LUTs and FFs are utilized only at 32% and 17% respectively.

4.5.2 Hash Join

For our prototype system (see Chapter 6), we implemented the proposed merge join operator. However, we also propose methods for integrating hash join modules into our streaming system by utilizing mechanisms in the proposed DSPI protocol and DMA module. Hash joining implements a relatively straightforward algorithm. It is divided into two stages of operation that execute sequentially and share intermediate data in the form of a hash table:

hash table generation : The first step is to fill a hash table with one of the two input tables (let's call it table A). This is done by applying a hash function to the keys and using the resulting hash as a bucket selector. When two keys have the same hash result (hash collision), their records are allocated to the same bucket.

hash table lookup : Once all the data from table A is streamed and hashed, then the processing element needs to stream the other table (table B) and find matching keys. This is done by utilizing the same hashing algorithm and looking up the corresponding bucket in the generated hash table. All entries from the bucket have to be compared to the candidate from table B.

The intermediate hash table data must naturally be stored in off-chip memory due to its potentially larger sizes than available on-chip memory [121, 45]. In software, buckets (collections of hash collided records) are stored in the form of linked lists. This allows for a flexible and efficient memory allocation but results in dynamic jumping when looking up the data to find matches. This solution has been proposed in hardware as well, where linked lists are stored in FPGA BlockRAMs with associated DDR addresses for bulk data [35, 135]. However, storing any kind of individual entry for every hashed record in on-chip memory is unfeasible for large problem sizes. Even storing only one DDR address and length per bucket limits the number of buckets to only thousands.

Ideally, we want a hash algorithm that distributes hash values in a uniform way [93]. Hash computing complexity is negligible with respect to modern FPGA fabric capabilities. With uniform distribution, hashing millions of records into thousands of buckets

still inevitably results in thousands of collisions per bucket, which poses a sufficiently large lookup complexity in the second stage of a hash join operator.

Direct memory access

Traditionally, FPGA hash join implementations hold DDR pointers into local scratch-pads. Implementing such algorithms is feasible in our system due to the support for direct memory access from the streaming PR modules (see Section 3.3.5).

Streamed hash tables

On the other hand, our system also supports addressing thousands of tables. We can utilize this by assigning buckets directly as streamed tables of records (see Figure 4.13):

hashing phase : The hashing phase (see Figure 4.13 a) is simply implemented by applying the hash function to the keys of the first streamed table (table A). The result from the hash is then used to select a virtual channel. The module does not need to build any kind of hash tables, since these tables (buckets) are handled already by the DMA module as streams.

joining phase : In the second phase of the stream, hash join (see Figure 4.13 b), the module streams in data from the second table (table B). The data enters a FIFO, where it will wait for the readout of the associated bucket. To read the bucket, the key is hashed and the result is fed as a virtual stream selector into an instruction targeted to the DMA module. It actually requires two instructions: first it needs to restart the corresponding virtual stream and then it can request data from it. Restarting the stream (see Section 3.3.2) will bring the read pointer back to the beginning of the bucket.

When the bucket is streamed as table A, it enters another dedicated FIFO. The output of the two FIFOs is then joined together by comparing the records' keys. Whenever the DMA module (and other modules) finish reading/computing data on a virtual stream, they then send an End-of-Stream packet. The hash join module uses this EoS packet as a mark of the end of a bucket and pulls the front of the B FIFO, thus moving stream B forward.

With the proposed approach, we abstract any challenges with the bucket storage on off-chip memory and take advantage of the DMA module, whose main function is to track the physical off-chip representation of streams. This is key to implementing module

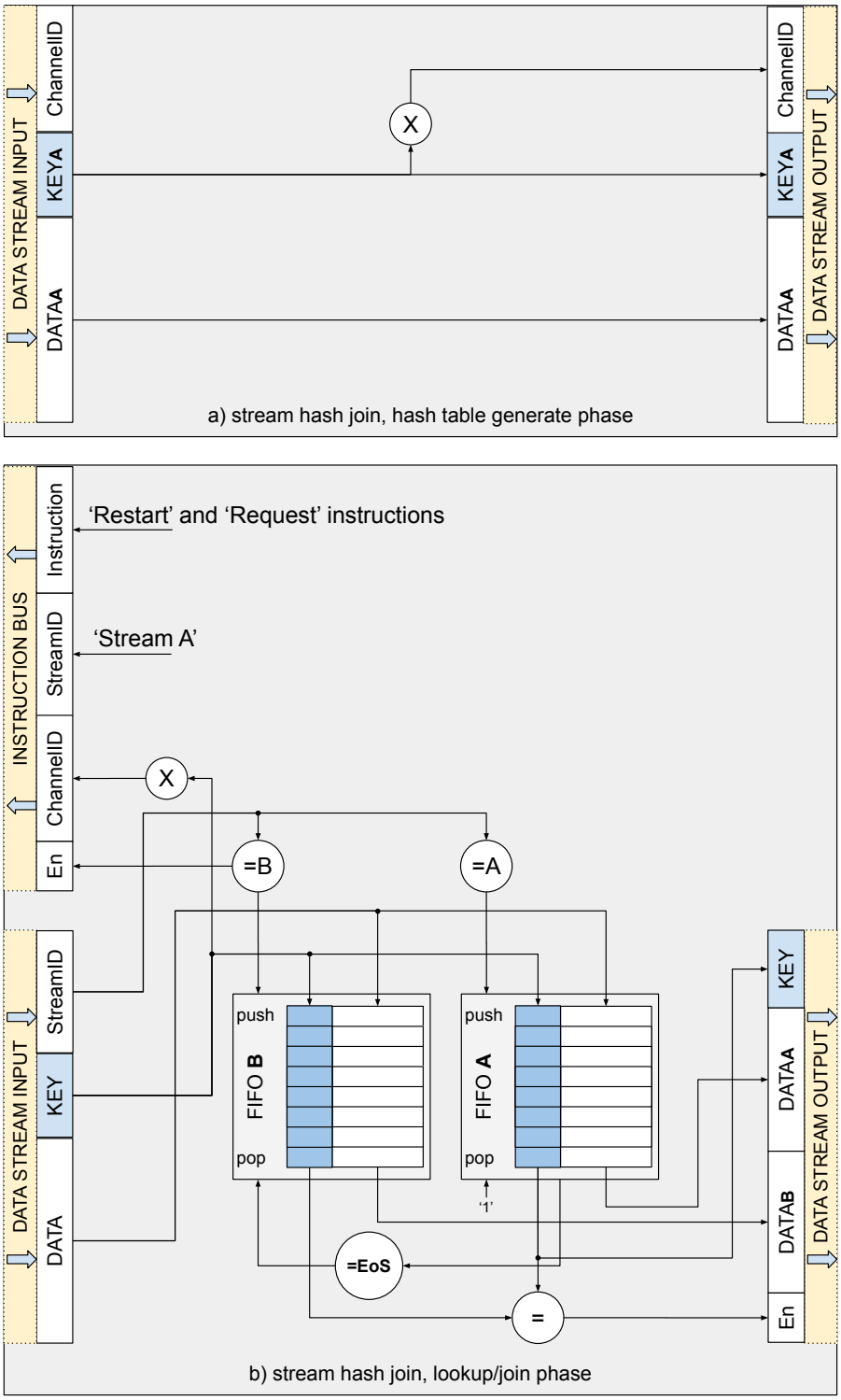

Figure 4.13: A hash join operates in two sequential phases: a) split data into buckets based on the hashed key, and b) lookup buckets based on the hashed key from another table and perform actual join.

portability, as the join module implementation remains the same across different FPGA targets. The DMA module is responsible to store and readback the buckets as streams and can vary between systems depending on the memory handling parameters. If the FPGA operating system allows hardware memory allocation, the bucket streams can be stored as a linked list of large (Kilobytes/Megabytes) blocks, which implemented in the DMA module will be reused for all data streams in the system. Furthermore, the DMA module eventually can also implement the direct usage of non-volatile memories, thus allowing for huge amounts (larger than any available DDR size) of intermediate table storage.

4.6 Conclusion

This chapter introduced a new methodology to implementing thin and tall partially reconfigurable modules by adopting design factors such as minimizing their vertical wiring. The presented concept includes a major increase in flexibility and utility for achieving a more versatile module library. The chapter introduced state-of-art designs and implementations for the most used functions in database systems:

DMA module : We implement a DMA module that is used to decouple our DSPI from standard AXI master and slave interfaces. This module provides address handling, data permutation to abstract these challenges from our streaming accelerators. Additionally, the DMA module provides stream prefetching in order to minimize serve times and maximize effective throughput.

Filter : We implement a filter module that utilizes standard Boolean expression representations (Disjunctive Normal Form) to provide flexible and efficient hardware design (see Section 4.3). Our large utility approach results in 13× fewer resources when executing on TPC-H Q19 than related work [135], while achieving both higher streaming and effective throughputs (see Section 4.3.4).

Merge sort : We propose and implement a highly optimized large utility merge sorter module that, depending on the allocated FPGA resources, achieves up to thousands of merged sequences at once. State-of-art related work on dynamic database acceleration implements an 8-way merge sorter as a static module, while our 64-way merge sorter (8× better utility) is PR-capable (see Section 4.4.3). Generally, other FPGA sorting research focuses on large throughput, but once DDR

bandwidth is saturated, their resource-costly designs are impractical (see Section 4.4.2). Increasing the number of sequences that a module can merge at once results in a reduction of the number of required runs, leading to an increased real-world sorting performance. We anticipate this is key for FPGA sorting and database acceleration (see Section 4.4.5).

Join : While state-of-art related work proposes a static one-to-many merge join [135], we implement a PR-capable many-to-many merge join. Additionally, we show how different operators can be integrated and orchestrated in our streaming system through the proposed hash join streaming modules.

State-of-art research implements merge sort, hash join, and merge join operators as static accelerators [135]. However, this could lead to overprovisioning or underprovisioning of compute resources depending on the target query. We show that it is possible to build these operators as partially reconfigurable modules, thus benefiting from the dynamic aspects of such a system. Not only does this allow us to only reserve the compute resources if they will be utilized, but also we enable the runtime system to trade-off between performance and resource usage by applying resource elasticity. Chapter 5 proposes the implementation of resource elastic techniques in the key modules of our module library. Prototyping the modules in Chapter 6 shows they achieve the target operating frequency of 300MHz (the speed required to saturate a DDR4 memory channel operating at a 512-bit datapath) and that accelerators operate at stream speeds. *Overall, all proposed designs are scalable, process one chunk per clock cycle, and achieve our target throughput of 19.2GB/s.*

Chapter 5

Resource Elastic Module Library

In this chapter, we describe the concepts, methods, and benefits of implementing resource elastic modules for our library. We do this by implementing composable modules with module alternatives, which allows a runtime scheduler to tailor the execution pipeline to the accelerated problem and the available resources, which are only known at runtime (see Section 5.1). We then exemplify the techniques by applying them to three key operators from our module library presented in Chapter 4 to propose:

- A composable DNF filter with module alternatives (see Section 5.2) that can trade-off between filtering capacity, throughput and resource requirements.
- A linear sorter and our composable merge sorter both with module alternatives (see Section 5.3) that can trade-off between throughput and resource requirements.
- A merge join module and its module alternatives (see Section 5.4) that can trade-off between join capacity and resource requirements.

5.1 Resource Elastic Techniques

While traditional CPU scheduling has mainly been fixated on scheduling in the time dimension, FPGAs provide spatial resources where we need to schedule as well. While CPUs can provide time slots of different length for the different tasks, FPGAs can schedule modules of varying resource requirement, throughput, utility. Resource elasticity is a technique to provide alternative execution plans to the runtime scheduler

which significantly improves placement flexibility, often resulting in improved effective performance [62, 114, 117]. In runtime systems that manage multiple FPGA PR slots, resource elasticity can be materialized into two approaches [117]:

Implementation alternatives utilize synthesis of modules with different utilities and targeting different resource footprints (see Figure 5.1 a). In FOS, this results in modules that can utilize 1,2,3, or 4 neighbouring FPGA PR slots. The alternatives implement the same function but allow for various techniques to be applied to achieve linear or superlinear performance scalability. For example, often enlarging a module implementation would allow for larger local scratchpad memories, extending local data reuse and resulting in superlinear performance improvement for certain algorithms (such as large matrix multiplication).

Replication allows a single module to be placed into multiple PR slots (see Figure 5.1 a). This approach is proposed for FOS by providing parallel execution for data-parallel OpenCL workgroups [77]. Replication of OpenCL compute kernels in FOS can provide multiple benefits. First of all, FOS can change dynamically the execution kernels. The addition of replicated single-slot module poses a significantly smaller reconfiguration overhead than changing the bitstream in multiple PR slots. Secondly, the spatial dynamic allocation would eventually cause fragmentation challenges. With module replication, the modules placement positions can be abstracted as long as they all have access to the same processed data. An example situation is whenever a slot is released and the selected operation for further acceleration does not occupy a neighbouring slot to the freed one. In that situation, the runtime system can use module replication to bypass the fragmentation and avoid large overheads of FPGA-wide module relocations.

While the concepts of resource elasticity remain the same, one does not simply apply the resource elastic solutions presented for FOS into a streaming system. FOS largely targets OpenCL [77] workloads for their ability for heterogeneous CPU+FPGA execution [113]. OpenCL splits the problem into workgroups that share no global state, thus allowing for straightforward parallelism. This allows for easy management for implementation alternatives and replicated modules in FOS. Such workload properties and benefits, however, are not present and applicable in streaming systems. Our system presents the same two approaches for resource elastic application, but with different considerations for integration, evaluation, and management.

5.1.1 Resource Elastic Module Alternatives

Since module compute kernels tend to scale either linearly or superlinearly, providing larger module alternatives can be highly beneficial. Similarly to FOS, we examine the implementations of modules with different tradeoffs between resources and utility/performance (see Figure 5.1 a). FOS targets four PR slots thus considering only four possible module alternatives. On the other hand, we target relatively fine-grained partial reconfiguration thus enabling tens to hundreds of possible module sizes and placements. However, most module sizes and placements will not be scenarios on the Pareto-Front (see Section 4.1.3) thus can be omitted. This means we will only include the smallest modules that deliver a specific performance or utility in our operator acceleration library. Let us consider this approach as a string matching problem. The larger the substring (module resource footprint), the fewer times it will occur in the search string (PR slot resources). This results in a disadvantage for increasing module resource footprints because often it results in a reduced number of possible bitstream placement positions. However, as long as a module alternative is on the Pareto-Front, its presence in the module library can only be beneficial. The runtime scheduler will omit the use of any unoptimal configurations depending on the runtime requirements. Thus, increasing the number of module alternatives can only enable alternative execution plans to the runtime scheduler and in the worst-case scenario results in no performance change.

Pareto-Front

Most often, implemented modules for such a system will have coarse-grained control over utility. This results in a synthesis process where we adjust utility and observe the resulting module footprint, rather than develop a specific utility to meet a certain resource footprint. However, fine-grained adjustments to the utility can result in a relatively small resource impact. Considering that our atomic double-column resources provide about 1-2K LUTs, we expect that small utility adjustments can result in an unchanged resource footprint but different internal fragmentation. In such cases, we can consider the Pareto-Front of the possible module alternatives and omit all modules that are not on the front. We show an example of this when implementing our resource elastic filter module (see Section 5.2.1).

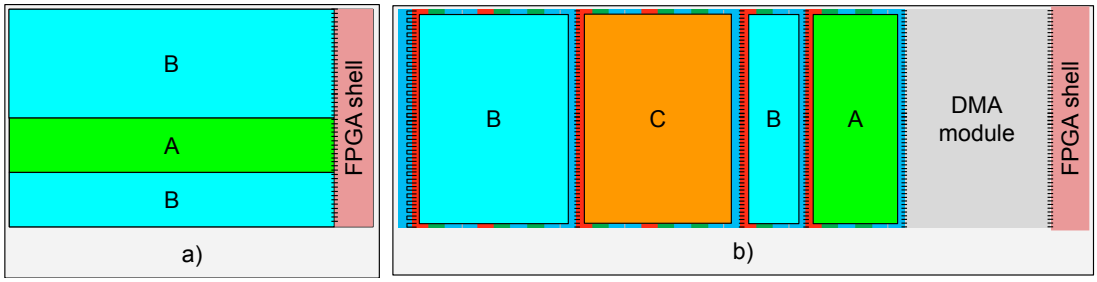

Figure 5.1: a) FOS utilizes four slots to accelerate OpenCL workgroups in resource elastic fashion by using module replication (to provide parralel execution) and implementation alternatives (to improve throughput). b) The proposed stream processing execution utilizes FOS slots in a fine-grained manner. All modules operate at stream speeds but utilize module alternatives to trade-off between effective throughput/utility/capacity and resource requirement (see accelerator B). Additionally, module composing allows accelerators to implement larger logical function (the two instances of accelerator B working as one logical operator).

5.1.2 Resource Elastic Composing

Similarly to the module replication in FOS, we propose methods for composing compatible modules to increase performance (see Figure 5.1 b). FOS implements such techniques to provide execution parallelism, ideally resulting in increased throughput. However, our system has a defined streaming throughput (19.2 GB/s in our example system) and does not naturally benefit from parallelism. However, we can extend the effective functionality of our modules by placing multiple instances sequentially in our streaming system. Enabling such functional extensions, module kernel functionality has to be considered during design and implementation. Often modules would need to implement intermediate data state transfers to allow for function extension. DSPI provides numerous ways for intermediate state transferring between modules. Most notably, DSPI provides a *State* bus (see Section 3.3.2) that can be used to pass any encoded intermediate state of the record between resource elastic modules. The used virtual stream of a data packet can also encode the data state, origin, and destination which can also be used to extend functionality.

Smaller modules have greater placement flexibility, thus their use results in a decreased external fragmentation and a globally improved resource utilization. Providing such modules will increase the scheduler with more placement flexibility for the runtime system. The runtime can decide to pick multiple smaller composable modules as an alternative to one large module if it cannot be placed due to fragmentation.

5.1.3 Resource Elastic Stream Processing Advantages

Streaming modules can vary greatly in their operation, thus also allowing for different characteristics to be targeted for improvement by resource elasticity. Many characteristics also have a varying impact that is only known at runtime, such as problem sizes and operation complexities.

One of the main benefits of resource elasticity is the ability to trade-off between key module parameters including utility, throughput, compute capacity, total runtime and resource requirements. This holds for both single-user and multi-tenancy scenarios. The trade-off parameters depend on the specifics of the module operation and implementation. The total runtime is a function of utility, throughput, and capacity and how do they satisfy the runtime problem:

utiltiy : The increase of modules utility results in more work performed per unit memory I/O, thus leading to increased performance for memory-bound problems.

throughput : Modules can vary the achieved effective throughput. Although DSPI runs at a fixed communication bandwidth, the effective throughput can vary if modules require data field replication, thus increasing the number of clock cycles per transferred tuple. Additionally, modules can potentially delay the generation of output packets (effectively producing empty cycles). Limiting the allocated FPGA resources for a module could lead to a requirement of multiple clock cycles to produce an output packet. In such a case, the kernel processing uses local scratchpad memories to decouple its input from its output while processing.

capacity : Stream operations require a certain amount of processing elements. If modules do not meet the required processing capacity, this can lead to effectively increasing the runtime. This is due to the need for additional runs of the data through the FPGA to achieve the targeted operation. Note that our dynamic stream processing approach foresees an execution in multiple runs if resource requirements exceed the currently available resources.

The applicability of these parameter trade-offs is dependent on the core operation of the targeted module. However, careful module design can elevate the benefits of allowing the runtime to selectively optimize the execution pipeline.

Decoupling PR slots

Another major benefit of composable resource elastic modules is that they enable the operation extension from different PR slots. When targeting acceleration in large-scale systems such as datacenter settings for warehouses and data analytics, systems could utilize multiple PR slots for processing element placement. These PR slots can then be composed such that the DSPI output of one is the DSPI input of the next one. This also includes cases with inter-FPGA PR slot composing. Our solution for seamlessly stitchable operation extension permits the utilization of fabric across multiple PR slots. This is otherwise implementable by using off-chip memory to decouple operations, which can further bottleneck an already-memory-bound system.

5.2 Resource Elastic Filtering

Filtering operations comprise data comparison followed by Boolean evaluation. Similarly to other operations, the complexity of the filtering operation is only known at runtime. Traditional solutions to this problem have been to overprovision compute resources, however since the runtime system knows the problem, the solution can be heavily tailored to the problem. Our proposed filter module that utilizes DNF boolean evaluation (see Section 4.3) benefits from both types of resource elastic approaches. The DNF boolean evaluation comprises DNF clauses that implement *computational capacity* (i.e., the runtime system must implement at least the number of DNF clause capacity than the problem requires). This further emphasises the need for the ability to tailor the amount of DNF clause capacity placed at runtime. Similarly, the number of comparison processing elements implements a second computational capacity. However, in the latter case, we can trade-off between throughput and capacity. The streaming throughput is, of course, fixed to line rate transactions at a fixed clock frequency. However, we allow the replication of certain data fields to be able to benefit from an increased number of compare PEs. By replicating data, we might need to increase the number of transactional clock cycles (number of chunks) for a particular record, thus decreasing the effective throughput.

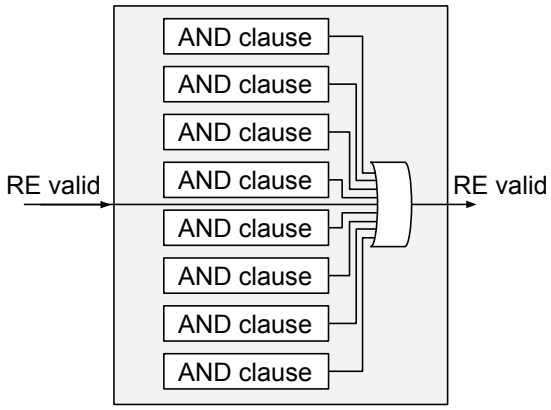

Figure 5.2: Any DNF filter module can compute the global result by incorporating the results from previously placed DNF modules. All DNF filter modules implement a specific number of AND clauses and the OR-reduction of all clause results is global.

5.2.1 Filter Module Alternatives

We can build multiple module alternatives by synthesizing our parameterized filter module with different computational capacities and resulting in different resource requirements:

Compare PEs : We can change the number of parallel compare elements per data field. Each compare element imposes not only the resource overhead for the comparison itself, but also can hold unique compare reference values.

Clause capacity : Each DNF clause has to implement solving for all literals that result from the compare elements. Each literal implements one of 3 states for each DNF clause.

Although we can implement any arbitrary capacity that fits the FPGA resources, we restrict our implementation to three common cases for each parameter. We build a total of nine modules with 8, 16, and 32 DNF clause capacity and 1, 2, and 4 compare PEs per field. *All module alternatives work at line speeds (filtering one record chunk per clock cycle).*

5.2.2 Filter Composing

Large DNF problems can be split into multiple buckets of DNF clauses and evaluated by different module instances. For this, we need to modify our initially proposed design (see Figure 4.6) to accommodate for the passing of an intermediate DNF global

Smallest resource footprint	DNF clause capacity	Compare PEs	LUT	LUTM	Flip-Flop
MDMM	8	1 per field (16 total)	2,655	1,313	3,171
DMDMM	8	2 per field (32 total)	3,656	1,761	3,285
DMMBDMD	8	4 per field (64 total)	5,635	2,657	3,513
DMDMM	16	1 per field (16 total)	2,866	1,441	3,301
DMDMM	16	2 per field (32 total)	4,059	2,017	3,415
MBDMDMM	16	4 per field (64 total)	6,295	3,169	3,643
DMDMM	32	1 per field (16 total)	3,326	1,697	3,560
DMMBDMD	32	2 per field (32 total)	4,927	2,529	3,674
BDMMBDMDMM	32	4 per field (64 total)	7,675	4,193	3,902

Table 5.1: Filter module resource requirements. Not all modules are on Pareto-Front: the two underlined configurations (8 clauses, 2 compare PEs and 16 clauses, 1 compare PE) share the same resource requirements with a more capable configuration (16 clauses, 2 compare PEs) that provides superset utility in both dimensions.

state. DNF implements a global OR operation of the results of all AND clauses. This naturally lets us split a DNF boolean expression into multiple modules as each module implements a certain number of AND clauses and then uses a global bit for the resulting OR reduction (see Figure 5.2). This intermediate value is streamed on the *State* bus of DSPI (see Section 3.3.2). All modules can be parameterized whether to use that intermediate state and/or to materialize the whole evaluation by optionally splitting the stream into two virtual streams: 1) a stream of all records that satisfied the conditions, and 2) a stream of all records that did not satisfy the conditions. This enables the filter module chain to be also used as a splitter of a stream for further use selection based on a condition.

5.2.3 Evaluation

The resulting FPGA resource utilization from the nine implemented filter module alternatives is presented in Table 5.1. All module alternatives are bound by their LUTM utilization, which is (in most cases) larger than 50% of the total LUT requirements.

Module alternatives

The utility of a filter module alternatives is defined in the two-dimensional space: 1) DNF clause capacity and 2) compare PEs per field. The results from the evaluated filter configurations show that four filter modules alternatives share the same resource footprint (DMDMM). While we observe that our LUTM utilization trends match our prediction from Section 4.3.4, the difference in two of the configurations is insignificant and still results in the same module bounding box. The two underlined configurations in Table 5.1 (8 clauses, 2 compare PEs and 16 clauses, 1 compare PE) are not on the Pareto-Front, since they share the same resource requirements with a more capable configuration (16 clauses, 2 compare PEs) that provides superset utility in both dimensions. On the other hand, another configuration (32 clauses, 1 compare PE) also shares that footprint, but is also on the Pareto-Front, since it achieves higher clause utility than the other configurations. Noting that while the two underlined configurations are unoptimal in the example system, on different FPGA fabrics or a system with different DSPI bus sizes, these utility configurations might achieve different Pareto-Front results. It is up to the runtime scheduler to utilize the available module alternatives to their full potential for building efficient execution pipelines.

Module composing

All implemented module variants also implement composability resource elasticity. What is more, different filter module alternatives can be composed together, which is an important property since the compare operations in a set of filter clauses can be unbalanced. We can demonstrate the concept by example. Let us consider an example filter operation: $(A = 5)\ OR\ (B > 10\ AND\ B < 15)$ and modules with only 1-2 compare PEs per field and 1-2 DNF clause capacity for simplicity. We then can have multiple module schedule alternatives to implement the operation:

Single large module : We can use a single large module implementing 2 DNF clauses (one for $A = 5$ and one for $B > 10\ \ AND\ \ B < 15$) and 2 compare operations per

field (to accomodate the two different reference values for B). In this case, no data replication is required.

Two small modules : Alternatively, we can compose two small modules implementing 1 DNF clause and 1 compare operation each. In this situation the module implementing the $(B > 10 \ \ AND \ \ B < 15)$ clause will need a replicated B field to be streamed into two different compare PEs. This replication can potentially lead to a decreased effective throughput if it results in an increased number of chunks for the streamed record.

Composite : Finally, we can utilize the ability to compose different module alternatives. We do this by using one filter module with 1 DNF clause and 1 compare PE, and a second filter module with 1 DNF clause and 2 compare PEs. In this situation the first module implements $(A = 5)$ and the second module implements $(B > 10 \ \ AND \ \ B < 15)$. Both modules do not require any data replication and work at full system throughput.

The different DNF clause requirements can be of arbitrary diversity when considering real-world filter examples and module alternatives could achieve varying resource footprints on different FPGAs. Then the runtime scheduler can optimize at runtime by exploring available execution configurations for a particular problem and select the most advantageous pipeline.

Filtering in multiple runs

A filter operation requires module(s) to achieve certain compare compute and Boolean evaluation capacities. The fallback mechanism to implementing larger compare capacity has already been discussed - replicating data fields to map to additional (in space and time) compare PEs. However, the DNF clause capacity has to also implement a mechanism to extend the functionality to adapt to problems of large complexity. Using our filter modules, we can split the DNF clauses to be evaluated in multiple runs of the data through the chip. This provides a method for implementing the evaluation of Boolean expressions of arbitrary complexity. To implement this, the last composed filter module materializes the intermediate DNF OR-reduction in each data run through the FPGA. This splits the data into two tables: 1) a table holding already validated records, and 2) a table holding records that are not yet validated. In all consecutive runs, only the second table is streamed through for validation using the remaining

DNF clause parameters. This approach is also suitable for implementing *SQL LIMIT* functionality, as it might stop searching for valid records once it has reached the desired results limit.

5.3 Resource Elastic Sorting

We propose and implement resource elastic variants of both the linear and merge sorter. For the linear sorter, we implement module alternatives, while for our large utility merge sorter, we implement both module alternatives and module composability.

5.3.1 Sort Module Alternatives

Linear Sort Module Alternatives

The linear sort phase of our sorting design benefits from the increase of local scratch-pad memories to allow for the building of larger sorted sequences (L in Figure 4.7). As described in Section 4.4.3, increasing L results in a reduction of the total sort runtime. We synthesize module alternatives for our linear sort with two different utilities: 1) $L = 512$, and 2) $L = 1024$. This enables them to input arbitrarily ordered data and produce sorted sequences of 512 or 1024 sorted records. *Both module alternatives work at line speeds (producing one sorted record per clock cycle).* At the implementation target clock speed of 300MHz and a 512-bit datapath, this corresponds to a peak sort throughput of 19.2GB/s if all fields can be used.

Merge Sort Module Alternatives

The merge sort phase of our sorting design benefits from the increase of local on-chip memory to accommodate more merge stages and results in more simultaneously merged sequences (E in Figure 4.7). As described in Section 4.4.3, increasing E minimizes the total sort runtime. We evaluate three utility configurations for our merge sorter: 1) $E = 32$, 2) $E = 64$, and 3) $E = 128$. They enable the simultaneous merging of 32, 64, or 128 presorted sequences of data. As described in Section 4.4.7, we have implemented the presented sorter kernel for up to $E = 2048$. However, it utilized a full Virtex FPGA, while in this system we have limited FPGA resources available. *All three module alternatives achieve the 300MHz target frequency in our system and produce one sorted record per clock cycle.*

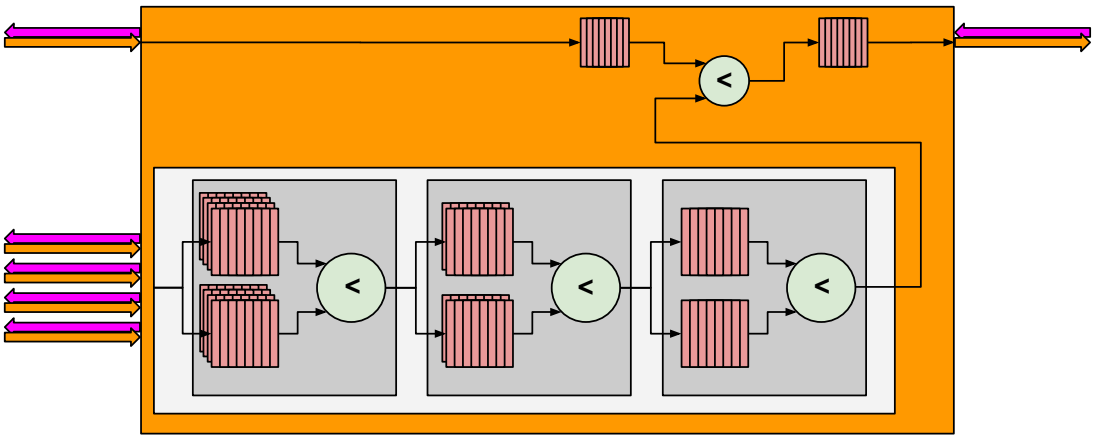

Figure 5.3: Resource elastic merge sort module that can be composed to increase utility. Each module has its own set of input DSPI ChannelIDs to enumerate virtual streams. All modules use $ChannelID = 0$ to pass intermediate sorted data for resource elastic integration.

5.3.2 Merge Sort Composing

We implement a composable merge sort module. To do this, we add an additional merge stage in our module (see Figure 5.3). The additional merge stage is also coupled with two FIFOs in order to decouple inputs and outputs. This is important in order to implement our proposed credit system (see Section 3.3.1). The latency between credit allocation and data receiving can vary arbitrarily depending on the targeted system. It will usually be negligible if only one PR slot is utilized and there are no stream links between PR slots/FPGAs. However, increasing the sizes of these buffers can be used for the elimination of any latency impact on the globally merged stream (i.e., $ChannelID = 0$). Meanwhile, the sorted sequences can be sourced locally when utilizing multiple FPGAs.

The first-most placed merge module does not utilize this additional merge stage, but all subsequent modules utilize it to merge their local output with the global sorted output. Then, the number of the merged sequences is the total sum of the utilities of all placed merge modules.

In our proposed design, one of the virtual input channels of the first of the composed merge modules must be on $ChannelID = 0$. This is because this virtual stream is then used to pass intermediate sorted data between modules (see Figure 5.4). The consecutive merge modules can actually use any arbitrary allocation of the available (in our case 1,024) virtual channels.

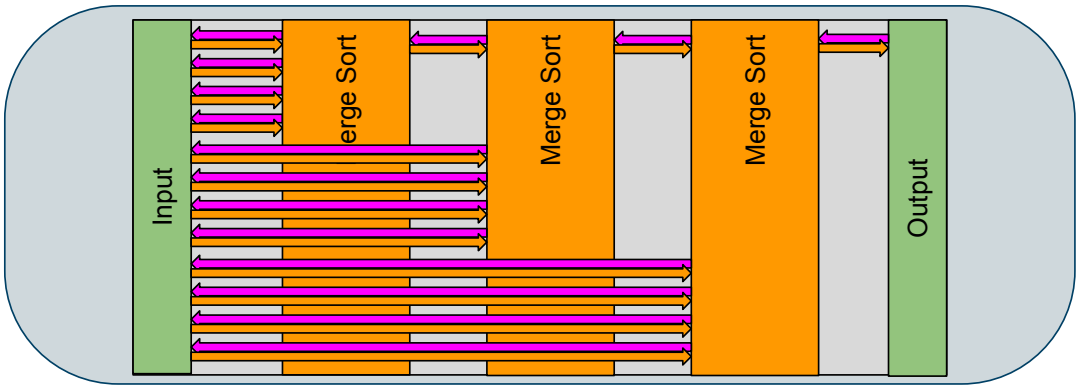

Figure 5.4: Merge sort integration with DSPI. The purple and orange arrows represent the data and instruction ports of the utilized virtual channels. The modules input many virtual channels and merge them into one. Each module has its own set of input DSPI ChannelIDs to enumerate virtual channels. All consecutive modules use $ChannelID = 0$ to pass intermediate sorted data for resource elastic integration.

Smallest resource footprint	Sequence size (L)	LUT	LUTM	Flip-Flop	BRAM
BDMMBDMDMM	512	9,531	3,635	9,113	35.5
BDMMBDMDMMBDMM	1024	14,269	5,603	10,739	70.5

Table 5.2: Linear sort module resource requirements.

5.3.3 Evaluation

Stable sorting is key for database acceleration since it is required for sorting tables for more than one field. Both sorters and all their module alternatives implement stable sorting.

Module alternatives

The implementation results of both the linear and the merge sorters demonstrate that they are bound by the number of allocated BlockRAMs. Table 5.2 shows the resource requirements of the linear sorter. Notice that while the second implementation provides twice the utility, it maps to only a 40% longer resource string. It requires twice the amount of BlockRAMs and can map to a string with one additional BlockRAM column (B).

Table 5.3 shows the resource requirements of the merge sorter. Similarly to the linear sorter, the resource string size scales sublinearly with utility. Again, this is due to the module being bound by the amount of the allocated BlockRAM columns.

Smallest resource footprint	Number of ways (E)	LUT	LUTM	Flip-Flop	BRAM
MBDMDMM	32	5,835	3,108	7,159	17.5
BDMMBDMDMM	64	5,922	3,084	7,471	34.5
BDMMBDMDMMBD	128	6,719	3,374	7,886	68

Table 5.3: Merge sort module resource requirements.

The throughput impact of the utility of our sorters is presented in Section 4.4.3. The presented module alternatives enable us to trade-off between allocated FPGA resources and achieved sort throughput.

Module composing

Due to being BlockRAM bound, as the allocated string for our merge sort module increases, the LUT(M) utilization decreases. This is why we implement the composing FIFOs using FPGA distributed memory (LUTM). This results in a free (in terms of FPGA resources) composing functionality (except for the smallest module alternative, which is LUT-bound).

Composing multiple of our merge sort modules increases the total merge utility achieved. In fact, we can compose using any arbitrary combination of merge sort module alternatives together. *The achieved merge utility is the sum of the merge utilities of all composed modules.* The presented 64-way merge sorter can be placed 3 times in the ZCU102 FOS PR slot, to achieve a total of $64+64+64=192$ sequences merged at once.

5.4 Resource Elastic Joins

We propose and implement resource elastic module alternatives for our many-to-many merge join module. As shown in Figure 4.11 b), the merge join module uses two local scratchpad memories (represented logically as FIFOs) to search for join matches. As described, FIFO A is only used to hide latencies between credit allocation and data arrival. However, FIFO B is traversed in a fully data-dependent manner and data can be overwritten if the number of consecutive colliding keys is larger than FIFO size. This is a rarely occurring situation if considering real-world data. However, it is important to maintain data correctness, thus it implements a fallback solution of flushing and repeating input data sequence. In such a case, performance is largely sacrificed when

Smallest resource footprint	FIFO A depth	FIFO B depth	LUT	LUTM	Flip-Flop	BRAM
BDMDMM	512	512	4,138	1,483	4,692	16
BDMMBDMDMM	512	1536	4,499	1,563	4,751	32
BDMMBDMDMMBD	1024	3072	4,939	1,723	4,813	64

Table 5.4: Join module resource requirements.

flushing the previous modules up to the DMA module, recalculating addressing and re-streaming the data. The depth of the FIFOs in our merge join module is an example of a computational capacity with an undesirable fallback solution. We can trade-off between this compute capacity and FPGA resources. We synthesize three module alternatives that vary the depth of these buffers. *All three module alternatives work at line speeds (computing one join candidate pair per clock cycle).* When no consecutive rows hold colliding keys, the join module operates at line speeds. Upon each unique key collision, the module requires two additional empty clock cycles (one to find the end of the collision sequence and one to roll back the pointers in FIFO B for re-match). Thus the data throughput is data dependent but in our experiments achieves near peak stream throughput.

5.4.1 Evaluation

Table 5.4 shows the utilization of the three considered module alternatives. The module is bound by the allocated BlockRAMs (except the smallest module alternative, which is LUT-bound). The presented sizes of FIFO B show the number of record chunks that can have key collisions before the module initiates repeat sequence. However, this only happens in scenarios of many-to-many joining. Most often the smallest module will suffice, as most real-world problems implement one-to-one or one-to-many relations.

5.5 Conclusion

Resource elasticity is a technique to provide alternative execution plans to the run-time scheduler which significantly improves placement flexibility, often resulting in improved effective performance. This chapter introduced the two principles that form resource elasticity: 1) building module alternatives, and 2) composing modules to implement larger functionality. While the first principle has similarities to the approaches in FOS [117], our composable modules are a new approach for the resource elasticity

concept. Related work has been targeting monolithic problems with forced OpenCL workgroup parallelism [114, 117]. However, these practices are not directly applicable to different problems and execution requirements. Dynamic Stream Processing is an example of a dynamically built runtime where the execution pipeline can be of arbitrary complexity. Additionally, the problem of database query execution is not a monolithic problem with data-dependent operators (e.g., sort) that eliminate any naive parallelism. This thesis proposes module composing to combine the operation of multiple module instances and achieve larger utility and throughput. This comes at the cost of more module instances, thus results in the trading off between accelerator utility/throughput and resource cost. *This allows the runtime system to spend extra resources at that part of the problem that will yield overall best performance for the given cost target.*

We showed how to apply the proposed accelerator composing to our module library, which allows us to maximize utility and performance for our cost target. For example, while related work utilizes 8-way merge sorting [135], we proposed 64-way PR merge sorter (see Section 4.4). However, utilizing our resource elastic composing we can place three PR modules sequentially in a ZCU102 FOS slot and achieve 192-way merge sorting (24$\times$ more sequences merged at once than related work).

We also show that the two resource elastic techniques can work together with cases where we compose the final PR slot configuration by utilizing different module alternatives. *Applying resource elastic techniques to our modules allows the runtime system to trade-off between resource requirements, throughput, utility, and computational capacity*. Providing this resilience to a runtime scheduler allows for the identification of the most beneficial execution strategies, resulting in improved performance for a runtime-only known problem and resource target. The proposed resource elastic modules have been prototyped and evaluated in Chapter 6.

Chapter 6

System Prototype and Evaluation

In this chapter, we describe and evaluate the concepts, benefits and overheads of the integration of our DSPI and our resource elastic module library into a prototype system. We aim to answer the following questions:

- What is the overhead of partial runtime reconfiguration when building the dynamic execution pipeline from our resource elastic module library?
- How to physically implement our partially reconfigurable modules interfaced with our DSPI using existing tools for recent FPGA architectures and what is the resulting performance?
- What are the resulting properties and performance of the full system and how does it compare with standard software query execution?

To answer these questions, we first quantify the FPGA reconfiguration overheads for each of our modules, since this is a key component in our cost models to be used by the runtime scheduler (see Section 6.1). Then we utilize PR tools to build our module library into bitstreams and we evaluate the resulting performance (see Section 6.2). The chapter ends with a case study targeting a standard workload (TPC-H [108]) that evaluates the achieved benefits from integrating our custom DSPI, modules, and resource elasticity into a capable system that accelerates query execution (see Section 6.3).

6.1 Partial Reconfiguration Speed

Accurate cost-modelling is key for efficient scheduling and performance maximization. The required model does not only accommodate parameters such as throughput

and utility but also costs such as resource requirements and reconfiguration overheads. While we have described the achieved throughput, utility, and resource requirements of the presented modules, we also need to study the partial reconfiguration overheads. To enable that, we need to look into how Xilinx UltraScale+ FPGAs reconfigure the fabric.

6.1.1 Fabric Configuration

Xilinx UltraScale+ FPGAs implement the runtime bitstream configuration by writing to individually addressable frames [88, 125]. We know the size of these frames thus we can accurately calculate the configuration overhead of modules. Configuring a frame is done by writing 93 32-bit words, thus each frame contains 372 Bytes of configuration data [125]. FPGA configuration is uniform across the device based on the configured resource type. A single clock region height of resources results in a fixed number of configuration frames:

CLB : CLBs implement the slices holding LUTs and Flip-Flops. A single column of clock region height implements 60 CLBs and requires 16 frames of configuration data which is about 6KB.

Wire interconnect : The wire interconnect implements routing multiplexers in the middle of each of our atomic double-columns. Single clock-height column implements 60 interconnect switch matrices and requires 76 frames of configuration data which is about 28KB.

BlockRAM : Single clock-height column implements 12 BRAM36Ks and requires 6 frames of configuration data for configuration and routing which is about 2.2KB. Additionally, the BlockRAM's internal data can be initialized through the configuration by using 256 additional frames (which results in 172% of the configured BlockRAM contents).

DSP : Single clock-height column implements 24 DSPs and requires 8 frames of configuration data which is about 3KB.

Additionally, every configuration burst also has a fixed overhead of 515 words [125]. Using this quantification of the bitstream sizes per resource footprint, we calculate the configuration runtimes. Writing frames can be done by utilizing the Internal Configuration Access Port (ICAP) [123], which achieves a theoretical configuration speed of

800 MB/s [88] and about 770 MB/s in practice when using Xilinx ZU9EG FPGA (the same chip as used in our system prototype) [87]. Table 6.1 shows the resulting frame sizes and configuration speeds for various module footprints.

6.1.2 Bitstream Compression

Bitstream compression is often implemented by simply omitting the empty frames in a certain bitstream configuration. Since frames can be written arbitrarily, the configuration ports can skip writing certain frames. In our system, an anticipated example is the expensive BlockRAM content frames. If a module does not use BlockRAMs or does not have initial values stored, all of the content frames can be skipped. For example, writing an empty B column to implement route-through mode of our custom DSPI (see Section 6.2.3) results in 0.332ms configuration runtime. However, considering we do not need to use any of the BlockRAMs when we skip writing the content frames, we write only 196 frames. This reduces the configuration data from 263KB to only 72KB, thus reducing the configuration time to 0.094ms (3.5× faster). The same principles can be applied to modules that don't use initial data in their buffers (which is the case currently for all of our accelerators) and modules that don't use BlockRAMs whatsoever (e.g., filter).

6.1.3 Module Configuration

Using the bitstream knowledge from Section 6.1.1, we calculate exact configuration speeds for the proposed accelerators in this thesis. All of the modules hold no initial BlockRAM data, which lets us compress their bitstreams (see Section 6.1.2). Table 6.2 shows the resulting bitstream frames and configuration runtimes for various module alternatives. The configuration uses ICAP at 200MHz [123, 88]. It is possible that in the future the configuration port's speed can be increased by increasing its operating frequency. On older FPGA families, the configuration ports on FPGAs have achieved very large operating frequencies (550MHz on Xilinx Virtex-5) [36], however, this topic is not yet sufficiently researched for our target device family (Xilinx UltraScale+).

Our evaluation results show low configuration overhead times for our modules. Most of the considered module alternatives configure in runtimes less than 1 millisecond. Considering a use case where we want to reconfigure all modules in our PR region except our DMA module (which remains the same for all runs of our system). In that

Resource footprint	Logic and Routing Frames	BlockRAM Content Frames	Bitstream size, Bytes	Configuration time, ms
M	216	0	80,352	0.103
B	196	512	263,376	0.332
D	200	0	74,400	0.096
BD	396	512	337,776	0.425
DM / MD	416	0	154,752	0.196
MB	412	512	343,728	0.432
MM	432	0	160,704	0.203
BDM	612	512	418,128	0.525
DMD	616	0	229,152	0.289
DMM / MDM	632	0	235,104	0.296
MBD	612	512	418,128	0.525
MMB	628	512	424,080	0.533
BDMD	812	512	492,528	0.618
BDMM	828	512	498,480	0.626
DMDM	832	0	309,504	0.389
DMMB / MBDM / MMBD	828	512	498,480	0.626
MDMM	848	0	315,456	0.397
BDMDM	1028	512	572,880	0.719
BDMMB	1024	1024	761,856	0.955
DMDMM	1048	0	389,856	0.490
DMMBD / MBDMD	1028	512	572,880	0.719
MMBDM	1044	512	578,832	0.726
BDMDMM / DMMBDM	1244	512	653,232	0.819
BDMMBD	1224	1024	836,256	1.048
MBDMDM / MMBDMD	1244	512	653,232	0.819
BDMMBDM	1440	1024	916,608	1.148
DMMBDMD	1444	512	727,632	0.912
MBDMDMM	1460	512	733,584	0.920
MMBDMDM	1460	512	733,584	0.920
BDMMBDMD	1640	1024	991,008	1.241
DMMBDMDM	1660	512	807,984	1.013
MMBDMDMM	1676	512	813,936	1.020
BDMMBDMDM	1856	1024	1,071,360	1.342
DMMBDMDMM	1876	512	888,336	1.113
BDMMBDMDMM	2072	1024	1,151,712	1.442

Table 6.1: The speed for partial reconfiguration for modules depends on resource footprint. All footprints with the same resources inside (but different order) have the same configuration data size and speed.

Module (alternative) [Resource footprint]	Configuration Frames	Bitstream size, Bytes	Configuration time, ms
Filter (DNF=8, CMP=1) [MDMM]	848	315,456	0.397
Filter (DNF=8, CMP=4) [DMMBDMD]	1444	537,168	0.674
Filter (DNF=16, CMP=2) [DMDMM]	1048	389,856	0.490
Filter (DNF=16, CMP=4) [MBDMDMM]	1460	543,120	0.681
Filter (DNF=32, CMP=1) [DMDMM]	1048	389,856	0.490
Filter (DNF=32, CMP=2) [DMMBDMD]	1444	537,168	0.674
Filter (DNF=32, CMP=4) [BDMMBDMDMM]	2072	770,784	0.966
Linear Sort (L=512) [BDMMBDMDMM]	2072	770,784	0.966
Linear Sort (L=1024) [BDMMBDMDMMBDMM]	2900	1,078,800	1.351
Merge Sort (E=32) [MBDMDMM]	1460	543,120	0.681
Merge Sort (E=64) [BDMMBDMDMM]	2072	770,784	0.966
Merge Sort (E=128) [BDMMBDMDMMBD]	2468	918,096	1.150
Merge Join (FIFO 512; 512) [BDMDMM]	1244	462,768	0.581
Merge Join (FIFO 512; 1536) [BDMMBDMDMM]	2072	770,784	0.966
Merge Join (FIFO 1024; 3072) [BDMMBDMDMMBD]	2468	918,096	1.150
DMA Module [BDMDMDMMBDMM]	2488	925,536	1.159

Table 6.2: The speed for partial reconfiguration for modules depends on resource footprint. Modules and their alternative utility parameters are described in Chapters 4-5. Configuration times are reported for default FOS ICAP configuration speeds [88].

case, all possible configuration schedules require a maximum of 2.9 milliseconds to reconfigure. In contrast, we can calculate the configuration time of a whole FPGA, which is the case with static designs. The bitstream size of a Xilinx XCZU9EG (ZCU102's FPGA) is 26.5MB [125] (19.2 without the BlockRAM contents). Configuring the entire FPGA requires 33.141 (24.094) milliseconds. This is $8.3 - 11.4\times$ larger configuration overhead when compared to placing our small efficient modules.

6.1.4 Module Relocation

The tool BitMan [89, 88] provides methods for relocating rectangular segments of bitstreams. We use BitMan to relocate our synthesized module alternatives and their resource footprint variants to their final placement destinations. However, the overheads of module relocation are relatively high when considering the time for FPGA fabric configuration. BitMan reports an average relocation function call time of 13ms [88], while most modules reconfigure in less than 1ms with default speeds. Considering that we place multiple small modules, ideally, we prefer to avoid this overhead. This can be achieved by relocating the modules offline, prior to runtime. This increases the size of our bitstream library, however, our modules are of relatively small bitstream size (between 300KB and 1.1MB) when compared to the system memory that most FPGA systems provide. Additionally, most footprint variants have only three possible destination placements (see Figure 4.2), thus the bitstream library does not grow substantially in size. Using this approach, we can avoid module relocation overhead at runtime.

6.2 Partially-Reconfigurable Module Library

In order for our modules to be stitchable, we need to define not only the functionally of our DSPI but also the physical interface. This is implemented by binding the wires of the interface to selected positions. For this purpose, we define the physical connections of DSPI's bus signals to utilize specific wires along the vertical axis. Similarly to Vesper [118], we utilize the horizontal wires of size 2 along the east and west edges of our modules. The resulting module bitstreams implement the module's kernel in a bounding box with the DSPI interface implemented as floating (unconnected) wires on the east and west borders of the module.

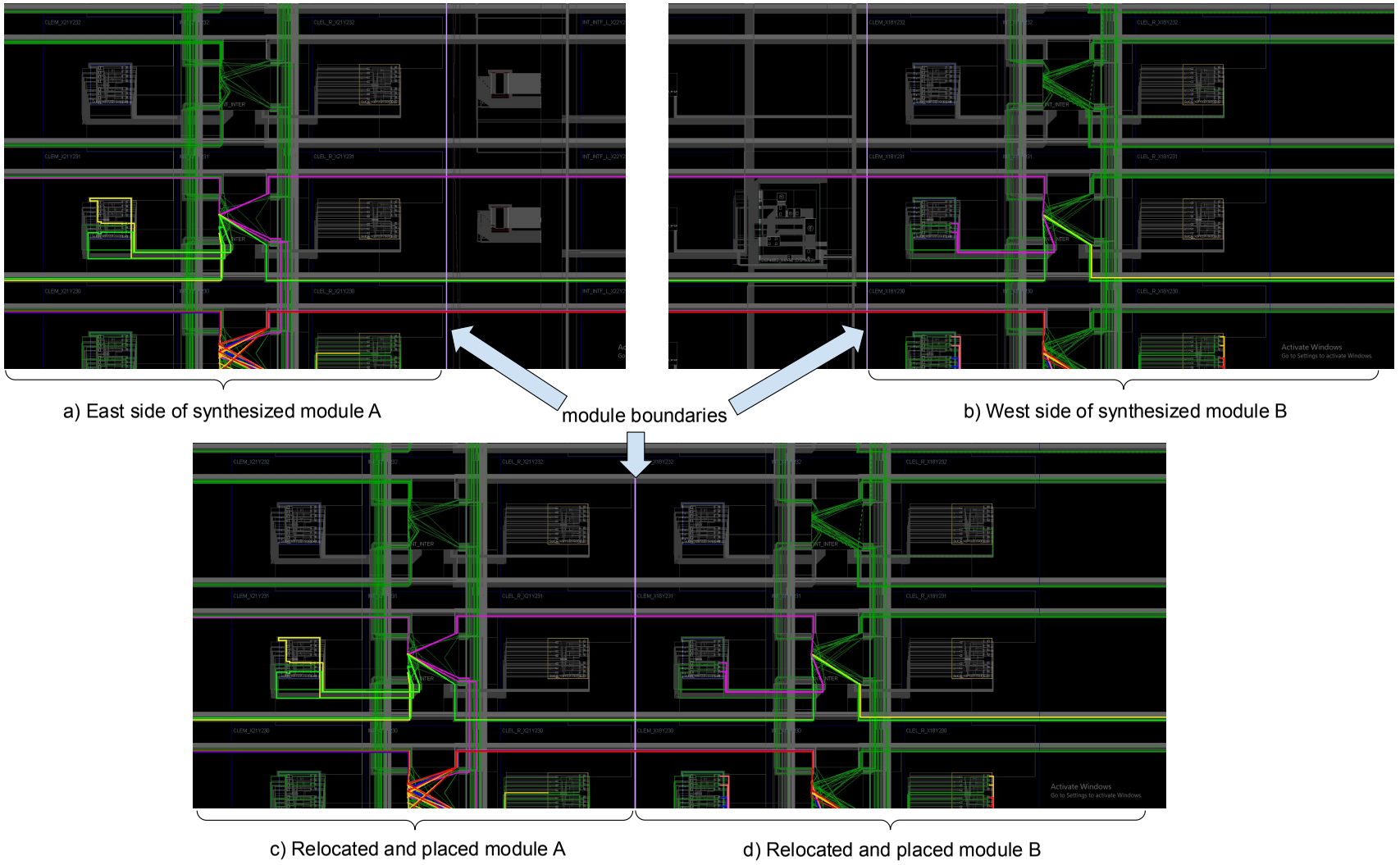

Figure 6.1: Physical definition of DSPI. Figures a) and b) show the east and west sides of two synthesized modules. Then c) and d) show relocation of the two modules to be placed next to each other. The physical definition of DSPI ensures that the two modules get logically connected.

Figure 6.1 shows the anticipated behaviour: The two modules in Figure 6.1 a-b can be relocated and placed next to each other. As shown in Figure 6.1 c-d, when they are placed next to each other, their interface wires automatically connect based on their bound placement during synthesis.

6.2.1 Implementing the Stichable Module Library

In order to synthesize modules to be stitchable, we use a series of tools and our DSPI functional and physical definitions. The implementation process involves the following steps:

- We synthesize the targeted module alternative using Vivado 21.1 in Project mode. This ensures that vendor tools apply their synthesis optimisations. This also provides us with the resource requirements (i.e., LUTs, BRAMs, and DSPs) which is used to define the size of module bounding boxes (i.e., resource footprint).
- We fix the positions of the interface wires using TCL commands that are generated by GoAhead [13] to ensure correct external connectivity.
- We forbid the use (block) of all possible wires leaving the module, except the already defined interface wires. This is done using the TedTCL library [118]. This ensures that no internal wire of the module will ever have segments outside of the module's rectangular bounding box.
- We place and route the design using Vivado 20.1. Using the vendor tools ensures optimal resource and wire allocation, and also precise timing analysis.

Most steps execute quickly (minutes) with the most time-demanding step being the blocking of the wires around the module (requiring about 30 minutes). For the place and route, we use two external banks of sources and sinks for the interface wires (see Figure 6.2). This helps place the internal logic of the module close to the bounding box border, thus improving achieved setup timing. The resulting synthesized module (see the rectangular bounding box in the middle of Figure 6.2) can then be cut and relocated using BitMan [89] to placement positions in the reconfigurable region that provide the resource footprint of the module. We can then apply this synthesis process to the designed modules for our module library. Figure 6.3 shows three module bitstreams for three different resource footprint targets. It can be seen that independent of module placement or resource footprint, all modules implement the same physical interface on their east and west borders.

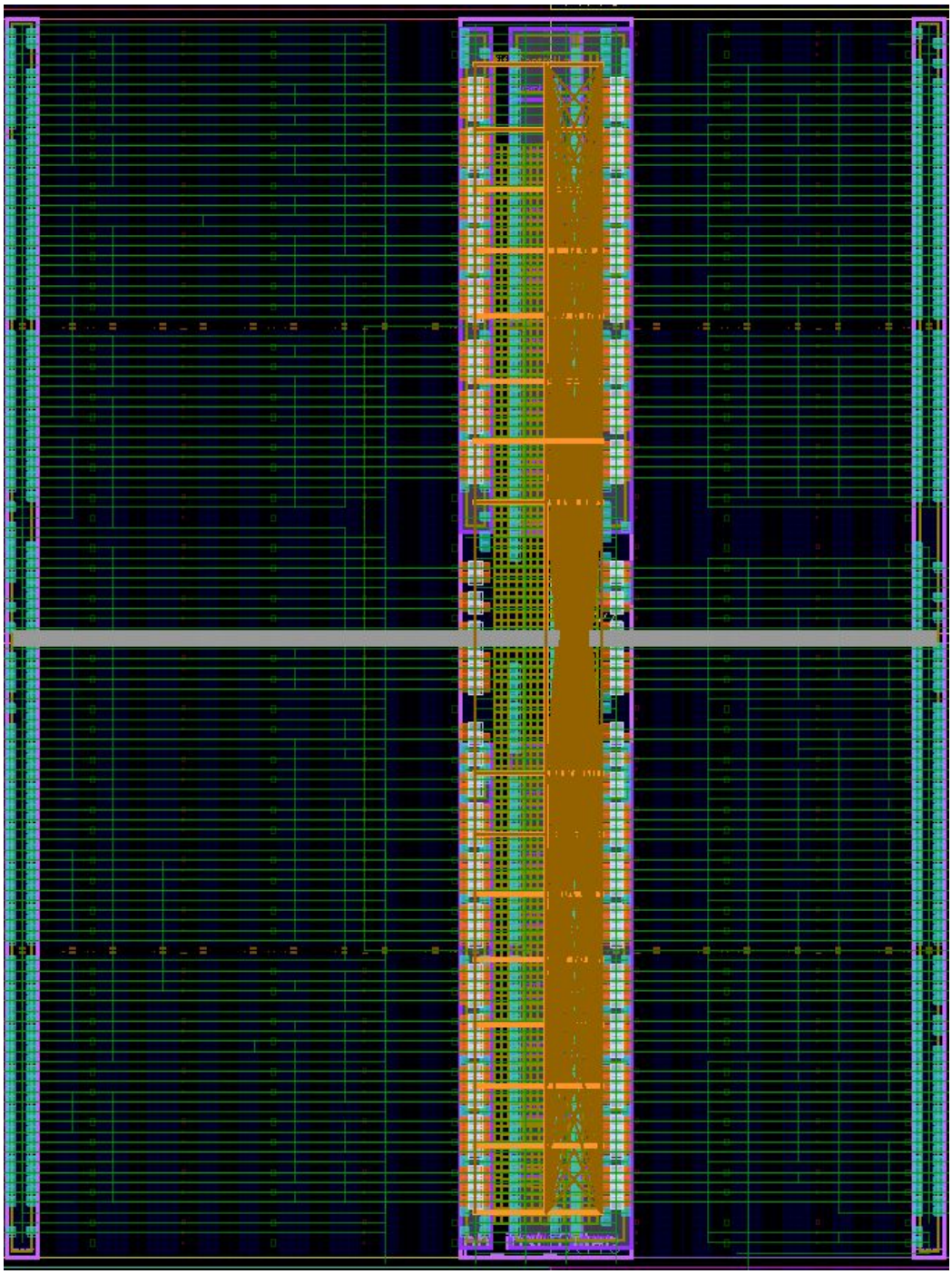

Figure 6.2: Filter (DNF=8, CMP=1) synthesized as a partially-reconfigurable module with module footprint MDMM . The filter module is implemented in the rectangular box in the middle. On the left and right sides of the figure, two banks of source and destination LUTs are placed to act as neighbour modules during implementation. While some of the interface wires look 'scrambled' outside of the PR module, at the border of the module all interface wires are constrained at exact positions (which is also visible in the figure).

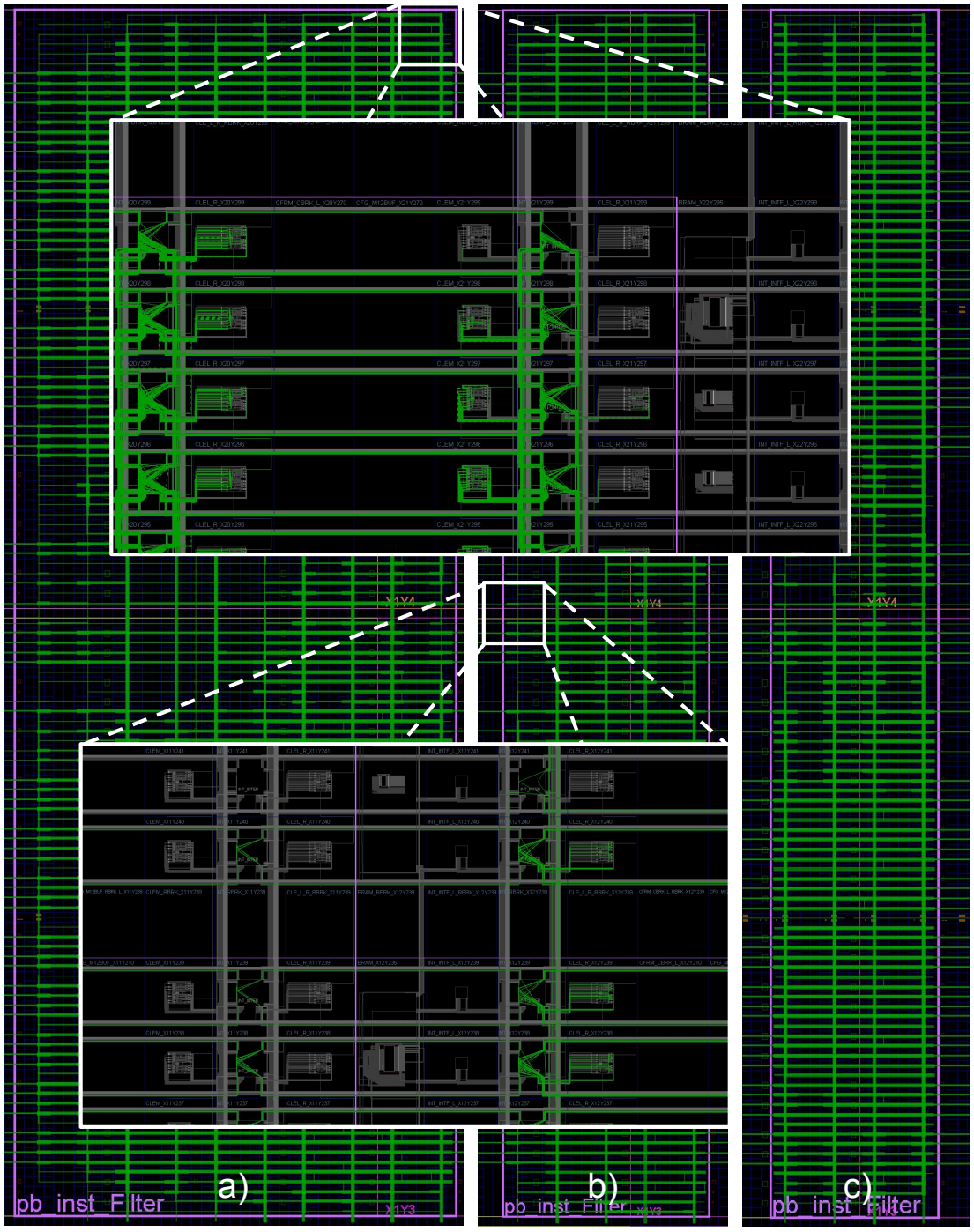

Figure 6.3: Example bitstream results: a) Filter (DNF=32, CMP=4) synthesized at BDMMBDMDMM , b) Filter (DNF=16, CMP=2) synthesized at DMDMM , c) Filter (DNF=8, CMP=1) synthesized at MDMM . The modules are implemented in a bounding box that contains all routing and logic, except the interface wires that are on the east and west borders.

6.2.2 Throughput

The example DSPI configuration (see Section 3.3.2) implements a 512-bit datapath width. All our modules synthesize successfully with that configuration and meet timing constraints set for 300MHz. Running timing analysis on the filter modules shows that when constrained to 300MHz, their placed and routed implementation achieves 351-375MHz (different depending on the module alternative). The identified critical paths are global signals such as enable/reset that also have high fanouts. State-of-art FPGA accelerator solutions (such as Xilinx Alveo U200/250 or Intel D5005 boards) provide 64-bit 2400MT/s DDR4 memories. These memories provide 512-bit interfaces running at 300MHz to the FPGA fabric. *Running our system results at 19.2 GB/s streaming throughput and the ability to saturate two DDR4-2400 memories (one for inputs and one for outputs).* Future implementations of the system could target wider datapath such as 1024-bits, however, global signals will require additional inserted pipeline stages to meet our timing targets.

6.2.3 DSPI in Atomic Resource Columns

Although implementing module resource footprint variants and module alternatives significantly minimizes external fragmentation, there might still be a need for empty resource columns between placed modules. This case requires special consideration, as when a resource column is empty, it still is required to route the DSPI stream through. For this purpose, we implement modules that route DSPI through. They are of atomic width for all resource footprint possibilities (M, B, D) and can be used to bridge holes that result from external fragmentation. Figure 6.4 a shows the routing in our M atomic module. The DSPI's physical definition utilizes specific wires such that the FPGA's interconnect can choose to either route the wire to a Flip-Flop or route-through on the exact same wire.

If the distance between two modules is significantly large, then the long implemented wire chains may not meet our tight timing constraints for 300MHz. In these cases, the system needs to insert pipeline stages on the DSPI. For this purpose, we need to also implement our atomic module with a pipeline stage for the DSPI signals. Figures 6.4 b-c show the routing in our proposed atomic modules with pipeline stages. They implement either a pipeline stage on the east-to-west stream or the west-to-east stream.

Additionally, the scheduler can implement execution pipelines that do not utilize

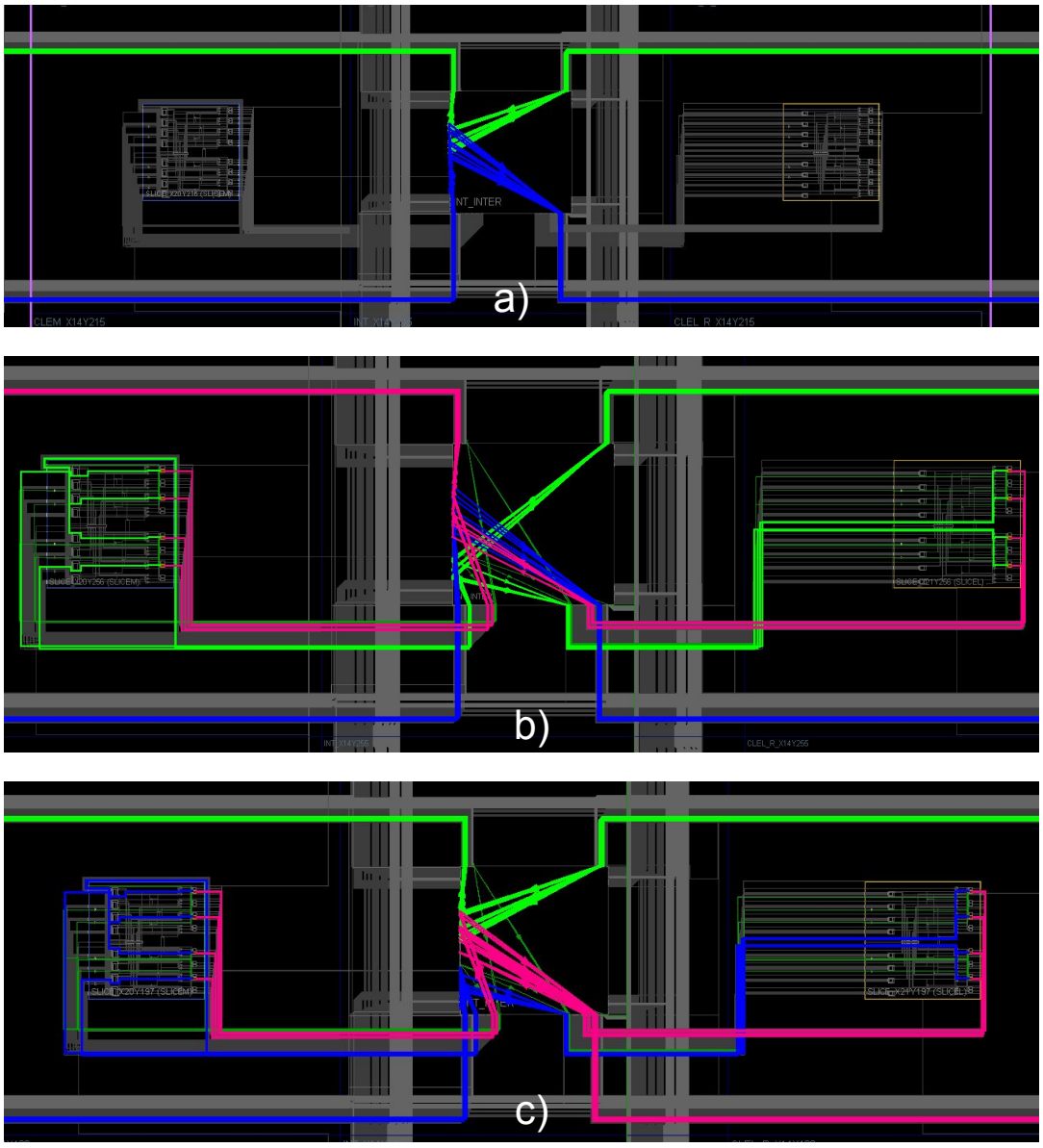

Figure 6.4: DSPI route-through in atomic resource columns: the interface wires can route through directly (a) or by first going through a pipeline stage (b-c). The green and blue busses show the wires of streaming in east-west or west-east direction respectively. The red wires show the interface path after a pipeline stage (in b-c).

the full PR slot. In such cases, placing the required processing modules next to our DMA module might leave a large empty region on the other side of the slot. To minimize configuration overhead we also provide an atomic module for early stream direction reversing. That module does not implement any wire connections on its west border, but only routes the input to the output on its east border.

6.3 Case Study: TPC-H

For detailed evaluation, we use the TPC-H decision support benchmark due to its broad industry-wide relevance and queries that are more complex than most OLTP transactions [108]. From all the queries of the TPC-H benchmark, Q19 is the most intensive for expression calculations. The filter operation is complex and consists of 28 unique integer and string compares and a Boolean evaluation that simplifies to 24 DNF clauses. Other than strong benchmarking of expression calculation (see Listing 6.1), Q19 also has medium complexity for aggregation and join handling [17]. It is an ideal test case to show the co-operation of the most used database operators (filter, join, arithmetic, sort, aggregate) in the proposed Dynamic Stream Processing system. Moreover, due to its complexity, it is the only query that Xilinx Vitis Database Library is unable to execute when utilizing two Xilinx Alveo U280 FPGA boards [133]. Therefore, by showing that our system can execute this query, we demonstrate the versatility of our approach. While Q19 is a difficult query to execute, it is straightforward to parse [17], thus, we do not artificially benefit from our manual query parsing in this system prototype.

This case study targets a dual PR slot in the FPGA Operating System (FOS) [115, 116] on a Xilinx ZCU102 board.

The Stream Processing pipeline is clocked at 300 MHz, which is the highest supported frequency for the Xilinx Zynq AXIs (see Section 2.1.3). The DMA module is connected to a High-Performance AXI port, which provides bandwidth of 4.8 GB/s write and 4.8 GB/s read (see Section 3.1). We generate four versions of TPC-H with four different Scale Factors (SF) of *0.01* (9.8MB), *0.1* (97.6MB), *0.3* (292.5MB), and *1* (975MB) to evaluate the performance scalability of our system with respect to data size. To obtain software baseline performance, we use a powerful machine comprising Intel Core i7-4930K processor with 64 GB DDR3-1333 main memory utilizing four CPU memory channels. The machine operates using Ubuntu 18.04.4 LTS and uses PostgreSQL 10.15 for TPC-H execution.

```
select
  sum(l_extendedprice* (1 - l_discount)) as revenue
from
  lineitem,
  part
where
  (
    p_partkey = l_partkey
    and p_brand = 'Brand#12'
    and p_container in ('SM CASE', 'SM BOX', 'SM PACK', 'SM PKG')
    and l_quantity >= 1 and l_quantity <= 1 + 10
    and p_size between 1 and 5
    and l_shipmode in ('AIR', 'AIR REG')
    and l_shipinstruct = 'DELIVER IN PERSON'
  )
  or
  (
    p_partkey = l_partkey
    and p_brand = 'Brand#23'
    and p_container in ('MED BAG', 'MED BOX', 'MED PKG', 'MED PACK')
    and l_quantity >= 10 and l_quantity <= 10 + 10
    and p_size between 1 and 10
    and l_shipmode in ('AIR', 'AIR REG')
    and l_shipinstruct = 'DELIVER IN PERSON'
  )
  or
  (
    p_partkey = l_partkey
    and p_brand = 'Brand#34'
    and p_container in ('LG CASE', 'LG BOX', 'LG PACK', 'LG PKG')
    and l_quantity >= 20 and l_quantity <= 20 + 10
    and p_size between 1 and 15
    and l_shipmode in ('AIR', 'AIR REG')
    and l_shipinstruct = 'DELIVER IN PERSON'
  );
```

Listing 6.1: TPC-H Query 19 [108] represents a complex query utilizing the most widely used database operations, but with large operation complexity. It consists of 28 unique integer and string compares and a Boolean evaluation that simplifies to 24 DNF clauses. It challenges all of the most common database operators for acceleration: filter, join, arithmetic, aggregate, and sort (sort is needed to implement the merge join operation).

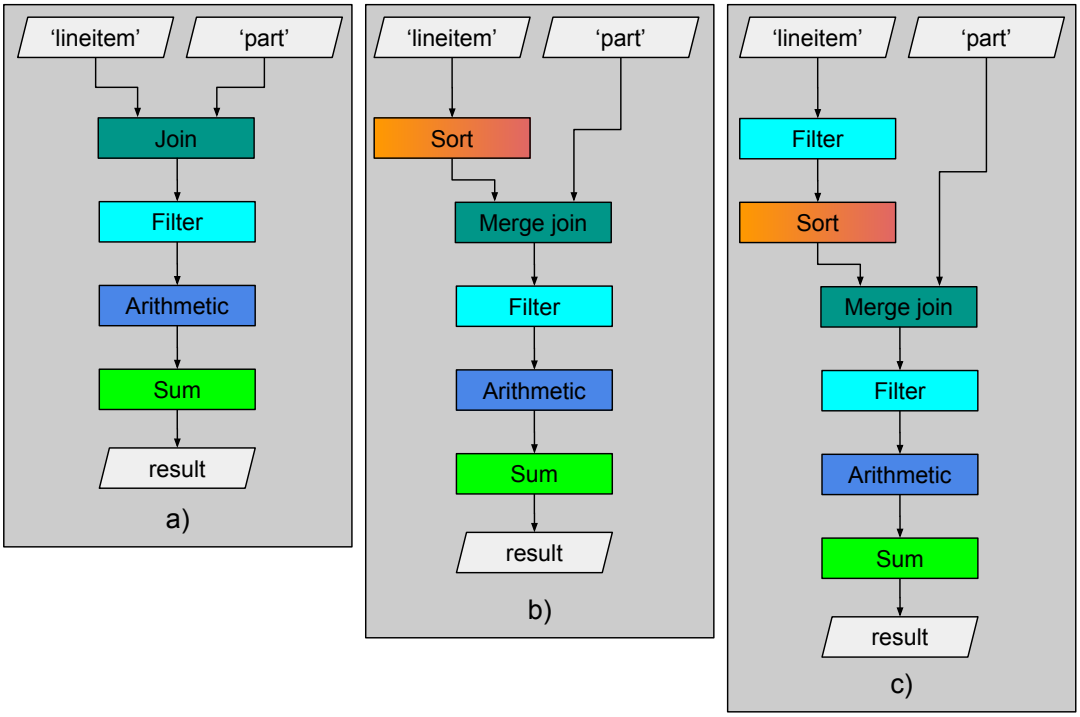

Figure 6.5: TPC-H Query 19 execution plans. All operations in the figure represent a logical view, while the physical module implementation depends on the exact operation parameters and data sizes at runtime. The runtime scheduler can tailor the query plans for hardware acceleration: a) initially the basic parsed operations graph does not define a type of join, b) for the join operator we use a merge join, which results in the need to sort 'lineitem' first, and c) sort is an expensive operation, but we can move the data filtering for table 'lineitem' to be processed before the sort to minimize sorting data.

6.3.1 Building the Execution Pipeline

The query plan in Figure 6.5 a represents a logical operator graph. The runtime host system will decide the physical implementation and order for these operators. This is done in iterations:

- Operators such as filter, arithmetic, sum each have their own module implementations and can be logically represented in a hardware acceleration graph. However, the join module has different algorithms that can be used (merge join or hash join). Selecting our merge join operator as a hardware accelerator target also imposes the need to sort one of the join keys, thus we need to add a sorting step as shown in Figure 6.5 b. The sort operator in our system is physically implemented by two different phases that are data-dependent but has no alternative algorithms, thus we do not have to evaluate it further in our logical graph.

- The next critical step is to move any data minimization forward since our system is memory-bound. Additionally, the sort operation is an expensive one, thus we need to reduce the data beforehand. In this query, we have a filter operation that is applied on records after the joining of the two tables. We can extract the valid data states for table 'lineitem' from the post-join filter into a new pre-sort filter as shown in Figure 6.5 c. This way we will decrease the number of tuples that need to be sorted.

We build the execution pipeline as follows:

- We start in order by placing the pre-sort filter together with the first phase for sorting (a linear sorter) (see Figure 6.6 a). Sorting is a natural barrier for reconfiguration, due to the inability to produce sorted records prior to observing all input data. The sorter input data is of unknown size since it is an output of a filtering operation. Therefore, we cannot extend the initial hardware run with other operations, so we utilize the remaining FPGA resources by placing our largest (in terms of both resources and utility) linear sort module alternative.

- After the first run, the filtered data is written back to DDR in pre-sorted sequences of 1024 records each. The system needs to finish the sorting process using merge sort accelerators and at this point, the system knows the exact size of the data to sort. Since the final merging run we consider merging of up to 64 sorted sequences (see Figure 6.6 c), we need to minimize the sequences to 64 or less prior to that run (see Section 4.4.3). Thus, if the number of sequences that need merging is more than 64 (like in the SF=1 case), the system places merge-sort-only intermediate runs (see Figure 6.7). In general, since sort is a blocking operator which forces partial reconfiguration, we can utilize the scheduler to optimize the following merge runs according to the available resources and the runtime-known compute problem (here defined as the number of sorted sequences that require merging).

- Then we continue towards the join operation. We place a final merge sort module (E=32 or E=64 depending on intermediate data), then we join, and finally, we filter the joined data (see Figure 6.6 b-c). The two configurations implement the same operations, but by using our module alternatives, one of them manages to avoid the configuration of three double columns (resulting in 0.3 milliseconds reduction). For the post-join filter module, we duplicate the data

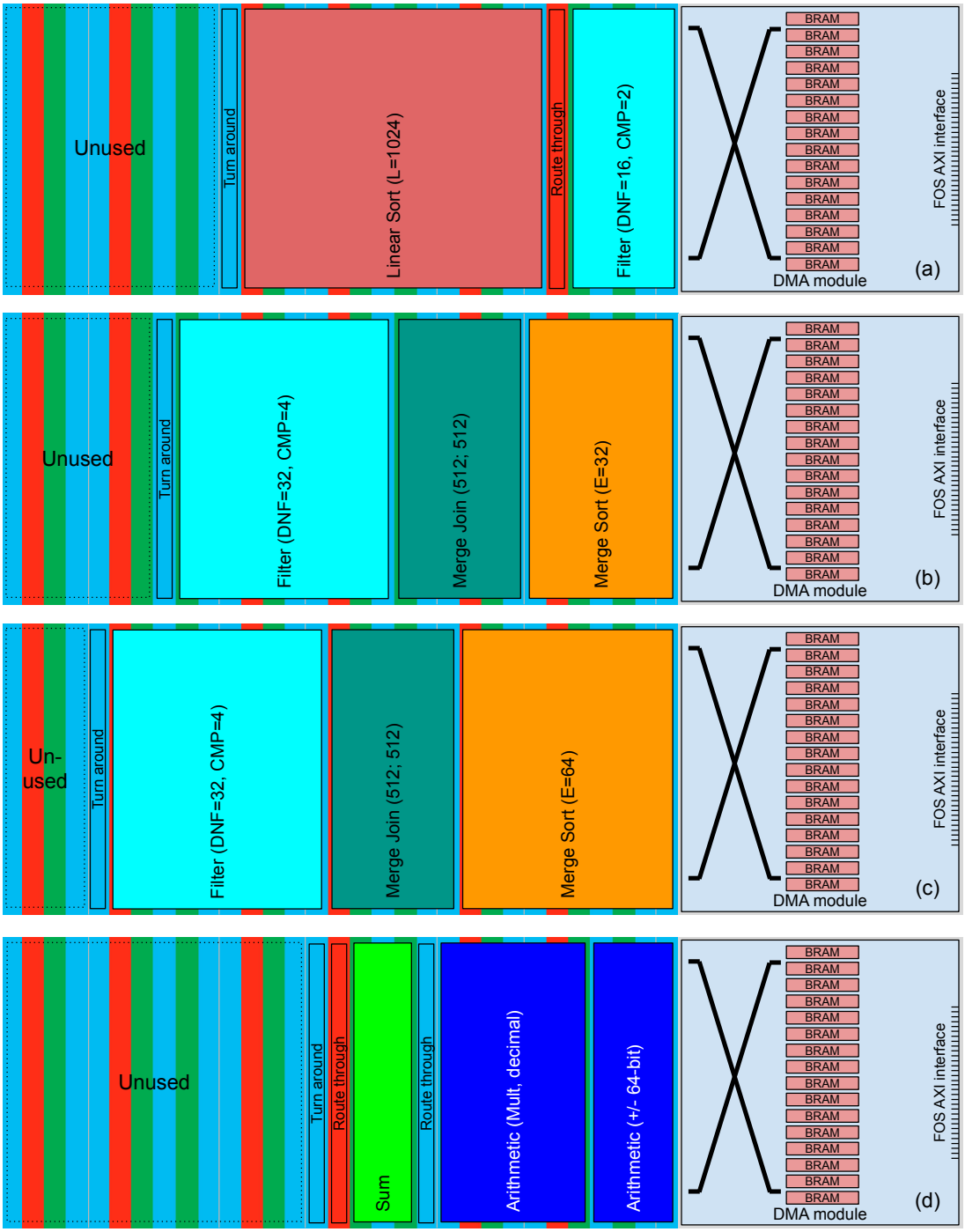

Figure 6.6: Execution pipeline for TPC-H Query 19 acceleration. The execution of the query is done in multiple phases. First, we filter and linear sort (a) the input table 'lineitem'. Then we can have intermediate merge sort runs depending on the resulting data size after the filter. Then we use either E=32 or E=64 merge sort (b-c) to finish the sorting, join with table 'part' and filter all resulting data. In the final run (d), we solve the query's arithmetic and execute the final aggregation.

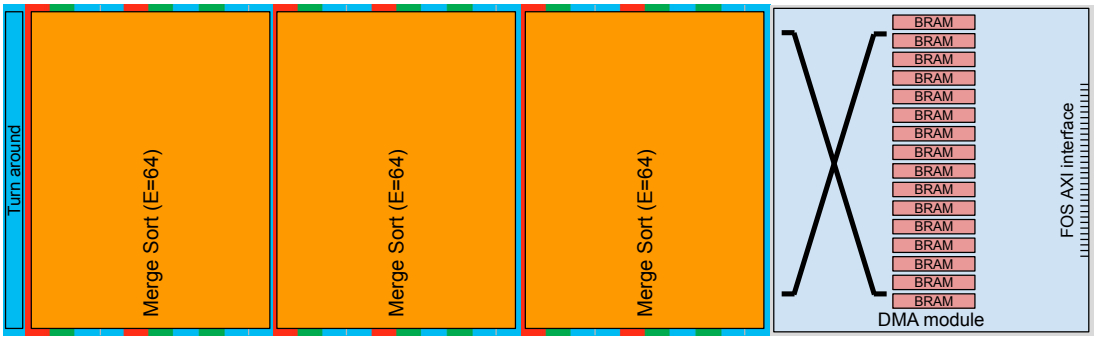

Figure 6.7: TPC-H case study: merge sort execution pipeline. In cases we need to merge many sorted streams, we can place three of our resource elastic merge sort modules to achieve $E = (3 \times 64) = 192$.

field *p_container* to accommodate for the twelve unique compare strings reference values. However, we manage to do this without a throughput reduction, since we do not reserve new chunks in the streamed records (see Section 5.2).

- Finally, we place the arithmetic operations followed by a sum aggregation. The arithmetic multiplier in this query multiplies two decimal fields and thus needs to do a division at the end of the operation resulting in a larger module than the other arithmetic modules.

At the end of each stage (writing back intermediate data to DDR), the programmable crossbar in our DMA module omits the unnecessary data fields to improve performance. For example, after we join and finish all data filtering, we only store two data fields per tuple back to DDR to be used for the final arithmetic and aggregation. The initial input data size in the SF=1 case is 975MB, while the intermediate data for the final FPGA run is only 2KB (121 records of 16 bytes each).

6.3.2 Evaluation

Table 6.3 presents the measured runtimes of our proposed streaming system and our baseline PostgreSQL running on an Intel i7 machine with four memory channels. Our baseline machine achieves 1.8× higher performance in PostgreSQL than Xilinx's Intel Xeon server base case, and even higher performance than hardcoding the query using C++ [132]. We also implemented a Performance Monitoring Unit (PMU) inside our DMA module that tracks runtime and AXI transactions. This allows us to do cycle-accurate time analysis, which is important so we can observe any overheads

TPC-H	Proposed System ZCU102 FPGA Board 1× DDR4-2400		Xilinx Baseline [132] Intel Xeon E5-2690 v4		Our Baseline Intel i7-4930K 4× DDR3-1333
Scale Factor	PR, ms	Execution, ms	PostgreSQL 9.6, ms	C++, ms	PostgreSQL 10.15, ms
SF=0.01	6	2.076	-	-	22.8
SF=0.1	6	21.707	-	-	73
SF=0.3	6.3	62.034	-	-	188
SF=1	7.5	207.837	983	596	538

Table 6.3: Evaluation results of prototype system on TPC-H Query 19 [108]

in the system. For our baseline results, we execute every TPC-H configuration in our PostgreSQL ten times and consider only the minimal achieved runtime.

Latency

Our latency results for the FPGA acceleration measure both FPGA acceleration runtime and FPGA partial reconfiguration latency (see Figure 6.8). In all four configurations, our proposed system achieves between 2.46× and 2.62× faster runtime when compared to software execution. We analyze the two main components that increase our execution latency:

FPGA reconfiguration : As described in Section 6.3.1, our system implements the query in three FPGA runs for *SF=0.01, 0.1, 0.3*, and uses one additional FPGA run for *SF=1.0* for merge sorting. This additional run adds an additional reconfiguration stage. However, in that case, two consecutive runs use a placed merge sort module as the first accelerator, thus the second of these runs can avoid the overhead of replacing the merge sorter. The total reconfiguration latency in our four runs is between 6.027 and 7.492 milliseconds. When correlating these numbers to our total execution runtimes, we observe that for smaller databases the reconfiguration overhead is significantly larger than the actual execution runtime. This was expected and discussed throughout this thesis: FPGA acceleration is best targeted when the problem size is sufficiently large to amortise system overheads. More precisely, in the *SF=0.01* case, our reconfiguration overhead amounts to 74.3% of the total execution runtime. This overhead is largely overcome by increasing the database size: in the *SF=1* case, our reconfiguration overhead equates to only 3.4% of the total execution runtime.

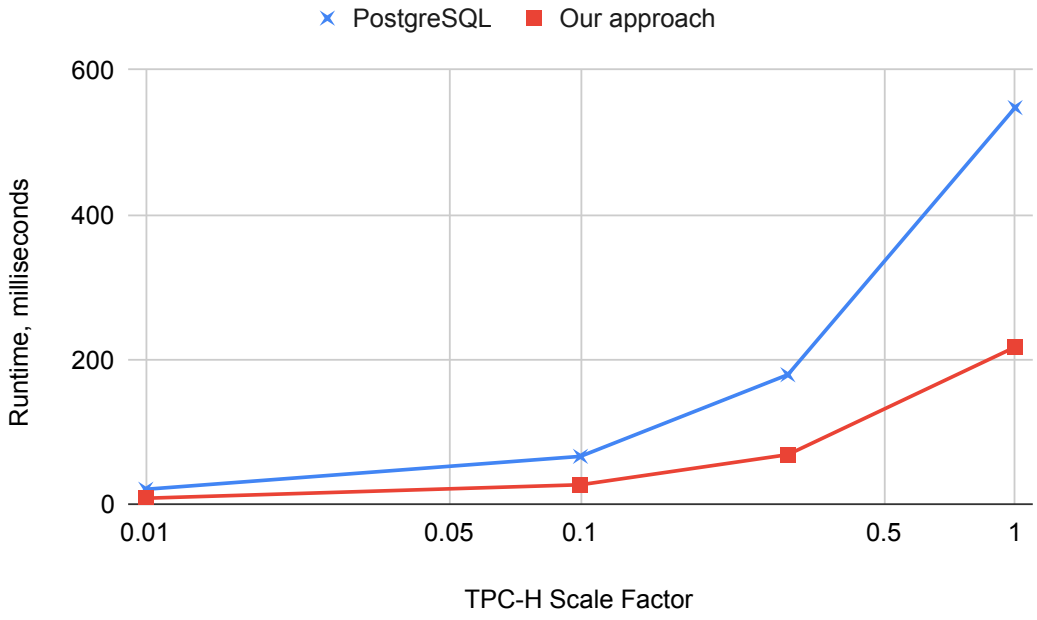

Figure 6.8: The total execution latency including both computation and FPGA fabric reconfiguration overheads.

Acceleration latency : As discussed in Section 6.3.1, it is important to drop all data that will not be needed for the following runs prior to DDR writing. We observe the large impact of this optimisation. The largest of our two input tables is table 'lineitem' which equates to 96.0% of the total initial input data. The first FPGA run executes our initial filtering on that table and results in 95% of the total execution latency. This is because, at the end of this initial run, most of the data is omitted (both invalid records and unneeded data fields from the remaining valid records) which results in less-demanding following runs.

Throughput

All our FPGA configurations are memory-bound and achieve the maximal available throughput as measured in our quantitative memory analysis (see Section 3.1). The currently proposed DMA module provides prefetching that is able to accommodate for a total of 1024 clock cycle latency, which is sufficient to decouple the streaming accelerators from the memory subsystem, while fully utilizing the available memory bandwidth. However, database records and their intermediate forms tend to implement

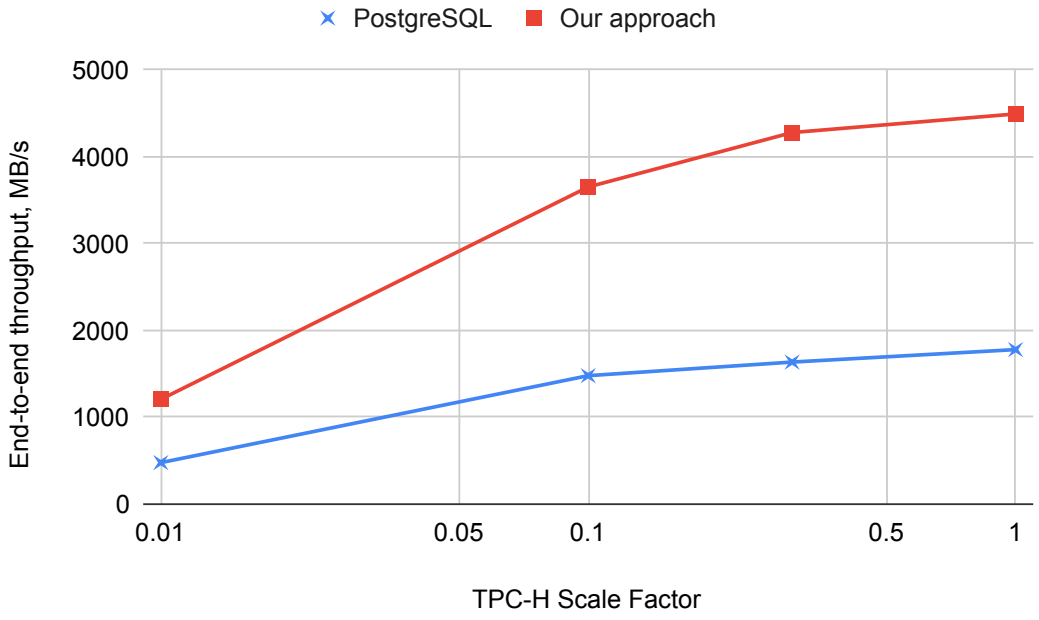

Figure 6.9: TPC-H case study: end-to-end throughput. We measure end-to-end throughput as the ratio of the input table size and the total query execution time.

random data sizes, which can result in inefficient positioning in memory. For example, the input records from table 'lineitem' implement 39 32-bit integers. To achieve near peak memory throughput utilization we use our memory subsystem evaluation to program our DMA module that can access an arbitrary number of records as a single packed burst. In our case study, we pack the records in groups of 8, 16, or 32 (depending on their size) to force memory access burst alignment to larger boundaries and achieve near-peak utilization.

To analyze the throughput between our proposed system and our PostgreSQL baseline, we evaluate end-to-end throughput. We measure this metric as the ratio of the total size of the two input tables and the total query execution time (including FPGA reconfiguration). Figure 6.9 shows the tendency of end-to-end throughput increase with respect to database size. The proposed approach in this thesis provides better execution performance and high scalability with the increase of database size.

In our case study, our utilized system has limited memory resources (both in terms of size and throughput) compared to our baseline machine. In Figure 6.10 we show the ratio for memory throughput utilization. It considers the achieved end-to-end throughput in our evaluation and the peak memory throughput of the two target machines. On

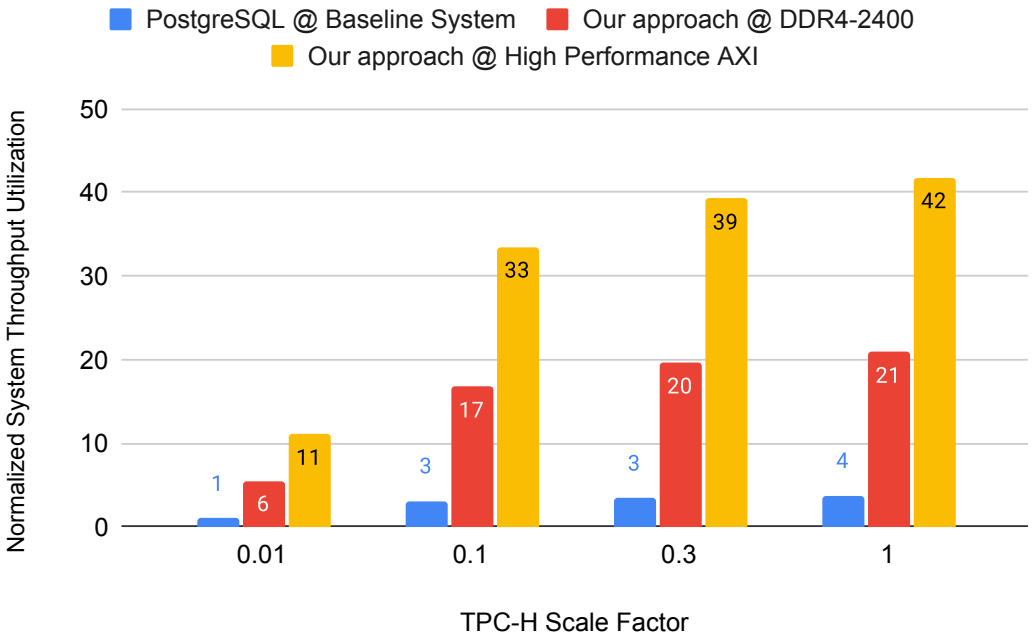

Figure 6.10: TPC-H case study: system throughput utilization. We measure this utilization as the normalized ratio of end-to-end throughput to peak system bandwidth (the peak of the DDR4-2400 is 19.2 GB/s, High-Performance AXI port is 9.6 GB/s, while for our PostgreSQL baseline system is 42 GB/s).

the ZCU102 board, while the maximal DDR throughput is 19.2 GB/s, our accelerators are constrained by the maximal 9.6 GB/s throughput of the 128-bit AXI operating at 300MHz. Depending on the configuration, our system achieves $5.4 - 11.4\times$ better utilization of the available DDR memory throughput. This efficient utilization of the available memory throughput is enabled by the large utility of the proposed accelerators. It showcases the importance to maximize the amount of practical work performed each time the data is streamed through the FPGA.

6.3.3 Summary and Related Work

While the Xilinx Vitis Database Library utilizing dual Xilinx Alveo U280 FPGA boards does not support the execution of TPC-H Query 19 [133], we show that using our dynamic approach we can accelerate this query. Since our system operates at 19.2 GB/s, it is memory-bound in ZCU102. It saturates the available 9.6 GB/s, which is limited by the Zynq UltraScale+ High Performance AXI port (see Section 2.1.3). The large throughput, efficiency, and scalability are highlighted better when we compare to

related work. Ziener et al. [135] utilize an FPGA board with 12 GB/s, but achieve a stream throughput of only 2 GB/s (6× less than peak). Similarly, AxleDB [98] utilizes an FPGA board with 15 GB/s DDR bandwidth, but achieves a processing throughput of only 3.2 GB/s (4.7× less than peak). Related works cannot saturate DDR bandwidth since the proposed systems are not scalable to wide datapaths and high frequencies (see Section 4.1.4).

Throughout this thesis, we have presented and quantified the scalability and utility of the proposed accelerators. For example, our DNF filter module prototype achieves 19.2 GB/s stream throughput and is guaranteed to process a chunk every clock cycle (see Section 4.3). Cost scales linearly when increasing the datapath width and stream throughput (see Section 4.3.4), while other approaches have exponential growth for cost [121, 58, 105]. For example, the filter units in [58] achieve up to 0.297 GB/s per filter unit, and their results are combined using a $k:1-bit$ look-up-table. In order to achieve our 19.2 GB/s stream throughput using the filter units approach, the system will need at least $k = 19.2/0.297 \approx 65$ filter units. This results in a $65:1-bit$ look-up-table, requiring 2^{65} bits of on-chip memory, which is infeasible.

6.4 Conclusion

This chapter proposed methods for building our accelerators as PR-capable modules and an analysis of the FPGA configuration overheads. Our PR modules connect when placed next to each other through their DSPI implementations. When building the runtime pipeline we can minimize the configuration overheads since the size of the used portion of the PR slot can be decreased by early termination using a turn-around atomic module. We show that the configuration overhead for building the execution pipeline is relatively low at 0.5-3ms depending on the configuration.

We evaluate the built accelerators from our module library and they all meet 300MHz operating frequency when synthesized for a wide datapath of 512 bits resulting in 19.2 GB/s throughput. *Running our system at 19.2 GB/s streaming throughput enables it to saturate two standard DDR4-2400 64-bit memories.* Our accelerators are highly scalable and have the potential to extend to wider datapath widths and still meet time requirements if additional pipeline stages are inserted.

We implement a case study targeting TPC-H [108]. While Xilinx Vitis Database Library utilizing dual Xilinx Alveo U280 FPGA boards does not support the acceleration of TPC-H Query 19 [133], we can execute the query using our system. Our system

is memory-bound and achieves near-peak throughput as expected from our quantitative memory analysis (see Section 3.1). We compare against a baseline case utilizing a powerful desktop machine with PostgreSQL and our approach achieves 2.5× higher performance and more than 5× better utilization of available system memory throughput.

Chapter 7

Conclusion

7.1 Summary

In this thesis, we proposed methods, concepts, and implementations to constructing a dynamic stream processing system that implements the principle of resource elasticity to optimize execution pipelines at runtime. This was demonstrated for database acceleration where acceleration services (e.g., for sorting) had been served using partial reconfiguration for using all the currently available resources for speeding up execution. The proposed system demonstrates: 1) a capable protocol for interfacing dynamic modules (see Chapter 3) that reduces control wires by 13% compared to the previous state-of-art (see Section 3.3.7), while enabling extended functionality such as direct memory-mapped registers in dynamic modules (see Section 3.3.4) and our accelerators can arbitrarily utilize multiple virtual channels for communication (up to thousands); 2) a library of large utility resource elastic modules that can be composed together to implement higher logical functions while meeting our target streaming throughput of 19.2GB/s; 3) 2.5× faster execution and 5× better memory utilization in TPC-H when utilizing a portion of a ZCU102 FPGA compared to PostgreSQL running on a high-end Intel Core-i7 machine.

To do this, we introduced a new methodology for implementing thin and tall partially reconfigurable modules by adopting design factors such as minimizing their vertical wiring (see Section 4.1). The presented concept includes the major increase in flexibility and utility for achieving a more versatile module library. We provided accelerator designs with good quality of results for the most used functions in database systems. Module alternatives allow the runtime scheduler to tailor the execution pipeline

to the accelerated problem, minimizing external fragmentation and increasing performance. Additionally, composable modules allow the scheduler to maximize utility and flexibility at runtime for optimal performance to resource requirement trade-offs (see Section 5.1). Our module library implements the most widely used database accelerators.

We demonstrate our module design methodology with our filter module (see Section 4.3). The module utilizes standard Boolean expression representations (DNF) to provide flexible and efficient hardware design. Achieving the integration of many different compare requirements and Boolean expression functions together in one module instance results in a large utility. For example, comparing that to the state-of-art approach [135] in dynamic stream processing of using small atomic modules for each operation, we achieve 13× fewer resource columns required to filter TPC-H Query 19, while enabling the filter to be executed with a single pass of the data through the chip, thus also maximizing effective performance (see Section 4.3.4).

To expand the benefits of FPGA acceleration, we identify expensive operations in database acceleration and design solutions for achieving high performance. One such operation is sorting, thus we propose and implement a highly optimized large utility merge sorter module (see Section 4.4) that, depending on the allocated FPGA resources, achieves up to thousands of merged sequences at once, which is key for performance gains (Section 4.4.3).

We analyze the benefits of resource elastic techniques and demonstrate their application to various modules in our system (see Section 5.1). For example, we can increase the utility and overall performance of our sort modules (see Section 5.3) or increase the computational capacity for data filtering (see Section 5.2). Resource elastic scheduling can also be used together with partial reconfiguration for blocking operators that split the activity of a processing pipeline in multiple phases (e.g., sorting) to allow composing optimized sub-pipelines for each phase.

To enable the seamless integration of our modules, we propose the Dynamic Stream Processing Interface (DSPI). We thoroughly examine the possible requirements of streaming applications for data rates, stream organization, and accelerator functionalities to achieve a highly efficient and versatile interface fit for dynamic stream processing modules. It implements a credit-based system where PR modules allocate tokens, which enables precise dataflow control leading to maximized scratchpad utilization. The interface is adaptive and most functionalities are selectively omitted by every module to eliminate logic overheads, with only a single key feature being mandatory (see

Section 3.3.4). The interface minimizes wire overheads, while drastically extending the achieved capabilities when compared to state-of-art research [118]. It also enables key functionality such as relaxed software integration through memory-mapped registers inside the dynamic modules.

To adapt our system for deployment in managed environments, we target an external standard AXI interface [115, 116]. Thus, we design a DMA module (see Section 4.2) that is used to decouple our DSPI from a set of standard AXI master and slave interfaces. This module provides address handling, data permutation to abstract these challenges from our streaming accelerators. Additionally, the DMA module provides stream prefetching in order to minimize serve times and maximize effective throughput. We show how different operators can be integrated and orchestrated in our streaming system through the proposed hash join streaming modules that carefully utilize DSPI and our DMA module (see Section 4.5.2). Overall, considering the flow and the instruction set of our DSPI protocol, the proposed DMA module implements a complete abstraction layer.

In our prototype implementation, we build the modules such that they automatically connect their interfaces when placed next to each other at runtime (see Section 6.2.1). To evaluate the system, we implement a case study targeting the standard TPC-H benchmark. Our system in particular allows accelerating the complex TCP-H Query 19, which is commonly discarded in related work (see Section 6.3). We use the resource elastic functionality of our module library to tailor the execution pipeline to the exact query requirements and maximize throughput and resource utilization. We show that our system can be scaled to high I/O datarates and we demonstrated this for matching the peak DDR4 memory data rate. The results from running our prototype system confirm the anticipated high performance and efficient resource utilization (see Section 6.3.2). Additionally, the scalability of the proposed methods and designs in our system allows for future scaling to match the ever-increasing FPGA capabilities.

7.2 Future Work

Next generation Xilinx FPGAs will bring major improvement in memory throughput, and UltraRAM capacity and versatility [130, 129]. The utilization of future FPGA technology will improve our system throughput and the capabilities of the proposed module library, especially modules that benefit from large scratchpads and

buffers (e.g., DMA, merge sort, linear sort). There are also three key identified areas that need additional research and development: 1) software integration, 2) module library extension, and 3) automation of module implementation:

- Software integration:
  - The researched topics provide solutions for the offline module library generation, module interfacing, scalability, utility, and partially their runtime integration and management. However, the integration of our proposed system into DBMS is required in order to provide a complete solution that can be deployed at scale. The research of integrating the proposed hardware into an existing DBMS and exploring heterogeneous scheduling is currently ongoing by PhD candidate Kaspar Mätas [66]. This integration includes the implementation of a scheduler that can find the most optimal execution plans. Such scheduling poses a large complexity due to the wide operator search space in our resource elastic module library. Additionally, such integration will provide software fallback solutions for cases that do not fit well for FPGA acceleration. This integration could pose hardware challenges as well. For example, our DMA module implementation currently realizes a row-store database, while software DBMS might require a column-store-capable DMA module [42]. In fact, the versatility of DSPI also enables column-based operators, which is also a possible topic for future research.
- Module library extension:
  - We propose hardware implementation for the most widely used database operators. However, the SQL standard defines 376 reserved keywords, many of which implement operators [44]. To fully support all operators, our module library needs to be extended with modules such as support for all data types and conversion between them and complex operators such as logarithm and trigonometry operators.
  - Module optimisations can be applied to accelerate certain problems. This can be achieved by research and implementation of specialized versions of our modules and extension of our module library. Examples of such use cases are sorting by multiple keys simultaneously and modules that process multiple packed small records every clock cycle.

- Automation of module implementation:
  - The prototype implementation of our system includes multiple manual operations when building module bitstreams. This does not allow us to fully explore module resource footprint variants. The development of automatic implementation toolflow will allow the completion of the module library with bitstream variants, resulting in improving or eliminating any resource fragmentation in the runtime execution pipeline.

Bibliography

[1] Jasmin Ajanovic. PCI Express (PCIe) 3.0 Accelerator Features. *Intel Corporation*, 10, 2008.

[2] Amazon.com, Inc. AWS EC2 FPGA Development Kit, 2021. `https://github.com/aws/aws-fpga`.

[3] Roberto Ammendola, Andrea Biagioni, Paolo Cretaro, Ottorino Frezza, Francesca Lo Cicero, Alessandro Lonardo, Michele Martinelli, Pier Stanislao Paolucci, Elena Pastorelli, Francesco Simula, et al. The next Generation of Exascale-class Systems: the ExaNeSt Project. In *2017 Euromicro Conference on Digital System Design (DSD)*, pages 510–515. IEEE, 2017.

[4] Andrew G. Kegel. METHOD AND APPARATUS FOR EFFICIENT PROGRAMMABLE INSTRUCTIONS IN COMPUTER SYSTEMS, December 2020. `https://www.freepatentsonline.com/y2020/0409707.html`, US: 20200409707.

[5] ARM Limited. AMBA® AXI and ACE Protocol Specification, 2020. `https://developer.arm.com/documentation/ihi0022/latest/`.

[6] Paul Gustav Heinrich Bachmann. *Die Analytische Zahlentheorie [Analytic Number Theory] (in German)*, volume 2. Teubner, 1894.

[7] Ronak Bajaj. Exploiting DSP Block Capabilities in FPGA High Level Design Flows. *Nanyang Technological University, Singapore*, 2016.

[8] Ayoosh Bansal, Rohan Tabish, Giovani Gracioli, Renato Mancuso, Rodolfo Pellizzoni, and Marco Caccamo. Evaluating the Memory Subsystem of a Configurable Heterogeneous MPSoC. In *Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, page 55, 2018.

[9] KE Batcher. Sorting Networks and Their Applications. In *Proceedings of the Spring Joint Computer Conference*, pages 307–314. ACM, 1968.

[10] Kenneth E Batcher. Bitonic Sorting. *Goodyear Aerospace Corp., Rep. GER-11869*, 1964.

[11] Andreas Becher, Florian Bauer, Daniel Ziener, and Jürgen Teich. Energy-Aware SQL Query Acceleration through FPGA-Based Dynamic Partial Reconfiguration. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8. IEEE, 2014.

[12] Andreas Becher, Daniel Ziener, Klaus Meyer-Wegener, and Jürgen Teich. A Co-Design Approach for Accelerated SQL Query Processing via FPGA-based Data Filtering. In *2015 International Conference on Field Programmable Technology (FPT)*, pages 192–195. IEEE, 2015.

[13] Christian Beckhoff, Dirk Koch, and Jim Torresen. GoAhead: A Partial Reconfiguration Framework. In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, pages 37–44. IEEE, 2012.

[14] Václad E Beneš. Optimal Rearrangeable Multistage Connecting Network. *Bell system technical journal*, 43(4):1641–1656, 1964.

[15] Saman Biookaghazadeh, Ming Zhao, and Fengbo Ren. Are FPGAs Suitable for Edge Computing? In *USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18)*, 2018.

[16] Burton H Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, 1970.

[17] Peter Boncz, Thomas Neumann, and Orri Erling. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 61–76. Springer, 2013.

[18] Andrei Broder and Jorge Stolfi. Pessimal Algorithms and Simplexity Analysis. *ACM SIGACT News*, 16(3):49–53, 1984.

[19] Jared Casper and Kunle Olukotun. Hardware Acceleration of Database Operations. In *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, pages 151–160, 2014.

[20] Surajit Chaudhuri and Umeshwar Dayal. An Overview of Data Warehousing and OLAP Technology. *ACM Sigmod record*, 26(1):65–74, 1997.

[21] Jatin Chhugani, Anthony D Nguyen, Victor W Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. Efficient Implementation of Sorting on Multi-Core SIMD CPU Architecture. *Proceedings of the VLDB Endowment*, 1(2):1313–1324, 2008.

[22] Edgar Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13, 1970.

[23] André DeHon. DPGA-Coupled Microprocessors. In *Proceedings of IEEE Workshop on FPGA's for Custom Computing Machines*, pages 31–39. IEEE, 1994.

[24] André DeHon. DPGA Utilization and Application. In *Fourth International ACM Symposium on Field-Programmable Gate Arrays*, pages 115–121. IEEE, 1996.

[25] Christopher Dennl, Daniel Ziener, and Jurgen Teich. On-the-fly Composition of FPGA-based SQL Query Accelerators using a Partially Reconfigurable Module Library. In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, pages 45–52. IEEE, 2012.

[26] Christopher Dennl, Daniel Ziener, and Jürgen Teich. Acceleration of SQL Restrictions and Aggregations through FPGA-Based Dynamic Partial Reconfiguration. In *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 25–28. IEEE, 2013.

[27] Manish Deo, Jeffrey Schulz, and Lance Brown. Intel Stratix 10 MX Devices Solve the Memory Bandwidth Challenge. *Intel White Paper*, 2016.

[28] Ulrich Drepper. What Every Programmer Should Know About Memory. *Red Hat, Inc*, 11:2007, 2007.

[29] Phil Francisco et al. The Netezza Data Appliance Architecture: A Platform for High Performance Data Warehousing and Analytics, 2011.

[30] Mohsen Ghasempour, Aamer Jaleel, Jim D Garside, and Mikel Luján. DReAM: Dynamic Re-arrangement of Address Mapping to Improve the Performance of

DRAMs. In *Proceedings of the Second International Symposium on Memory Systems*, pages 362–373, 2016.

[31] KAHN Gilles. The Semantics of a Simple Language for Parallel Programming. *Information processing*, 74:471–475, 1974.

[32] Matthias Göbel, Ahmed Elhossini, Chi Ching Chi, Mauricio Alvarez-Mesa, and Ben Juurlink. A Quantitative Analysis of the Memory Architecture of FPGA-SoCs. In *International Symposium on Applied Reconfigurable Computing*, pages 241–252. Springer, 2017.

[33] Graysort Benchmark. Sort Benchmark, 2013. `http://sortbenchmark.org/`.

[34] Nicolae Bogdan Grigore and Dirk Koch. Placing Partially Reconfigurable Stream Processing Applications on FPGAs. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2015.

[35] Robert J Halstead, Bharat Sukhwani, Hong Min, Mathew Thoennes, Parijat Dube, Sameh Asaad, and Balakrishna Iyer. Accelerating Join Operation for Relational Databases with FPGAs. In *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 17–20. IEEE, 2013.

[36] Simen Gimle Hansen, Dirk Koch, and Jim Torresen. High Speed Partial Run-Time Reconfiguration Using Enhanced ICAP Hard Macro. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 174–180. IEEE, 2011.

[37] Scott Hauck. The Roles of FPGAs in Reprogrammable Systems. *Proceedings of the IEEE*, 86(4):615–638, 1998.

[38] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K Govindaraju, Qiong Luo, and Pedro V Sander. Relational Query Coprocessing on Graphics Processors. *ACM Transactions on Database Systems (TODS)*, 34(4):1–39, 2009.

[39] John L Hennessy and David A Patterson. A New Golden Age for Computer Architecture. *Communications of the ACM*, 62(2):48–60, 2019.

[40] Edson L Horta, John W Lockwood, and Sérgio T Kofuji. Using PARBIT to Implement Partial Run-Time Reconfigurable Systems. In *International Conference on Field Programmable Logic and Applications*, pages 182–191. Springer, 2002.

[41] Jonathan Hou and Jan-Erik Schmitt. All change in edge computing: As AMD buys Xilinx and Nvidia acquires Arm, we ask. two industry experts what this could mean for the vision sector. *Imaging and Machine Vision Europe*, (102):12–14, 2020.

[42] S Idreos, F Groffen, N Nes, S Manegold, S Mullender, and M Kersten. MonetDB: Two Decades of Research in Column-oriented Database. *IEEE Data Engineering Bulletin*, 2012.

[43] Intel Corporation. Intel Stratix 10 GX/SX Device Overview, 2020. `https://www.intel.com/content/www/us/en/programmable/documentation/joc1442261161666.html`.

[44] ISO. Information Technology — Database Language — SQL. ISO 9075-1:2016, International Organization for Standardization, Geneva, Switzerland, 2016.

[45] Zsolt István, Gustavo Alonso, Michaela Blott, and Kees Vissers. A Hash Table for Line-Rate Data Processing. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 8(2):1–15, 2015.

[46] M Ito, H Hayashi, and M Ohara. Logic-Saving FPGA-Based Merge Sort on Single Sort Cells (in Japanese). *IPSJ SIG Technical Report*, 2014(18):1–6, 2014.

[47] Ming-Yee Iu, Emmanuel Cecchet, and Willy Zwaenepoel. JReq: Database Queries in Imperative Languages. In *International Conference on Compiler Construction*, pages 84–103. Springer, 2010.

[48] Jonas Julian Jensen. Reconfigurable FPGA Accelerator for Databases. Master's thesis, University of Oslo (Norway), 2012.

[49] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. Shore-MT: A Scalable Storage Manager for the Multicore Era. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, pages 24–35, 2009.

[50] Tim Kaldewey, Guy Lohman, Rene Mueller, and Peter Volk. GPU Join Processing Revisited. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, pages 55–62, 2012.

[51] D Knuth. Sorting and Searching, 2nd edn. The Art of Computer Programming, vol. 3, 1998.

[52] Dirk Koch, Christian Beckhoff, and Jurgen Teich. Recobus-Builder — a Novel Tool and Technique to Build Statically and Dynamically Reconfigurable Systems for FPGAs. In *2008 International Conference on Field Programmable Logic and Applications*, pages 119–124. IEEE, 2008.

[53] Dirk Koch, Christian Haubelt, and Jürgen Teich. Efficient Reconfigurable On-Chip Buses for FPGAs. In *2008 16th International Symposium on Field-Programmable Custom Computing Machines*, pages 287–290. IEEE, 2008.

[54] Dirk Koch and Jim Torresen. FPGASort: A High Performance Sorting Architecture Exploiting Run-time Reconfiguration on FPGAs for Large Problem Sorting. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 45–54, 2011.

[55] Edmund Georg Hermann Landau. Handbuch der Lehre von der Verteilung der Primzahlen [Handbook on the Theory of the Distribution of the Primes] (in German). *Berlin: BG Teubner. X*, 1909.

[56] Alexander Lange. ACCELERATING DATA ANALYTICS-50X FASTER APACHE SPARK RANDOM FOREST CLASSIFICATION. *Xelera Technologies*, 2019.

[57] Jason Lawley. Understanding Performance of PCI Express Systems. *WP350 (v1. 2). Xilinx*, 97, 2014.

[58] Ying Li, Jinyu Zhan, Wei Jiang, Junting Wu, and Jianping Zhu. An fpga based network interface card with query filter for storage nodes of big data systems. In *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 556–561. IEEE, 2020.

[59] Shaoshan Liu, Liangkai Liu, Jie Tang, Bo Yu, Yifan Wang, and Weisong Shi. Edge Computing for Autonomous Driving: Opportunities and Challenges. *Proceedings of the IEEE*, 107(8):1697–1716, 2019.

[60] Gorker Alp Malazgirt, Nehir Sonmez, Arda Yurdakul, Adrian Cristal, and Osman Unsal. High Level Synthesis Based Hardware Accelerator Design for Processing SQL Queries. In *Proceedings of the 12th FPGAworld Conference 2015*, pages 27–32, 2015.

[61] Kristiyan Manev and Dirk Koch. Large Utility Sorting on FPGAs. In *2018 International Conference on Field-Programmable Technology (FPT)*, pages 334–337. IEEE, 2018.

[62] Kristiyan Manev and Dirk Koch. Resource Elastic Database Acceleration. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 363–364. IEEE, 2020.

[63] Kristiyan Manev, Anuj Vaishnav, and Dirk Koch. Unexpected Diversity: Quantitative Memory Analysis for Zynq Ultrascale+ Systems. In *2019 International Conference on Field-Programmable Technology (ICFPT)*, pages 179–187. IEEE, 2019.

[64] Kristiyan Manev, Anuj Vaishnav, Charalampos Kritikakis, and Dirk Koch. Scalable Filtering Modules for Database Acceleration on FPGAs. In *Proceedings of the 10th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*, pages 1–6, 2019.

[65] Susumu Mashimo, Thiem Van Chu, and Kenji Kise. High-Performance Hardware Merge Sorter. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 1–8. IEEE, 2017.

[66] Kaspar Mätas and Dirk Koch. Transparent Integration of a Dynamic FPGA Database Acceleration System. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 365–366. IEEE, 2020.

[67] Iakovos Mavroidis, Ioannis Papaefstathiou, Luciano Lavagno, Dimitrios S Nikolopoulos, Dirk Koch, John Goodacre, Ioannis Sourdis, Vassilis Papaefstathiou, Marcello Coppola, and Manuel Palomino. ECOSCALE: Reconfigurable Computing and Runtime System for Future Exascale Systems. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 696–701. IEEE, 2016.

[68] Meikel Poess et al. New TPC Benchmarks for Decision Support and Web Commerce. *ACM Sigmod Record*, 2000.

[69] Oskar Mencer, Dennis Allison, Elad Blatt, Mark Cummings, Michael J Flynn, Jerry Harris, Carl Hewitt, Quinn Jacobson, Maysam Lavasani, Mohsen Moazami, et al. The History, Status, and Future of FPGAs. *Communications of the ACM*, 63(10):36–39, 2020.

[70] Nooshin Mirzadeh, Yusuf Onur Koçberber, Babak Falsafi, and Boris Grot. Sort vs. Hash Join Revisited for Near-Memory Execution. In *5th Workshop on Architectures and Systems for Big Data (ASBD 2015)*, 2015.

[71] Sparsh Mittal. A Survey of FPGA-based Accelerators for Convolutional Neural Networks. *Neural computing and applications*, 32(4):1109–1139, 2020.

[72] Gordon E Moore et al. *Cramming More Components onto Integrated Circuits*. McGraw-Hill New York, NY, USA, 1965.

[73] Rene Mueller and Jens Teubner. FPGA: What's in it for a Database? In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 999–1004, 2009.

[74] Rene Mueller, Jens Teubner, and Gustavo Alonso. Data Processing on FPGAs. *Proceedings of the VLDB Endowment*, 2(1):910–921, 2009.

[75] Rene Mueller, Jens Teubner, and Gustavo Alonso. Glacier: A Query-to-Hardware Compiler. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 1159–1162, 2010.

[76] Rene Mueller, Jens Teubner, and Gustavo Alonso. Sorting Networks on FPGAs. *The VLDB Journal*, 21(1):1–23, 2012.

[77] Aaftab Munshi. The OpenCL Specification. In *2009 IEEE Hot Chips 21 Symposium (HCS)*, pages 1–314. IEEE, 2009.

[78] Toshio Nakatani, S-T Huang, Bruce W. Arden, and Satish K. Tripathi. K-Way Bitonic Sort. *IEEE Transactions on Computers*, 38(2):283–288, 1989.

[79] Raghunath Othayoth Nambiar and Meikel Poess. The Making of TPC-DS. In *VLDB*, volume 6, pages 1049–1058, 2006.

[80] Eriko Nurvitadhi, David Sheffield, Jaewoong Sim, Asit Mishra, Ganesh Venkatesh, and Debbie Marr. Accelerating Binarized Neural Networks: Comparison of FPGA, CPU, GPU, and ASIC. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 77–84. IEEE, 2016.

[81] Andreas Oetken, Stefan Wildermann, Jurgen Teich, and Dirk Koch. A Bus-based SoC Architecture for Flexible Module Placement on Reconfigurable FPGAs. In *2010 International Conference on Field Programmable Logic and Applications*, pages 234–239. IEEE, 2010.

[82] Oracle Corporation and/or its affiliates. MySQL 8.0 Reference Manual, 2020. `https://dev.mysql.com/doc/refman/8.0/en/`.

[83] Philippos Papaphilippou, Chris Brooks, and Wayne Luk. FLiMS: Fast Lightweight Merge Sorter. In *2018 International Conference on Field-Programmable Technology (FPT)*, pages 78–85. IEEE, 2018.

[84] Philippos Papaphilippou, Chris Brooks, and Wayne Luk. An Adaptable High-Throughput FPGA Merge Sorter for Accelerating Database Analytics. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 65–72. IEEE, 2020.

[85] Philippos Papaphilippou and Wayne Luk. Accelerating Database Systems using FPGAs: A Survey. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 125–1255. IEEE, 2018.

[86] Philippos Papaphilippou, Holger Pirk, and Wayne Luk. Accelerating the merge phase of sort-merge join. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 100–105. IEEE, 2019.

[87] Khoa Pham, Dirk Koch, Anuj Vaishnav, Konstantinos Georgopoulos, Pavlos Malakonakis, Aggelos Ioannou, and Iakovos Mavroidis. Moving Compute towards Data in Heterogeneous multi-FPGA Clusters using Partial Reconfiguration and I/O Virtualisation. In *2020 International Conference on Field-Programmable Technology (ICFPT)*, pages 221–226. IEEE, 2020.

[88] Khoa Dang Pham. *FPGA Virtualisation on Heterogeneous Computing Systems—Model, Tools, and Systems*. PhD thesis, The University of Manchester, Manchester, UK, 2020.

[89] Khoa Dang Pham, Edson Horta, and Dirk Koch. BitMan: A Tool and API for FPGA Bitstream Manipulations. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 894–897. IEEE, 2017.

[90] Khoa Dang Pham, Anuj Vaishnav, Malte Vesper, and Dirk Koch. ZUCL: A ZYNQ UltraScale+ Framework for OpenCL HLS Applications. In *FSP Workshop 2018; Fifth International Workshop on FPGAs for Software Programmers*, pages 1–9. VDE, 2018.

[91] Orestis Polychroniou and Kenneth A Ross. Vectorized Bloom Filters for Advanced SIMD Processors. In *Proceedings of the Tenth International Workshop on Data Management on New Hardware*, pages 1–6, 2014.

[92] Willard V Quine. The Problem of Simplifying Truth Functions. *The American mathematical monthly*, 59(8):521–531, 1952.

[93] Stefan Richter, Victor Alvarez, and Jens Dittrich. A Seven-Dimensional Analysis of Hashing Methods and its Implications on Query Processing. *PVLDB*, 9(3):96–107, 2015.

[94] Ron Savage. Backus-Naur Form: SQL 2003, 2020. `https://ronsavage.github.io/SQL/sql-2003-2.bnf.html`.

[95] Mohammadsadegh Sadri, Christian Weis, Norbert Wehn, and Luca Benini. Energy and Performance Exploration of Accelerator Coherency Port Using Xilinx ZYNQ. In *Proceedings of the 10th FPGAworld Conference*, pages 1–8, 2013.

[96] Makoto Saitoh, Elsayed A Elsayed, Thiem Van Chu, Susumu Mashimo, and Kenji Kise. A High-Performance and Cost-Effective Hardware Merge Sorter without Feedback Datapath. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 197–204. IEEE, 2018.

[97] Makoto Saitoh and Kenji Kise. Very Massive Hardware Merge Sorter. In *2018 International Conference on Field-Programmable Technology (FPT)*, pages 86–93. IEEE, 2018.

[98] Behzad Salami, Gorker Alp Malazgirt, Oriol Arcas-Abella, Arda Yurdakul, and Nehir Sonmez. AxleDB: A novel programmable query processing platform on FPGA. *Microprocessors and Microsystems*, 51:142–164, 2017.

[99] Nabeel Shirazi, Dan Benyamin, Wayne Luk, Peter YK Cheung, and Shaori Guo. Quantitative Analysis of FPGA-based Database Searching. *Journal of VLSI signal processing systems for signal, image and video technology*, 28(1):85–96, 2001.

[100] Valery Sklyarov, Iouliia Skliarova, João Silva, and Alexander Sudnitson. Analysis and Comparison of Attainable Hardware Acceleration in All Programmable Systems-on-Chip. In *2015 Euromicro Conference on Digital System Design*, pages 345–352. IEEE, 2015.

[101] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. Knights Landing: Second-Generation Intel Xeon Phi Product. *Ieee micro*, 36(2):34–46, 2016.

[102] Wei Song, Dirk Koch, Mikel Luján, and Jim Garside. Parallel Hardware Merge Sorter. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 95–102. IEEE, 2016.

[103] Harold S Stone. Parallel Processing with the Perfect Shuffle. *IEEE transactions on computers*, 100(2):153–161, 1971.

[104] Bharat Sukhwani, Hong Min, Mathew Thoennes, Parijat Dube, Balakrishna Iyer, Bernard Brezzo, Donna Dillenberger, and Sameh Asaad. Database Analytics Acceleration Using FPGAs. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 411–420. IEEE, 2012.

[105] Xuan Sun, Chun Jason Xue, Jinghuan Yu, Tei-Wei Kuo, and Xue Liu. Accelerating data filtering for database using FPGA. *Journal of Systems Architecture*, 114:101908, 2021.

[106] Swarm64 and Xilinx. Swarm64 PostgreSQL Accelerator, 2021. `https://www.xilinx.com/publications/solution-briefs/swarm64_solutionbrief.pdf`.

[107] The PostgreSQL Global Development Group. PostgreSQL 13.1 Documentation, 2020. `https://www.postgresql.org/docs/13/`.

[108] Transaction Processing Performance Council. TPC Benchmark™ H Standard Specification, 2018. `http://tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.18.0.pdf`.

[109] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy Transactions in Multicore In-Memory Databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32, 2013.

[110] Takanori Ueda, Megumi Ito, and Moriyoshi Ohara. A Dynamically Reconfigurable Equi-Joiner on FPGA. *IBM Tehnical Report RT0969*, 2015.

[111] Takuma Usui, Thiem Van Chu, and Kenji Kise. A Cost-Effective and Scalable Merge Sorter Tree on FPGAs. In *2016 Fourth International Symposium on Computing and Networking (CANDAR)*, pages 47–56. IEEE, 2016.

[112] Anuj Vaishnav, Khoa Pham, and Dirk Koch. Live Migration for OpenCL FPGA Accelerators. In *2018 International Conference on Field-Programmable Technology (FPT)*, pages 38–45. IEEE, 2018.

[113] Anuj Vaishnav, Khoa Dang Pham, and Dirk Koch. Heterogeneous Resource-Elastic Scheduling for CPU+FPGA Architectures. In *Proceedings of the 10th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*, pages 1–6, 2019.

[114] Anuj Vaishnav, Khoa Dang Pham, Dirk Koch, and James Garside. Resource Elastic Virtualization for FPGAs using OpenCL. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pages 111–1117. IEEE, 2018.

[115] Anuj Vaishnav, Khoa Dang Pham, Kristiyan Manev, and Dirk Koch. The FOS (FPGA Operating System) Demo. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 429–429. IEEE, 2019.

[116] Anuj Vaishnav, Khoa Dang Pham, Joseph Powell, and Dirk Koch. FOS: A Modular FPGA Operating System for Dynamic Workloads. *ACM Trans. Reconfigurable Technol. Syst.*, 13(4), September 2020.

[117] Anuj Ashokbhai Vaishnav. *Modular FPGA Systems with Support for Dynamic Workloads and Virtualisation*. PhD thesis, The University of Manchester, Manchester, UK, 2020.

[118] Malte Vesper. *Dynamic Streamprocessing Pipelines on FPGAs Targeting Database Acceleration*. PhD thesis, The University of Manchester (United Kingdom), 2019.

[119] Michael J Wirthlin and Brad L Hutchings. DISC: The dynamic instruction set computer. In *Field Programmable Gate Arrays (FPGAs) for Fast Board Development and Reconfigurable Computing*, volume 2607, pages 92–103. International Society for Optics and Photonics, 1995.

[120] Alexander Wold, Dirk Koch, and Jim Torresen. Enhancing Resource Utilization with Design Alternatives in Runtime Reconfigurable Systems. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 264–270. IEEE, 2011.

[121] Louis Woods, Zsolt István, and Gustavo Alonso. Ibex: an Intelligent Storage Engine with Support for Advanced SQL Offloading. *Proceedings of the VLDB Endowment*, 7(11):963–974, 2014.

[122] Xelera Technologies. Xelera Secra, 2021. `https://xelera.io/assets/productbrief_xelera-secra.pdf`.

[123] Xilinx. PG134: AXI HWICAP v3.0, 2016. `https://www.xilinx.com/support/documentation/ip_documentation/axi_hwicap/v3_0/pg134-axi-hwicap.pdf`.

[124] Xilinx. UG574: UltraScale Architecture Configurable Logic Block, 2017. `https://www.xilinx.com/support/documentation/user_guides/ug574-ultrascale-clb.pdf`.

[125] Xilinx. UG1085: Zynq UltraScale+ DeviceTechnical Reference Manual, 2020. `https://www.xilinx.com/support/documentation/user_guides/ug1085-zynq-ultrascale-trm.pdf`.

[126] Xilinx. UG573: UltraScale Architecture Memory Resources, 2020. `https://www.xilinx.com/support/documentation/user_guides/ug573-ultrascale-memory-resources.pdf`.

[127] Xilinx. UG579: UltraScale Architecture DSP Slice, 2020. https://www.xilinx.com/support/documentation/user_guides/ug579-ultrascale-dsp.pdf.

[128] Xilinx. UG909: Dynamic Function eXchange, 2020. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug909-vivado-partial-reconfiguration.pdf.

[129] Xilinx. AM007: Versal ACAP Memory Architecture, 2021. https://www.xilinx.com/support/documentation/architecture-manuals/am007-versal-memory.pdf.

[130] Xilinx. DS950: Versal Architecture and Product Data Sheet Overview, 2021. https://www.xilinx.com/support/documentation/data_sheets/ds950-versal-overview.pdf.

[131] Xilinx. DS963: Alveo U280 Data Center Accelerator Data Sheet, 2021. https://www.xilinx.com/content/dam/xilinx/support/documentation/data_sheets/ds963-u280.pdf.

[132] Xilinx. GQE Kernel Acceleration Demo, 2021. https://xilinx.github.io/Vitis_Libraries/database/2019.2/gqe_guide/kernel_demo.html.

[133] Xilinx. Vitis Database Library, 2021. https://www.xilinx.com/products/design-tools/vitis/vitis-libraries/vitis-database.html.

[134] Jingren Zhou and Kenneth A Ross. Implementing Database Operations Using SIMD Instructions. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 145–156, 2002.

[135] Daniel Ziener, Florian Bauer, Andreas Becher, Christopher Dennl, Klaus Meyer-Wegener, Ute Schürfeld, Jürgen Teich, Jörg-Stephan Vogt, and Helmut Weber. FPGA-Based Dynamically Reconfigurable SQL Query Processing. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 9(4):1–24, 2016.

Appendix A

Xilinx Zynq UltraScale+: Quantitative Memory Analysis

A.1 Hardware Setup

We have implemented and placed RTL modules in the Programmable Logic (PL). These modules independently use the read and write ports of the 4 High-Performance AXI ports. Our modules can simulate the placement of up to eight accelerators — four write memory intensive (e.g., mandelbrot frame rendering) and four read memory intensive (e.g., a MapReduce accelerator) or combinations of these. This as well fits our problem of stream processing, as it simulates data-bound problems. However, we do not constrain the ratios of read-to-write operations to 1:1 due to the wide variety in data requirements of stream processing accelerators (see Section 3.2.1). Figure A.1 depicts the connection of these modules with the PS. Each test module is fully programmable from the host CPU, allowing us to change its state at runtime for adjusting the number of operations, memory address spaces and burst lengths. To ensure there are no conflicts between accelerators, we reserve a unique address space in the DDR memory for each accelerator. All modules are instrumented with an integrated Performance Monitoring Unit (PMU), which allows us to obtain *real-time information* about read/write operations executed, active runtime, as well as average and maximum read latencies. The PMU registers are memory-mapped and are controlled using the AXI Slave as shown in Figure A.1.

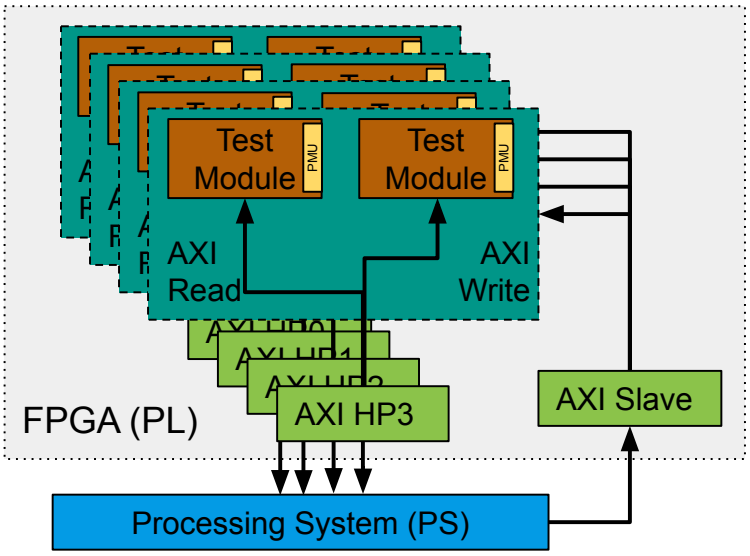

Figure A.1: Experiment setup for memory transaction handling on Xilinx Zynq UltraScale+ [63].

A.2 Experimental Setup

The base experimental setup consists of all AXI combinations configured to measure read-only, write-only, and read-write performance for each setup. The test modules are directly connected to the PS HP AXI ports without any additional interconnect as an intermediary. Further, these modules are controlled from the CPU in bare-metal mode to minimize memory overhead and interference caused by running applications and an operating system. Moreover, our tests are long and free-running, i.e. each configuration runs for 5 seconds before being stopped to capture the results. The long period of execution includes many DDR refresh cycles, which tests for worst-case scenario latencies. Every configuration is tested 10 times to minimize and quantify errors.

The system runs with a global clock of either 100MHz or 300MHz in different test scenarios. The read-only and write-only tests are executed for both ZCU102 and Ultra96 at 300MHz to achieve maximum available single-AXI performances, while our base read-write tests run at 300MHz and 100MHz on the ZCU102 and Ultra96 respectively symbolising their most common use cases as datacentre FPGA and personal/IoT FPGA respectively.

The ARM CPU cache line size is 64 Bytes in Zynq Ultrascale+ devices. Given that the DDR memory controller is also supplied by ARM, we would anticipate that

it is largely optimised to operate on memory accesses of this burst size. Moreover, cache lines are always aligned in memory, which should also be considered when using the DDR controller. In our experiments, small bursts (up to 512 Bytes) are multiple of 16 Bytes and large burst sizes are always the multiple of the ARM cache line size to ensure optimal operation of the DDR controller. Additionally, all of our accesses are memory aligned to the burst size for the test (i.e. 16 Byte bursts will have data aligned to 16-Byte boundaries). Data alignment and positioning is a common technique that can overcome DDR controller alignment issues as well as optimal cache line utilization from software [30]. The maximum achievable AXI burst size for a 128-bit AXI configuration is 4 KiB, thus we evaluate AXI performance with configurations of up to 4 KiB. Note that the DDR memory controller is capable of executing memory requests out of order and has QoS modules that can define the order of requests to the memory. In situations where the controller cannot select more optimal execution ordering, read requests are prioritised over write requests [125].

We use a DDR controller configuration to map the DDR address space in Row-Bank-Column fashion for our base case, which is the default and widely used configuration in computing systems, as it results in bank interleaving when accessing large sequential arrays of data. Additionally, we set the System Memory Management Unit (SMMU) into by-pass mode for the accelerators to minimize the throughput overhead caused by the Translation Buffer Units (TBU).

With these base settings, we exhaustively test every AXI and frequency combination for multiple different burst sizes to characterise memory performance under various scenarios. Individual changes are made in the base setup to evaluate the impact of access patterns, multiplexing in PL and Quality of Service (QoS) in isolation (i.e. keeping all other parameters constant).

A.3 Memory Subsystem Evaluation

We have evaluated eight primary areas: 1) peak performance, 2) performance of AXI ports, 3) transaction frequency, 4) access patterns, 5) memory organisation, 6) multiplexing overhead in PL, 7) quality of service and 8) performance distribution impact [63]. Based on our experiments we found that throughput scales linearly with AXI width size and, hence, to quantify the maximum throughput achievable all the experiments from here on use 128-bit AXI connections.

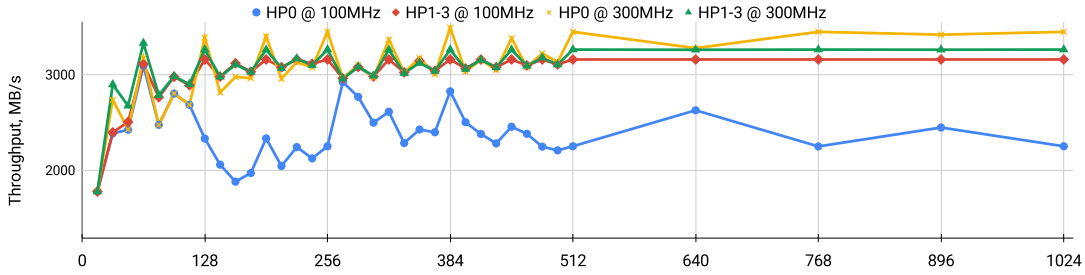

Figure A.2: Available duplex throughput with respect to burst size in Bytes of single AXI configuration in Ultra96 running at either 100MHz and 300MHz.

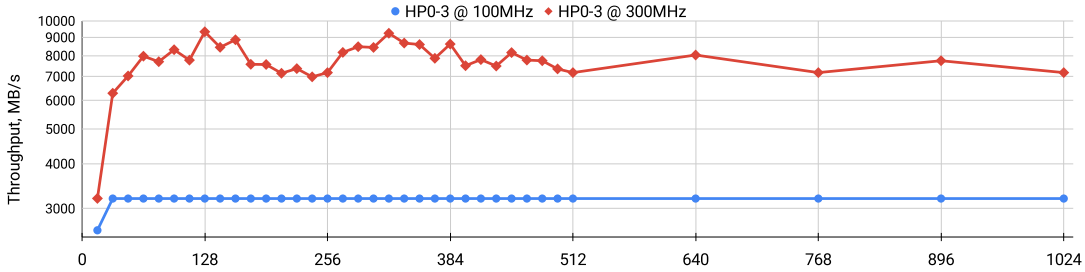

Figure A.3: Available duplex throughput with respect to burst size in Bytes of single AXI configuration in ZCU102 running at either 100MHz and 300MHz.

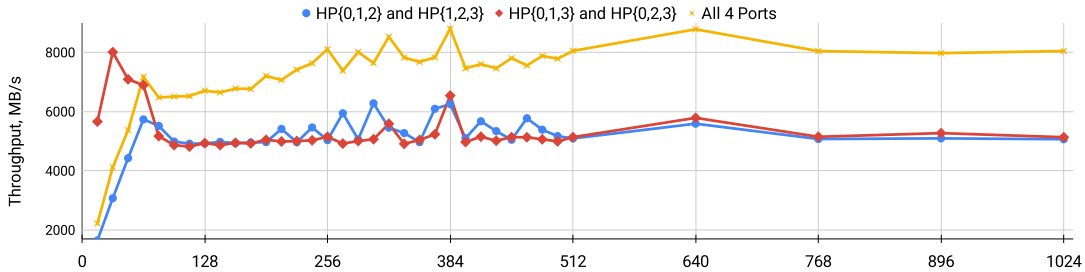

Figure A.4: Available duplex throughput with respect to burst size in Bytes for multiple AXI users in ZCU102 running at 100MHz.

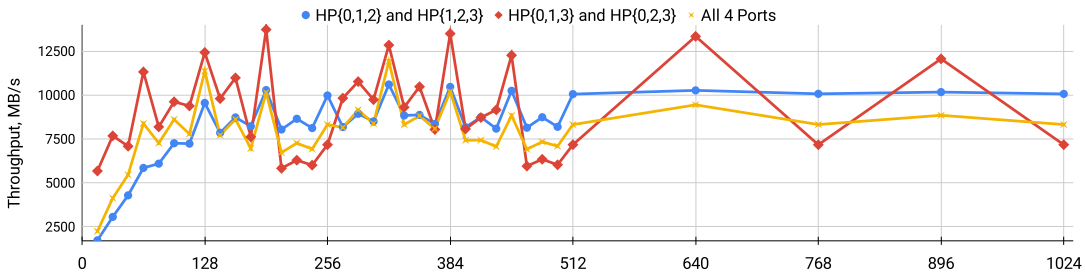

Figure A.5: Available duplex throughput with respect to burst size in Bytes for multiple AXI users in ZCU102 running at 300MHz.

Peak Performance

To minimize switching between R/W operations, we evaluate the peak memory performance provided by the boards using read-only and write-only test scenarios. Since the DDR theoretical performance for ZCU102 is larger than the available throughput of a single AXI port, we also include the test case using multiple AXI configuration that achieves the largest throughput (using HP0, HP1 and HP3 simultaneously). We observed that:

- The write-only throughput is larger than the read-only throughput with on average 11-13% higher write speed.
- At a burst size of 128 Bytes, the throughput reaches near maximum for all configurations.
- Burst sizes using a multiple of 64 Bytes yield local peak performances except for the burst sizes which are also multiple of 256 Bytes as they show decreased throughput for various AXI ZCU102 read-only configurations.

The Ultra96 configuration uses a single AXI that issues sequential bursts on either the read or write port. This avoids request multiplexing or changing between read and write mode in the DDR controller, thus achieving a high maximum throughput of 92.5% of the theoretical DDR memory peak on the Ultra96. In contrast, the ZCU102 needs to utilise multiple AXI ports to achieve the highest throughput, which leads to some multiplexing in the DDR controller between the requests and does not scale linearly with the number of ports. The peak performance in ZCU102 was found at 128 Byte bursts, which is 75% of the theoretical peak for the DDR memory.

Note, contrary to the common assumption, *using all AXI ports does not lead to the highest throughput* in either of the boards as HP1 and HP2 are multiplexed in the PS (see Figure 2.2).

A.3.1 Performance of AXI Ports

We execute read/write base tests utilising all AXI combinations and burst sizes, to observe their respective behaviours. The tests are executed at 100MHz and 300MHz respectively for both Ultra96 and ZCU102.

Standalone

When a single AXI port is accessing memory, both of its read and write ports typically utilise different memory regions, which can essentially lead to row pollution of each other. We find that all AXIs in ZCU102 behave similarly, while HP0 in Ultra96 behaves differently than HP1-3 (see Figure A.2).

We can rationalise the behaviour discrepancy between the ports on Ultra96 by the fact that HP0 shares its DDR connection with the DisplayPort that uses DDR memory for frame buffering. However, the same behaviour discrepancy is not observed for ZCU102 which objects our explanation. Upon further examination, differences between the Quality of Service (QoS) configurations were found. However, this is highly unexpected since according to technical documentation [125], the QoS module of HP0 does not influence the behaviour of the memory accesses of the DisplayPort. Overall, we can conclude that *1) same type of AXI ports may not necessarily show same performance behaviour and 2) after a point, increase in burst size may backfire depending on port's QoS settings*.

The average read latency in all single AXI configurations on both boards was found between 250 and 10,000 clock cycles and scales linearly with respect to burst size.

Combinations

When multiple AXIs perform memory requests on the same memory region, significant row pollution might occur. The trends on Ultra96 are similar for all configurations with at most about 20% difference between best and worst-performing configurations. We found that the best performing configuration is the one including HP1-3. This configuration also has balanced read/write distribution, but due to the multiplexing of HP1 and HP2 in PS, HP3 receives 50% of the available throughput, leaving HP1 and HP2 with only a 25% share for each. In all configurations with HP0 enabled, HP0 obtains a significantly larger portion of the available throughput and read requests deliver at least 20% more throughput than write requests. The ZCU102 in contrast always has a balanced distribution between different AXIs (except HP1 and HP2, which share an AXI port to DDR). Configurations HP0,1,3 and HP0,2,3 show oscillating throughput behaviour with respect to the burst size. Whereas all two-AXI configurations except HP1-2 achieve a more stable throughput trend, peaking at 128-Byte bursts to 11,600 MB/s and 320-Byte bursts to 12,640 MB/s. Notably, *the more AXIs in a configuration,*

the higher the imbalance between read and write throughput, with 4 AXI combinations and burst sizes of more than 128 Bytes, the write throughput reaches only 1% of the total.

On the Ultra96 board, the average read latency of HP0 in all AXI combinations is the same as standalone HP0. In configurations including HP0, all other AXI ports experience an average latency increase of up to an order of magnitude higher than their standalone cases. This is because the read port of HP0 is configured to a higher priority by default, which pollutes the read (and write) requests from all other AXI ports. On the ZCU102, the average read latency in multiple AXI configurations increases to about 2× for 3-AXI and 4-AXI configurations. In both devices, HP1 and HP2 face up to a 2× latency increase over other AXI ports if they are simultaneously used in a configuration.

We have not observed any large periods of AXI read unresponsiveness, which might be caused by long DDR refresh procedures. The DDR controller manages to hide the refresh cycles as (other than HP0, which is polluting the other AXIs in Ultra96) we observe maximal AXI read inactivity of about 200-300 cycles in our tests.

A.3.2 Frequency

As both boards feature the same ARM SoC, the theoretical aggregated throughput between PS and PL is 12.8 GB/s (4 AXI ports at 2x1.6 GB/s each) when running the PL at 100MHz. When we test the ZCU102 at 100MHz, we observe a peak performance of 8,800 MB/s when operating all AXI ports at bursts of 384 Bytes. This is only 14% lower than the same configuration running at 300MHz but is 36% lower than the HP0,1,3 and HP0,2,3 configurations running at 300MHz. Additionally, the read requests take all the available AXI read ports throughput and the mentioned configuration achieves 6.4 GB/s read throughput but only 2.4 GB/s write throughput. Since all 4 AXI read ports achieve their theoretical maximum of 1.6 GB/s, this reveals that the HP1-HP2 multiplexing in PS is happening after clock domain crossing at a higher PS frequency.

Whereas, running Ultra96 at 300MHz results in a major increase of throughput in configurations including the use of HP0 except for burst lengths of 64B and 128B. The distribution between throughput to the different AXIs is much more spread, as HP0 reaches its peak performance of ≈3GB/s, while other AXIs in the configuration achieve peaks of up to 200-600 MB/s only.

Overall, the higher frequency can translate to higher memory throughput depending

on the AXI combinations and DDR memory chips available on the board. However, *even a 300MHz PL clock speed in the best case only gains 55.6% over the best case of 100MHz PL configuration* in our experiments.

A.3.3 Access Pattern: Sequential vs Random

We also run the experiments using random access patterns to identify and measure the loss of performance caused by the rapidly changing memory sections in a multi-tenant environment. Note, here the access pattern implies address differences between separate burst requests and that the memory accesses inside a particular burst request are always sequential. For Ultra96 we observe a decreased throughput of up to 70% for burst sizes of only 16 Bytes and up to 2-10% decrease for large burst sizes compared to the best performing configurations from the base case. This is expected, since large burst sizes compensate the associated overhead of switching rows in the DDR memory, while for small bursts, the overhead is large relative to the work performed.

For the same experiment performed on the ZCU102, we observe less predictable behaviour than Ultra96. All configurations in our random test provide at least 6% less throughput than the peak configuration from the base case (see Figure A.9).

Overall, *for multi-tenant systems, burst sizes of $\geq$ 512 Bytes minimize the overhead of changing access pattern in the DDR controller*. When using small burst sizes it is recommended to maintain sequential access patterns as much as possible for the highest memory throughput. This also indicates that the DDR controller attempts to perform bulk operations if possible.

A.3.4 Memory Organisation: Row-Bank vs Bank-Row

Since FPGAs provide a high degree of flexibility in how we organise and use the hardware, customising the memory architecture along with hardware needs is an important optimization avenue. Hence, we also ran experiments on Bank-Row-Column address space organisation along with Row-Bank-Column in our DDR memory.[1] To try and utilise this to our advantage, we place the work memory regions of our test accelerators into different banks, such that each of our accelerators takes two reserved banks - one for read accesses and one for write accesses.

Running the experiment on ZCU102 resulted in a very minor difference in throughput pattern compared to the Row-Bank-Column. Most points in our results are within

[1] By configuring the address map inside DDR settings for Zynq IP core.

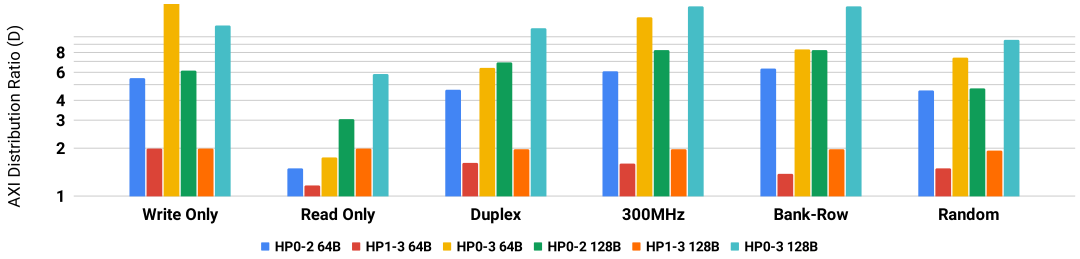

Figure A.6: Throughput distribution between AXI configurations on Ultra96. The fairest distribution is when $D = 1$. Note, **Write Only HP0-3 64B** has a ratio of 5165 and is not captured entirely in the graph.

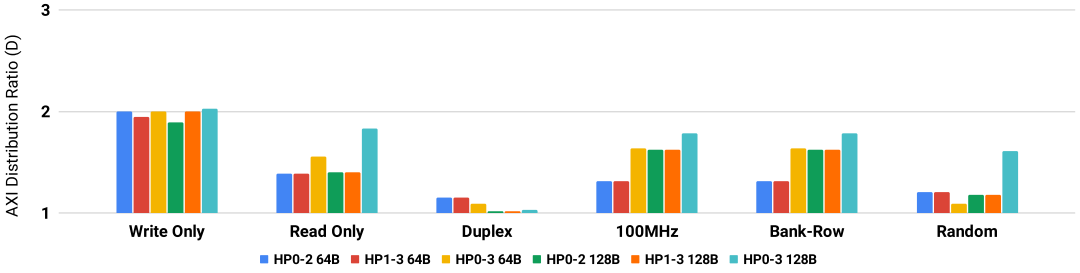

Figure A.7: Throughput distribution between AXI configurations on ZCU102.

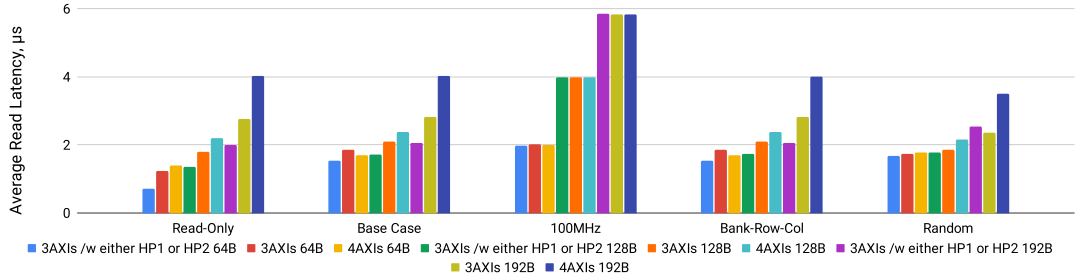

Figure A.8: Read latencies measured on ZCU102.

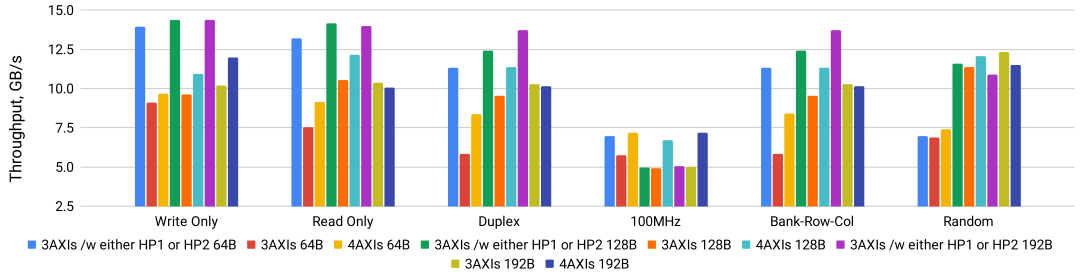

Figure A.9: Throughput measurements on ZCU102 for various setup parameters.

statistical error of around 0.5% compared to our base test results. The only notable difference in throughput is when using both HP1 and HP2, which yields about a 7% decrease in throughput for some burst lengths. These unexpected results on the ZCU102 might be explained by the DDR memory used, which is organised into bank groups in which banks share pre-charge components and our experiment setup used 8 banks that are organised into 2 bank groups symbolizing half of the DDR memory resources available. Read latency does not change with respect to the base case (see Figure A.8).

Overall, *changing the address space organisation mostly affects performance at small burst sizes for selected ports (up to 31% improvement for Ultra96) and depends on whether the memory is organised in bank groups or not*. For large burst sizes, there is no significant change in throughput behaviour. What is more, it can backfire if the number of accelerators is higher than the banks available.

A.3.5 Quality of Service

Upon further examination of the AXI interconnect and DDR subsystem to identify the cause of the difference in behaviour across different boards, we found that there are differences in QoS buffer modes and R/W priorities. The mode for QoS can be 1) High Priority (low latency), 2) Best Effort (bulk transfers) and 3) Isochronous (regular, time-sensitive, e.g., audio and video traffic) [125]. On ZCU102, all HP AXI ports share the same Isochronous mode and memory access priorities on both read and write ports, whereas on Ultra96 AXI HP0 is in Isochronous mode while the other AXI ports (HP1-3) are Best Effort and the write ports are assigned a higher priority than read ports. Changing the mode and priorities on Ultra96's AXI HP0 port to the same configuration as the other Ultra96 HP ports, produced results where all AXI ports behave similarly with slightly higher throughput, lowered average read latency, and more fair AXI and R/W distributions (see Figure A.6). Note, despite having the same priorities for read and write, in practice, we observed that ZCU102 AXIs and Ultra96's HP0 prioritises read operations, while Ultra96's HP1-3 have balanced R/W ratios. This is due to the general DDR memory controller prioritisation of read over write operations. This default prioritisation in the controller can be explained by the nature of most CPU applications, where applications frequently stall for read operations before the actual compute. However, one must note that different FPGA applications have different access patterns and read-to-write ratios and they can utilize Block RAMs as internal buffers and operate in a highly parallel or streaming manner.

A.3.6 Performance Distribution

To understand the performance distribution in multi-tenant environments, we measure the AXI distribution ratio D as A_{max}/A_{min}, where A_{max} is the throughput of the highest performing AXI port and A_{min} is the throughput of lowest-performing AXI in a configuration. The results of the distribution are shown in Figure A.6 and Figure A.7. *Most configurations have an uneven distribution which implies that it is very easy to have one accelerator steal all the bandwidth in a multi-tenant environment.* Two configurations stand out as possible options for multi-tenancy on Ultra96. 1) PL multiplexing which achieves ideal distribution ($D = 1$) but is subjected to change if the accelerators use different burst lengths. 2) Multiplexing in PS with the same QoS mode for each AXI port. This ensures that the default high priority ports do not steal performance. Note that for configurations involving both HP1 and HP2, the ideal distribution is $D = 2$ as they are multiplexed in the PS and that activating DisplayPort or FPD DMA components may affect the performance correspondingly of HP0 and HP3.

Conclusion

The conclusion of our memory analysis can be found in Section 3.1.2.