RUNTIME MANAGEMENT OF DYNAMIC DATAFLOWS WITH PARTIALLY RECONFIGURABLE PIPELINES ON FPGAS

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

2023

Kaspar Matas

School of Engineering
Department of Computer Science

Contents

Word Count: 41444

List of Tables

List of Figures

Glossary

blocked operation An elastic resource operation which the scheduler cannot yet schedule as previous data-dependent dependencies have not been resolved yet. 106

consumer In dataflow systems, consumer nodes are nodes that are waiting for input data to be ready before starting execution. 34, 37

data packet A base data element that a module targets. A stream consists of multiple data packets, which can consist of multiple words. 34, 53, 59–61, 66, 72

execution plan The scheduler maps different operations to chosen modules in time and space. Then with different runs, this execution plan is executed. 35, 103, 110

external fragmentation Unused resources within a partially reconfigurable region outside of module bounding boxes. 44, 67, 74

functional capacity Module parameter denoting how much processing the module can provide in an abstract way. 46, 47, 53, 66, 72, 76, 83

hierarchical reconfiguration Swapping a part of a configuration inside another partial module.. 44

initialisation Runtime system programming the modules by writing operation parameters to memory-mapped registers. 64, 65, 69, 70

internal fragmentation Unused resources within a partially reconfigurable module. 74

module alternative A module which is processing the same operation as the original module but with a different functional capacity. 47, 64, 66, 68, 151

module bounding box A module design constraint that sets the boundaries outside of which a module can use not a single resource. 43, 46, 70, 74

module composing Building larger virtual modules with multiple physical modules. 47, 67, 68, 74

module variant A module which is processing the same operation as the original module but with a different resource footprint. 47, 55, 66–68, 151

multiplexing Sharing data between multiple processing elements. With time-multiplexing, this is done over time rather than with splitting routing paths. 49, 50, 101

partially reconfigurable region The region where partially reconfigurable modules can be placed. Consisting of numerous slots in resource-elastic systems. 23–26, 44, 46, 49, 50, 70–72, 75–78, 80, 81, 83–85, 87, 88

producer In dataflow systems, producer nodes are nodes that are producing data as a result of their execution for next nodes to consume. 34, 37

projection A common data analytics operation that filters out unnecessary columns of data. 131

resource elasticity The ability to optimise performance by using a variable amount of resources during runtime.. 19, 28, 43, 44, 46, 56, 59, 66, 69, 74, 83

resource string After defining each slot within a reconfigurable region with a character based on the resources it provides, the combination of multiple slots correlating to the requirements of a module is characterised with a resource string - a combination of different letters. 23, 72, 81, 93, 94

scheduling spin A single execution of the scheduler in the scheduling state to schedule as much as possible. 83, 105, 106, 112, 113

service request A front-end client can request various operations to be accelerated, given that the library supports them. 26, 68, 72, 105

static system FPGA design elements that are compiled and placed offline and stay unchanged during runtime. 25, 42, 43, 65

utility The average amount of practical computation performed on data per pass through the accelerator. 69, 70, 74

Abstract

In order to overcome the famous von Neumann bottleneck, FPGAs employ a dataflow model that processes data through a pipeline of operator modules, akin to an assembly line for computation. This approach, which resembles how an assembly line streamlines logistics, is highly effective in utilising I/O resources. However, unbalanced producer-consumer rates in these pipelines cause underperformance by idling parts waiting for data. Furthermore, even if a pipeline is optimal for one target workflow, it will still be unoptimised or, even worse - unable to execute another potential target workflow. Creating a universally optimal pipeline ahead of time can become unattainable with data-dependent behaviour where the constraints and objectives change during runtime.

FPGAs fit perfectly the dataflow model with their reconfigurable grid of resources. This thesis proposes a middleware that utilises partial reconfiguration to enable dynamic adaptation of these resources at runtime in response to changing requirements. On the one hand, with smart enough scheduling, at the very least, the system automatically compiles static pipelines as domain-specific accelerators while avoiding reconfiguration costs. On the other hand, when the advantages of dynamically created pipelines overcome the associated overhead costs, the system can improve performance.

This thesis examines how to operate and integrate a general-purpose system for dataflow processing consisting of resource elastic modules, schedulers, input parsers, and data and memory management. We implement and evaluate the system in data analytics and image processing workloads. The results show that with large enough datasets, the pipeline scheduling overhead and fragmentation costs are neglectable and that reconfiguration can consistently outperform comparable static systems that do not employ partial reconfiguration.

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

First of all, I would like to say “Cheers” to my supervisor, Prof. Dirk Koch, from the bottom of my heart. I deeply appreciate your tireless support, which motivated me to challenge myself and my ideas more and more.

Then I would like to thank everyone in our Advanced Processor Technologies research group and in the department of computer science for creating a cooperative and productive environment. I would particularly like to express my gratitude to Nikola Grunchevski, Joseph Powell, Tuan Minh La, and Khoa Dang Pham for our fruitful collaborations. My special thanks go to Kristiyan Manev for helping me stay focused on the essential details of my research.

Then for last but not least, I am beyond grateful to my family and friends, both close and far, to keep supporting me through all of the good and challenging times unconditionally.

Chapter 1

Introduction

Thesis statement: Reconfigurable hardware (HW) conducted by a software (SW) orchestration layer is the most appropriate to accelerate dynamic problems given an agile utilisation of resources despite the seemingly large overheads of reconfiguration.

1.1 Motivation

Initial Turing-Complete machines were incredibly slow for modern general-purpose requirements (e.g., the ENIAC from 1945 was clocked at 0.005MHz [91] while Windows 11, released in 2021, requires at least a 1GHz processor/CPU [206]). In general, we want as much compute work done as possible at any moment and start prioritising **throughput** given acceptable **latency** for increased **performance**. Therefore, the technology industry has thrived on selling products (various CPU types or specialised devices, from GPUs to ASICs) with continuously improving raw computational capabilities, often measured in floating point operations per second (FLOPS).

Nowadays, the single-core CPU performance increase trends due to shrinking transistor sizes and increasing clock rates are mostly stagnating [297]. Meanwhile, multicore CPUs, simultaneous multithreading, and the "single instruction and multiple data" (SIMD) instructions have driven the advertised FLOPS. Nevertheless, temperature and power constraints and the communication requirements between the cores and the system memory are becoming more complex with growing core counts and are hitting performance. Consequently, with a steadily increasing demand for advanced data analytics [286], there is a need for heterogeneous acceleration [303][1].

[1] Heterogeneous acceleration refers to the use of multiple types of computing HW, including FPGAs this thesis focuses on, to leverage the strengths of each type of HW.

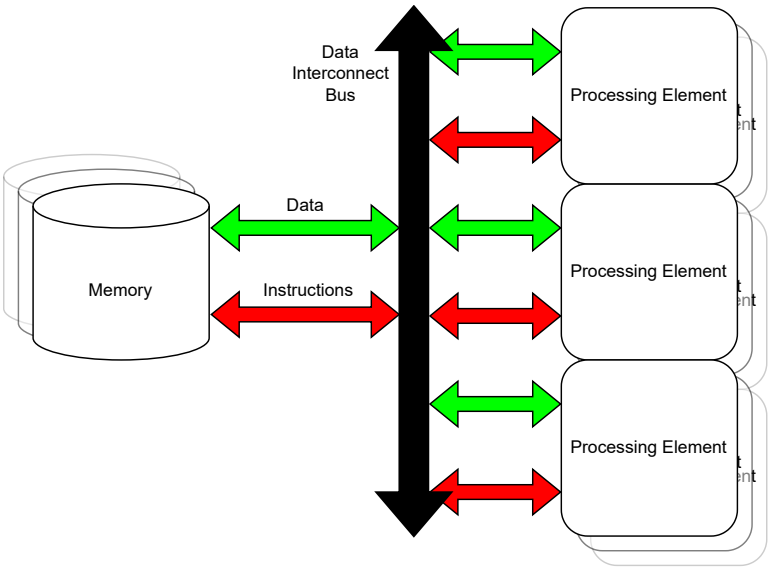

Figure 1.1: With the Von Neumann architecture (or with the Harvard architecture), processing elements get both data and instructions from memory which causes contention over shared datapaths when trying to scale systems with more memory and processing elements (optimised to an extent with caching).

The need for accelerators stems from the von Neumann bottleneck apparent in modern CPUs due to the computational power of the processing elements (PEs) and memory density advancing faster than the data transfer speeds [185]. This problem is exacerbated when multiple processing elements share the same communication infrastructure and require instruction data, prevalent in multi-core CPUs, as shown in Figure 1.1. We need to balance the resources (possible with customised HW) to avoid being **compute-bound** or **memory-bound** by finding the most optimal resource allocation using the roofline model in Figure 1.2. The following architectural design principles in accelerators stem from the fact that *data movement is expensive*:

- Move compute closer to memory like with in-memory or near-memory processing to improve memory bandwidth [349, 240].
- Pipeline processing elements to form a *dataflow* system to reuse data as much as possible and improve the compute density of the problem [218].

Heterogeneous reconfigurable HW can follow these principles and thus is already used to address the growing demand for faster data processing (due to the emergence of *big data*) in large-scale computing solutions (e.g., a famous example of using FPGAs is in the search engine Bing - Microsoft's Catapult [45]). The energy efficiency of FPGAs [53] and their reconfigurability enables additional architectural optimisations after

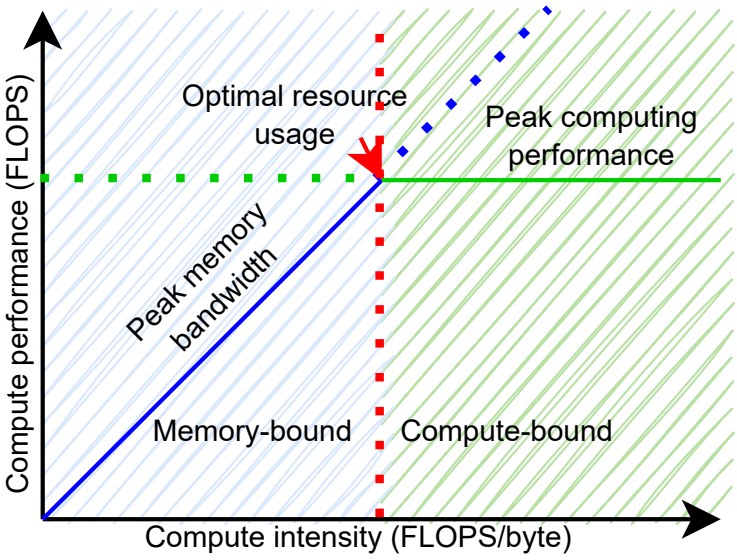

Figure 1.2: The roofline model divides performance by the time it spends on the following: 1) moving data around between memory and processing elements (PE), and 2) executing the required steps to finish the task with the given data.

the technological and algorithmic optimisations have been exhausted, with numerous ways to integrate FPGAs into data centres [278] (including standalone use [2]). As a result, CPU manufacturers Intel [121] and AMD [12] have acquired Altera and Xilinx for billions of USD, respectively [2]. While FPGA specialists expertly summarised the fine details of using FPGAs in data centres (Bobda et al., [31]), they noted that the missing piece for making FPGAs a commodity has been the scarcity of SW tools and the immaturity of the whole SW stack. Nonetheless, FPGAs excel at dataflow processing and *techniques from the dataflow processing paradigm that can allow scaling processing pipelines* - which further flames the need for productive SW tools to handle the increasing complexity and facilitate code reuse.

1.1.1 Static Dataflow

So far, the industry has been increasing the core count and maximising their use with SIMD instructions as much as possible (e.g., with large counts of cores on GPUs [180]) to alleviate the von Neumann bottleneck. However, in applications that utilise the dataflow processing model, such as machine learning (ML), the power consumption

[2] In this thesis former Altera's and Xilinx's products are referred to as Intel's and AMD's.

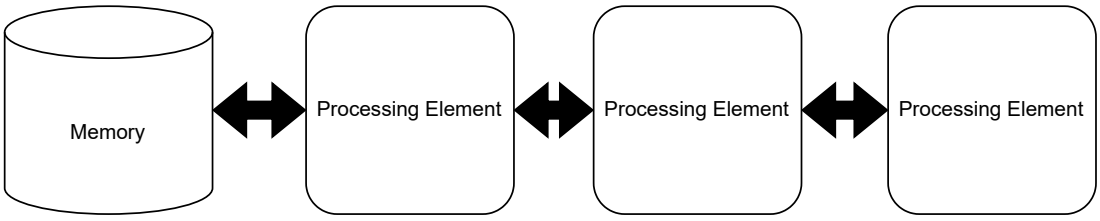

Figure 1.3: Dataflow architecutre allows sharing data inherently, which, in turn, circumvents the von Neumann bottlenck.

of GPUs has become excessive [290]. As a result, heterogeneous hardware is being increasingly used to take advantage of the benefits of dataflow [264]. Therefore, there has been a recent focus on exploring solutions based on ASIC (such as Google's TPU [130]) and automated design flows [221] that create embedded systems with the dataflow architecture depicted in Figure 1.3, especially for DSP applications [346]. The main benefit of this architecture is that it avoids sending expensive instruction data through contested paths which themselves become more expensive as they require a more significant number of multiplexers (HW switches for choosing data destination) when the number of directly connected processing units grows.

The pipelined execution (Figure 1.4) is one of the reasons why dataflow processing is compelling, as all of the **operator modules** (processing elements) can work in parallel with minimised memory overheads even if there are direct dependencies between the required operations, contrary to alternative parallel computing patterns [232]. The system of modules is most efficient when all pipeline stages take the same amount of time as all other stages to prevent faster modules from idling and waiting for new data to arrive. Designing these balanced systems is more straightforward to accelerate in applications with a stable and deterministic flow, like machine learning and video processing [208, 70]. However, application domains that require more universality to support a broader range of workloads with data-dependent conditions must use an excessive number of operators with a "one fits all" static configuration to cover all corner cases like, for instance, the relational database management system (RDBMS) AxleDB [269]. An alternative to this overprovisioning is to accelerate only a subset of possible operations (i.e., filtering and compression) and schedule the remaining work on a CPU [79, 342] - which misses exploiting the advantages of deep pipelining available with dataflow processing. Nevertheless, with data-dependent conditions, the following is certain: *for increasingly dynamic problems, static pipelines become inefficient or inoperable.*

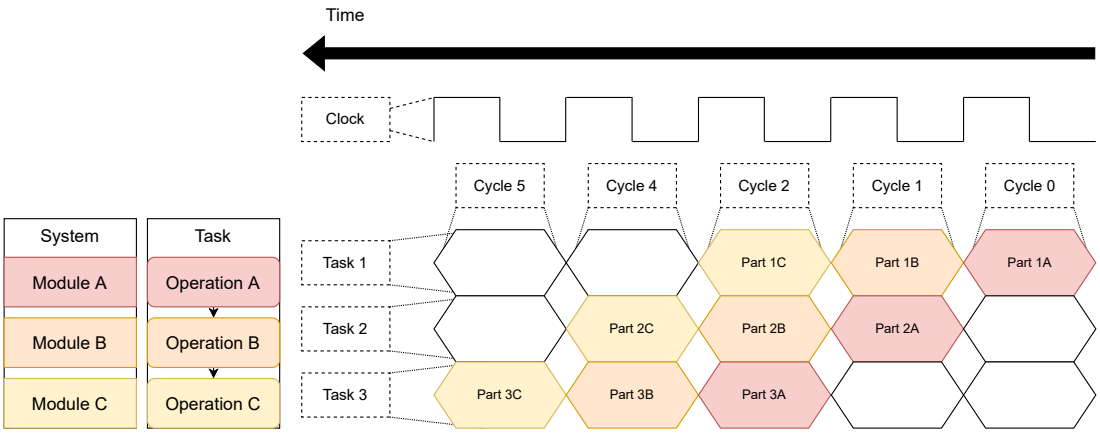

Figure 1.4: If all of the steps in the pipeline take the same amount of time with balanced producer-consumer rates, then data gets processed at a different module each cycle.

1.1.2 Dynamic Dataflow

We propose using reconfigurable HW to create pipelines **online** rather than offline, analogous to optimal dataflow systems that are dynamically created for distributed systems [112, 295, 272]. The dynamic approach allows adaptation to conditions and requirements that become apparent only during runtime. A runtime middleware can provide adaptive acceleration services with optimised resource allocation with a **modular approach** where the most optimal dataflow system is built on the fly out of the available "lego blocks" in its module library. The modules must provide enough flexibility in terms of resource, performance, and functionality tradeoffs for the middleware to find the most optimal balance for all variable runtime requirements (with the limited resources available on a single device). In order to balance all resource-performance-reconfiguration costs transparently for a general extendible solution, the implementation details have to be virtualised with a *resource-elastic* module library consisting of pre-synthesised accelerators[3]. The property *resource elasticity* allows changing the resource allocation of a task transparently to the user, which is commonly used in distributed stream processing [64] and more recently proposed by Vaishnav for virtualising FPGA processing [301]. In this thesis, we use a more fine-grained resource-elastic resource allocation required for dataflow systems while maintaining the additional benefits of abstraction, multi-tenancy, resource management, and data isolation, connected with FPGA virtualisation techniques [254].

[3]Synthesis is an HW design flow step analogous to SW compilation where the hardware description language (HDL) code gets transformed to an optimised gate-level netlist - circuitry.

1.2 Big Picture

Because of the lack of community-supported tools, reconfigurable HW is seldom used in the industry as compared to more mature devices such as CPUs and GPUs. Furthermore, often in the industry, reconfigurable HW resembles an "updatable ASIC" as the platform's dynamic partial reconfigurability (DPR) capability is ignored as part of the operation of a system. Therefore, most advancements improve static pipelines by making them more efficient for smaller problem subsets or generalising static pipelines. DPR allows changing parts of the logic of a circuit during runtime, but it has a high cost as the reconfiguration process takes a substantial amount of time. Due to the perceived high cost and minimal support from device vendors with tooling, there exists no mature ecosystem to build systems that can manage DPR with optimisations that can "hide" the costs.

There are two types of reconfigurable HW systems: 1) coarse-grain configurable systems like CGRAs that trade their flexibility for more performance and faster reconfiguration [182] and 2) fine-grain configurable systems like FPGAs that are flexible enough to emulate any ASIC designs given enough resources [163]. These types share the benefits of reconfigurable HW (energy efficiency and a high level of parallel computational power) and the common disadvantages (complex to develop due to lacking open-source community sharing knowledge and SW tooling). However, it can be challenging to scale CGRAs due to the lack of flexibility [246], and as FPGAs are widely used in various complex applications in the industry [266, 85], this work will focus on FPGAs for a more general purpose solution.

1.2.1 Productivity

Using reconfigurable HW (especially with DPR) is difficult as it requires experience to perform better than highly mature CPU or GPU alternatives with a fast enough time-to-market. The vendor tools are relatively slow (with long synthesis times compared to compiling SW), and open-source tools had not shown improvements (in terms of quality-of-results and CAD tool time) for large physical implementation problems [118].

However, vendor tools are good at producing "Hello World"[4] projects and using prepackaged IPs, while most other custom solutions outside the vendors' vision are more complicated (including DPR, where additional tools are required, like IMPRESS [339] and many others discussed in the next Chapter).

FPGA design productivity can be improved by reducing long CAD tool times (especially apparent for large designs filling data centre FPGAs) and facilitating design reuse in more ways. Recent HLS improvements have led to higher quality designs while increasing developers' productivity to address this problem [166]. Furthermore, more tools have become available with reusable IPs from FPGA vendors, like the HLS compilers in AMD-Xilinx's Vitis and Intel's Quartus. However, in this thesis, we propose to improve productivity by reusing physically implemented modules (either synthesised from low-level HDL code or high-level HLS) that provide the following benefits both during design time and runtime:

- Every aspect of the system can be developed in isolation (given a set of common interfaces and constraints).
- Fast integration of complex systems (orders of magnitudes faster than full monolithic synthesis [328]).
- Bugs are more reproducible and easier to spot (important for handcrafted optimisations and to reduce CAD tool noise disruption).
- Tools often perform better when synthesising smaller designs (e.g., some tools automatically partition designs [164]).
- Speed up design space exploration with module alternatives providing resource and performance trade-offs (critical for resource-elastic systems).
- Enabling partial reconfiguration (ideal for dynamic time-variant compute problems where the functionality may change at runtime or where resource requirements exceed the device capacity).
- Enabling schedulers to manage the integration of the modules (allowing the further abstraction of implementation details for design automation).

[4]The phrase "Hello world" is often used in programming introductions to print a line of text and is a valuable way to test that the system works. Printing a line of text is complicated on configurable HW due to the required low-level interaction with peripherals, and as such, "Hello world" projects here refer to potentially complex systems, albeit still routine projects.

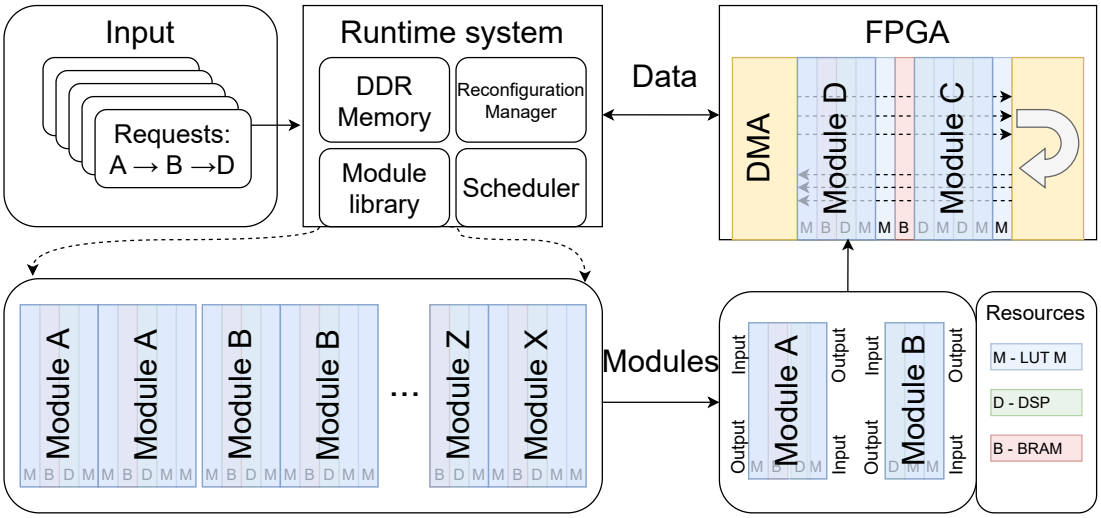

Figure 1.5: The different parts of an ecosystem reusing physically implemented and partially reconfigurable modules (in blue). The number of FPGAs, how the data is routed, and the size of the modules are all flexible.

Our approach brings FPGAs closer to modern SW applications that pay an overhead cost for an operating system to facilitate a shared ecosystem of reusable code where library functions are linked together for various requirements. Various FPGA design workflows could benefit from reusing physically implemented modules, including 1) rapid prototyping, 2) component-based design, and 3) partial reconfiguration.

There are open-source FPGA tools to build frameworks for both designing standalone or embedded custom FPGAs [181, 150, 177] and designing partially reconfigurable systems as surveyed by Vipin et al., [312] (e.g., [24, 28, 168]). For these frameworks to be practical there are also open-source tools for synthesising HLS (as surveyed in [54, 261]) and HDL (e.g., [277, 125]) designs while there are also tools for manipulating the resulting bitstreams that configure the FPGAs [238, 46, 192]. Given all of these available approaches and tools summarised in by Tessier et al., [294] we need to look at how to increase the productivity through an ecosystem that can maintain and share reusable code required for various parts of the process of running acceleration services on FPGAs.

1.2.2 Ecosystem

In our approach, we model the heterogeneous resources of an FPGA and determine what kind of ingredients are required to build up an ecosystem (Figure 1.5) consisting of 1) a runtime manager, 2) a static infrastructure managing the communication with

the manager and the operation of the reconfigurable modules, 3) a library of modules, and 4) a method to organise the acceleration tasks. These ingredients can be developed independently and with flexible composition capabilities, which fosters the creation of a community around a shared ecosystem of reusable building blocks

We propose such an ecosystem through the abstraction of resource allocation for which we define constraints for modules in PR regions (areas where multiple PR modules may be configured) while connected by **regular routing** (also placed with PR) constrained by the modules' interfaces. We will introduce constraints that allow simplifying the mapping problem to a one-dimensional placement problem (it can be extended to arbitrary 2D shapes and using multiple regions, but for the sake of brevity in explaining the concept, we model it as a 1D problem, as shown in Figure 1.5). We model the heterogeneous resource columns using symbols for any resource type available in an FPGA fabric (as the types of resources appear in a regular pattern dividing them into columns for the 1D model or a grid for the 2D model). The corresponding symbol strings form *resource strings*. The modules can be placed into a PR region when the resource string of that module matches any of the substrings in the PR region's resource string. With this, placing modules becomes a **string matching problem**. The HW library consists of multiple modules that compute an elementary operation or some usually small functions while requiring various resource columns. A module can have different implementations to allow trading performance or additional functionality for resources. Furthermore, modules may also compute partial results, which allows combining them with other modules. We developed a runtime system with a scheduler for automatically solving the dynamic placement problem. The runtime system also sends control signals to the FPGA to operate and integrate the reconfigurable modules.

To create such a general purpose (yet optimised) system capable of orchestrating dynamic dataflow pipelines, we need to address all following parts of the system in detail:

Operations: In order to accommodate a diverse set of conditions and constraints, the system needs to support a pool of operations. Therefore, a module library facilitates the creation of a list of matching operations that can be mapped by a higher-level client-side parser to execute the requested acceleration services. The module library also contains the metadata about the resource requirements and possible functionality/performance tradeoffs. For example, joining two data streams requires a join-like operation.

Dependencies: As stream processing dependencies and operation nodes can be modelled as a dataflow graph, we have to support all possible topologies. Therefore, any acceleration requests are mapped to an intermediate representation (IR) graph containing operation nodes with input and output dependencies. For example, a join operation requires multiple inputs while only having a single output.

Constraints: All HW modules can have unique constraints about the incoming data or their positioning on the device. These have to be handled transparently, for which we have a generalised system where module-specific drivers (SW to manage the HW) inform the system of any constraints. For example, to do a merge join, all of the inputs must be sorted beforehand.

Data management: As acceleration services may use operations on various forms of data, the input data has to be encoded in an acceptable format when placed on the datapath. Consequently, as part of the static infrastructure on the FPGA, a data mover has to be programmed to deliver data correctly on the wide data path (agreed upon time and location - using corresponding offsets) to the processing modules and retrieve the results. For example, for both table data containing columns with different data types or pixel data from image files, an abstraction layer is required to represent the supported encodings to the data mover and the modules.

Scheduling: All the constraints, data dependencies, and limited resources require solving a complex module scheduling problem. More complex streaming jobs may require multiple *runs*[5] with batch processing. Whether these batches are substantial or fast micro-batches depends on the capabilities of the available HW modules and the latency requirements of the accelerated application. For example, complex data analytics acceleration requests containing multiple filtering, joining, and sorting steps require *dynamic scheduling in time and space*.

Memory: In order to handle complicated topologies that span over multiple time-multiplexed runs, we need a memory management subsystem that handles storing the inputs and outputs of different modules. Any batch of data stored in RAM must be directly accessible by the data mover, and after the data has been

[5]With a *run*, we refer to a pass of data through the PR region, which may or may not have different pipelines of modules configured. Occasionally, a large dataset must be partitioned for streaming through the same pipeline with multiple passes (e.g., when sorting large problems)

processed, that memory address has to get freed. For example, data writing and reading require managing dirty memory regions, which can be done with a filesystem-like approach.

High-level integration: Lastly, to make integration with larger-scale applications seamless, we must explore how to integrate the FPGA systems. This integration requires application programming interfaces (APIs) which in streaming applications is often done with high-level queries in structured query language (SQL). SQL queries in stream processing allow for real-time, continuous data analysis (views of data snapshots) by providing the ability to perform complex operations on bounded or unbounded data streams. Moreover, while enhancing the ease of integration with other systems, the security of expanding the set of available operators with third-party developers needs to be carefully considered.

1.2.3 Integration

The proposed approach can be applied in various scenarios, from static offline integration on a single FPGA to complex runtime systems managing FPGA networks with multiple PR regions and concurrent tasks, as shown in Figure 1.6. The proposed system can find the optimal resource allocation on large devices even without using DPR when targeting a limited set of applications, provided that there are sufficient resources available. The module library contains highly optimised HW designs with a set of drivers that can move complex data types through a wide datapath with various control signals, enabling our approach to alternatively provide a domain-specific compiler.

On the other hand, in more resource-constrained dynamic scenarios, our approach can be used as capable middleware managing multiple simultaneous queries using DPR to allocate more resources to performance-hungry operations (similar to Just-in-Time[6] compilers) Therefore, given that the implementation of our approach is optimised enough, this novel tool can be used extensively as, by default, the resulting system will be as good as a comparable static system. If the performance benefits

[6]Just-in-Time (JIT) compilation is a process where source code is compiled into machine code during runtime rather than offline beforehand. This compilation approach gives the added benefits of portability and the option for additional optimisations and error checking, possible with information only attainable during runtime. One of the most famous examples of this is Java, where for performance reasons, the code is initially compiled to bytecode which then is compiled further to machine code during execution and the possible optimisations to overcome the added overheads have been covered in literature, for example by Ishizaki et al., [122]. Jain et al.,[124] also proposed JIT compilation on FPGAs with overlays while compiling OpenCL.

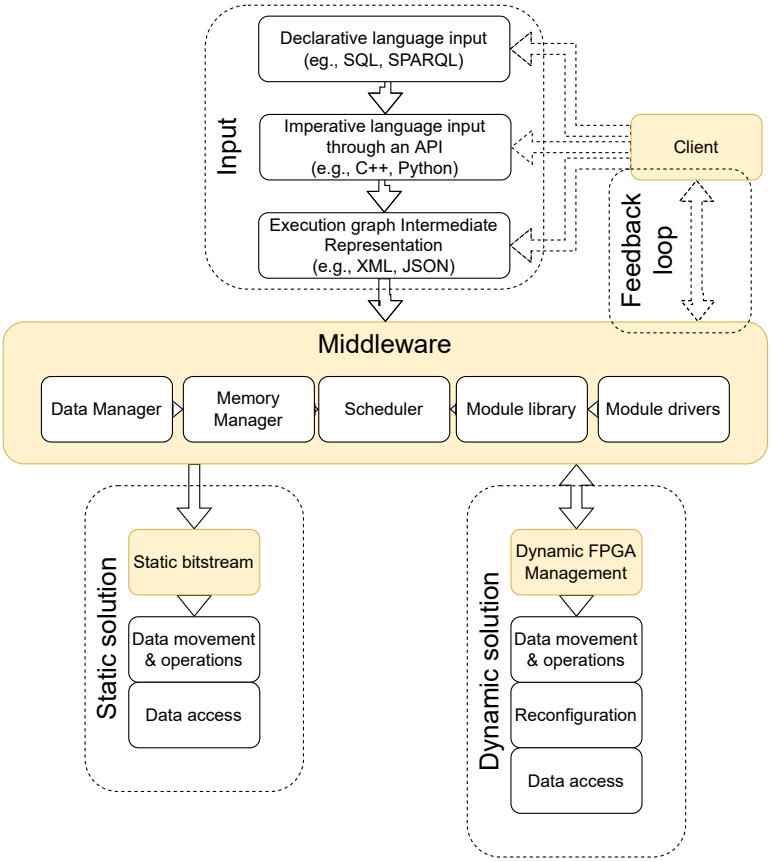

Figure 1.6: Our proposed methodology is centred around a middleware that parses and schedules input acceleration service requests (any level of abstraction) from multiple clients using substitutable system parts to provide either static bitstreams or dynamically managed FPGAs. The platform can be extended to manage multiple FPGAs with multiple PR regions and various ways to access data or configure the HW.

overshadow the costs of the dynamic approach, then the system can improve performance transparently to the end-user (seemingly for free).

1.3 Scope and Contribution

The scope of this work can be broadly categorised into two main areas: 1) The introduction of a design framework for creating middleware that orchestrates dynamic dataflow acceleration, establishing the theoretical foundation, and 2) a practical implementation example to evaluate this overarching approach.

For the theoretical framework to be applicable, we make the following assumptions about the system under design:

- The system can access a pre-synthesised module library. This library can be placed arbitrarily in the PR region, utilising pre-existing regular routing based on the resource pattern matching detailed in Section 1.2.2.

- The system can access a static FPGA partition that contains partially reconfigurable regions, allowing for the loading of appropriate module bitstreams. Subsequently, these regions and modules can be fed with data payloads (comprising both operational and instruction data) and also return results.

Given these prerequisites, we can design a middleware that accepts a dataflow problem described by a graph as input. The middleware then processes the graph to produce a sequence of dataflow pipeline configurations. These configurations detail the list of modules to be loaded, their respective locations, and the instructions required for their correct initialisation. Furthermore, this middleware is tasked with executing these pipelines to produce the final results.

This thesis then develops the scalable and maintainable middleware, called *OrkhestraFPGAStream*, to answer questions about the efficacy of the dynamic dataflow approach on FPGAs marking the aforementioned second main segment. For evaluating the costs and benefits of using DPR to create a dataflow pipeline on the fly, we target two applications: 1) image processing; and 2) online analytical processing (OLAP) workloads of database management systems (DBMS). The example implementation, crafted using the generic framework, is adept at processing both workloads (concurrently if necessary). This capability ensures the evaluation aligns with the theoretical dynamic dataflow approach, rather than being influenced by problem-specific implementation nuances.

OrkhestraFPGAStream is built on top of related work that aligns with the previously highlighted assumptions of the framework. First, the underlying low-level HW implementation for executing the required operations has been done with resource-elastic modules developed by Kristiyan Manev for his PhD thesis [195]. These modules are integrated into a static system with floor planning, which incorporates regular routing backbones derived from the ZUCL [241] work. The regular routing backbone arrangement allows for flexible module placement, ensuring that interface locations are adaptable. Lastly, loading bitstreams onto the FPGA and reading and writing the memory-mapped registers and input and output data buffers has been done with a library called *Cynq* developed by Joseph Powell for the project FOS [305].

1.3.1 Remaining Chapters

Background (Chapter 2): This background chapter examines the general dataflow theory and what constraints must be met for practical use. Afterwards, the chapter also reviews how to build DPR systems in practice. The chapter concludes by looking at related work in building dynamic dataflow systems.

Runtime Management (Chapter 3): This chapter gives an overview of the general-purpose approach to dynamic dataflow processing. Mainly, the chapter explains the module and data placement problem in time and space. Consequently, we will define the constraints and objectives of such a general-purpose resource-elastic module scheduling problem.

Implementation & Evaluation (Chapter 4): The implementation and evaluation chapter will cover the technical details of managing and operating dataflow pipeline configurations for image processing and data analytics acceleration. With this, we examine the efficacy of a heuristical approach to scheduling the PR modules. Then we benchmark *OrkhestraFPGAStream* and compare its performance against comparable SW, static and dynamic alternatives' performance.

Generalisation (Chapter 5): This chapter will highlight how to make the proposed system generic and maintainable through an extendable module library with modular drivers. Furthermore, the chapter will evaluate ways to create module alternatives for trading between resource-performance-functionality benefits. For industry and the community to adopt this approach, we also examine the potential security risks and how to mitigate them.

Conclusion (Chapter 6): We conclude with the proposed methodology and its strong and weak points. Then the thesis will conclude with potential use cases, including open-source tools that can be used today due to work presented here.

1.3.2 Main Research Questions

1. How can a **dynamic** stream processing manager reuse **physically implemented** FPGA designs **transparently** while using their **resource elasticity** optimally?

2. How to approach an **NP-hard** scheduling problem of placing different task accelerators both in **time** and **space** with non-linear cost-performance relations?

3. How do the dynamic FPGA acceleration results compare with **static** solutions?
   (a) How significant is the penalty for using **partial reconfiguration**?
   (b) How much can the **scheduling time** be traded for the **quality of results**?

For the first question we highlight the various theoretical related design aspects that must be considered when designing a dynamic system that reuses previously physically implemented designs in Chapter 3. Then Chapter 4 introduces an example system that demonstrates how to address these design challenges in practice. Similarly, for the second question, we outline all constraints and objectives of the scheduling problem in Chapter 3 and evaluate an example implementation in Chapter 4.

Regarding the final question, we present various existing static solutions in Chapter 2. We then compare an example system using the dynamic approach with the static approach in Chapter 4. During this evaluation, we segment the runtime into distinct process durations. By comparing the relative durations of each process, we gain insights into the penalty of partial reconfiguration and scheduling in relation to the processing time of the loaded modules.

1.4 Publications

First author:

- **Conference papers**:
  - **Kaspar Mätas**, Kristiyan Manev, Joseph Powell, and Dirk Koch. Automated Generation and Orchestration of Stream Processing Pipelines on FPGAs. In *International Conference on Field-Programmable Technology (FPT)* 2022

    This work presents the results that show how a dynamic dataflow system can have at least comparable performance with a static system while surpassing the performance with large enough data sets where the processing acceleration overweighs the overhead costs. This work relates to the evaluation results that are presented in Chapter 4.
  - **Kaspar Mätas**, Tuan Minh La, Khoa Dang Pham, and Dirk Koch. Powerhammering through glitch amplification - attacks and mitigation. In *IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* 2020

A short paper which presents an attack that uses glitches for fast signal switching and to draw high amounts of power. As mitigation, a new virus signature was added to the virus scanner capable of scanning FPGA bitstreams. The attack method and the solution to defend against this when using 3rd party designs is discussed in Chapter 5.

- **Demo**:

  - **Kaspar Mätas**, Kristiyan Manev, Joseph Powell, and Dirk Koch. FPL Demo: Runtime Stream Processing with Resource-Elastic Pipelines on FPGAs. In *International Conference on Field-Programmable Logic and Applications (FPL)* 2022

    This work presents an interactable SQL interface with which analytical queries can be executed on relational database tables in PostgreSQL while transparently using FPGA acceleration with partially reconfigurable resource-elastic pipelines. The example system, *OrkhestraFPGAStream*, demonstrated in front of a live audience processing given SQL queries with the FPGA, is discussed in Chapter 4.

- **Workshop paper**:

  - **Kaspar Mätas**, Tuan Minh La, Nikola Grunchevski, Khoa Dang Pham, and Dirk Koch. Invited tutorial: FPGA hardware security for datacenters and beyond. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)* 2020

    This work presents various security attacks on data centre FPGA applications and how to mitigate them with a virus scanner scanning for multiple malicious patterns in the bitstreams to be executed. The idea of splitting the implemented design into a set of static and dynamic connections allows for the identification of various patterns. These patterns can flag potential security risks, as discussed in Chapter 5.

- **PhD forum**:

  - **Kaspar Mätas** and Dirk Koch. Transparent Integration of a Dynamic FPGA Database Acceleration System. In *International Conference on Field-Programmable Logic and Applications (FPL)* 2020

This work presents the idea that dynamic partial reconfiguration can be used to create dataflow pipelines on the fly to answer the problem of meeting acceleration requirements with conditions only known during runtime. The initial idea of transparent SQL acceleration using DPR is presented here in Chapter 4.

Involved as a collaborator:

- **Journal**:
  - Tuan Minh La, **Kaspar Mätas**, Nikola Grunchevski, Khoa Dang Pham, and Dirk Koch. FPGADefender: Virus Scanning for Multi-tenant FPGAs. In *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 2020

    This work concludes the potential attacks on FPGAs that can be detected with the virus scanner. The mitigation provided by the virus scanner is presented here in Chapter 5.

- **Conference paper**:
  - Kristiyan Manev, Joseph Powell, **Kaspar Mätas**, and Dirk Koch. byteman: A Bitstream Manipulation Framework. In *International Conference on Field-Programmable Technology (FPT)* 2022

    This work presents a bitstream manipulation tool that can quickly change a partially reconfigurable module location to be used in this thesis's proposed system with many supported FPGAs. The validity of the resulting bitstreams was tested with our dynamic system as the library evaluated in Chapter 4 is built with this tool.

- **Demos**:
  - Joseph Powell, **Kaspar Mätas**, Kristiyan Manev, and Dirk Koch. FPL Demo: FPGA Bitstream Virus Scanning. In *International Conference on Field-Programmable Logic and Applications (FPL)* 2022

    This work presents an updated version of the bitstream virus scanner capable of running on a larger variety of devices. This work demonstrates the implementation improvements of the initial ideas where the ideas are discussed in Chapter 5.

  - Tuan Minh La, **Kaspar Mätas**, Joseph Powell, Khoa Dang Pham, and Dirk Koch. Demo: A Closer Look at Malicious Bitstreams. In *International Conference on Field-Programmable Logic and Applications (FPL)* 2020

    This work demonstrates a denial of service attack on an FPGA and how the virus scanner can detect the attack while scanning the malicious bitstream. The scanning process is elaborated upon in Chapter 5.

- **PhD forum**:

  - Tuan Minh La, **Kaspar Mätas**, Khoa Dang Pham, and Dirk Koch. Securing FPGA Accelerators at the Electrical Level for Multi-tenant Platforms. In *International Conference on Field-Programmable Logic and Applications (FPL)* 2020

    This work presents the initial idea of what kind of undetectable attacks are possible with malicious FPGA designs and how they could be detected using a virus scanner. The attack surface that opens up when using third-party modules is discussed in Chapter 5.

Chapter 2

Background

In this section, we introduce all the required background information about the theory of dataflows. This is done in Section 2.1 which includes practical application examples. We will look at FPGAs in more detail in Section 2.2 to better describe all of the related work in Section 2.3.

2.1 Dataflow

Models of Computations (MoCs) provide an abstract view of how data processing is organised. When designing a system from the ground up, evaluating and comparing different MoCs helps observe and verify any inherent effects and limitations for the targeted application [75] (e.g., the cellular automata MoC as used in Conway's Game of Life is suitable for simulating biological processes due to reduced memory requirements [143]). Rather than just looking at the Von-Neumann model amongst Turing-complete MoCs, there is a large number of models with different characteristics (e.g., expressiveness, observability, compositionality and reconfigurability) from which the most fitting one must be chosen [127][1]. Mathematical frameworks have been proposed to classify and evaluate different models (e.g., Tagged Signal Model is used to characterise different operations on symbols [176]), and tools use these frameworks to characterise MoCs (for simulation purposes) [115] as different MoCs in practice are often combined [37]. As a result of this formal analysis, processing network systems are

[1] Even non-Turing MoCs exhibit beneficial characteristics, as shown by the following examples. The classification of different types of state spaces and processing times to be continuous or discrete [187] helps define natural computational models that fit different types of interactions in robotics [71]. Alternatively, there are attempts to extend the model of Turing Machines [50] to help model data containment models required for quantum computing [43].

found to be fitting for embedded applications to ease the integration of independent actors that work in parallel [175] and dataflow is a network MoC with coarse grain parallelism that fits the local and persistent memory resources commonly available on reconfigurable HW [17].

2.1.1 Definition

At its core, dataflow can be represented as a graph consisting of nodes that correspond to individual operation steps and arcs that indicate the data flow between nodes. The operations performed at each node have no side effects (no impact to other nodes) [137].

A generalised model for such distributed computations is Petri networks that symbolise available tokens (data) in *places* between the operation nodes [270]. In corresponding systems, first in, first out (FIFO) buffers are used for *places* where tokens reside until they are needed by the next PEs acting as operation nodes. However, these places can be shared between multiple transition nodes making the model non-deterministic. The alternative stricter Kahn Processing model creates operation *firing* rules (clearly defined and tracked point-to-point connections) to make the system deterministic [133]. Nodes of the system only fire tokens forward when tokens are present in their buffers. However, such systems may require unbounded buffer sizes, which are difficult to emulate on practical systems without multi-level cache hierarchies and hard-disk space [87], and as such, to avoid deadlocks, non-dynamic dataflows are used with static global scheduling [174]. These synchronous dataflow models are often used for static applications like digital signal processing (DSP) on embedded systems where the a priori static scheduling determines how many tokens can be fired for each node to work with bounded memory.

Dataflow systems need to consider both **producer** and **consumer** rates, taking into account the possibility of non-deterministic workflows. With static scheduling, it is possible to determine all potential rates beforehand. However, when dealing with non-deterministic workflows, it is necessary to monitor runtime behaviour and adapt accordingly. Regardless of the speed at which an operator node produces output data packets in comparison to the speed at which the consuming nodes accept them, the system must ensure that no data is lost despite having limited buffers. There is a large variety of *dynamic* dataflow models to handle this problem [36]. To avoid relying on centralised flow control, Manev's proposal for FPGA dataflow modules employs a credit system that utilises a suitable MoC similar to integer dataflow (IDF) [40]. These

credits use module-to-module signalling to coordinate the system (e.g., to convey information on the number of tokens that can be transmitted) [195]. However, to assure liveliness (prevention of deadlocks where no nodes fire), the graphs representing the dataflow execution plans must be directed *acyclic* graphs (DAG).

2.1.2 Examples

Dataflow MoCs can be used effectively in many applications that benefit both from parallel processing and the loose coupling of different processing stages. For instance, machine learning (ML) is an application field that fits the dataflow paradigm such that there are many specialised dataflow frameworks for languages (like Tensorflow [1]), for distributed computing (like Apache Spark MLlib [204] and Apache Mahout [14]) and for FPGAs (like FINN [30]).

Building these general-purpose dataflows is challenging as they must handle various infrastructure requirements such as out-of-order processing, system interfacing ("plumbing"), fault tolerance, graph processing, various data format parsing, encoding, and decoding. In this context, we will briefly overview the SW landscape of stream processing tools.

Here, the landscape of stream processing tools is mature enough that the industry has adopted numerous open-source tools, often from the Apache Software Foundation (here, marked in *italics*). Apache provides a supportive community and a transparent development process, allowing stream processing projects to become production-ready and enable fast time-to-market [15]. The most relevant stream-processing open-source tools under the Apache umbrella include:

- *Spark*: A Cluster computing system for large-scale data processing (one of the most popular big data frameworks).
- *Hadoop*: A framework for distributed storage and processing of big data using MapReduce (foundational algorithm for big data processing infrastructure).
- *Flink*: A scalable stream processing framework for both batch and real-time data processing (with exactly-once processing guarantees).
- *Beam*: A unified programming model for both batch and stream data processing pipelines (compatible with multiple execution engines).
- *Kafka*: A distributed, high-throughput publish-subscribe messaging system for real-time data streaming (often used with other frameworks like Spark and Flink).

- *Storm*: A distributed real-time computation system for processing real-time data streams (with a more accessible programming model).
- *Airflow*: A platform for programmatically authoring, scheduling, and monitoring workflows, including data pipeline workflows (a high-level monitoring framework).

The stream processing tools can be integrated with other Apache frameworks that offer data sources and sinks. Examples include *Cassandra* (for write-intensive operations) and *HBase* (for read-intensive operations), which are NoSQL databases with column-oriented data; *Flume*, a log data collection and analysis framework; and *Pulsar*, a high-scale publish-subscribe messaging system. These frameworks can utilize data storage formats such as *Parquet*, a general-purpose storage format, and *Arrow*, a high-performance in-memory columnar format. Most importantly, they can be customised further, such as adding stateful operations through APIs (e.g., *Storm's* Trident API), which provides support for window operations, stateful transformations, and aggregations, as well as built-in fault tolerance and state recovery. As there is such a vast degree of customisability with these open-source frameworks, other software stacks aim to give a complete package like Google's Dataflow [6] or Amazon's Kinesis [76].

These tools and other similar ones are mainstays for big data applications [210, 226] showing that the dataflow MoC meets the ever-increasing demands of data processing. Heterogeneous systems are well-suited for big data workloads, and there are various approaches to integrating these systems with FPGAs. For example, Xekalaki et al. [326] propose using TornadoVM with *Flink* for transparent acceleration. TornadoVM can virtualize heterogeneous hardware and manage high-level code execution on FPGAs via API calls, although the FPGA use suffers from long synthesis times [81]. However, currently, there are no tools available for managing dynamic dataflows on FPGAs, and therefore, we will examine two example applications that are suitable for execution on FPGAs.

Image Processing

For image processing, most operations fit FPGA acceleration [101]; for example, there are accelerators to encode image data (e.g., through trigonometric transforms [265]), which also allows embedding covert data via steganography [132]. As such, there is a rich set of dataflow frameworks specifically for image processing ([173, 35, 242, 131, 34]). Most image processing dataflows are deterministic and therefore are scheduled

statically. There can be different consumer and producer rates when the dimensions of the image or the pixel data encoding change, but these are usually derived beforehand from the image sizes. There are even branching dataflow graph topologies when different channels of the image data are processed in parallel, and later the results are aggregated. As such, image processing is an ideal application to to be accelerated with dataflow systems [141] and a good target for us to test generality given the various consumer and producer rates and varying topologies.

Palumbo et al. proposed a multi-dataflow composer (MDC) tool which automatically composes dataflow pipelines using any given HDL components in a module library, which was tested with image processing workloads [228]. However, this tool creates close-to-static dataflows as the different possible scenarios are executed by a reconfiguration process which entails switching between different routing options through non-programmable modules with MUX and LUT configurations which is only possible with overprovisioning. Such an approach fails to effectively support dataflows with data-dependent non-deterministic effects, for which we also look at supporting database management systems (DBMS) with a runtime manager.

Database Management Systems

Dynamic dataflow systems are used in applications like IoT that need to make data-dependent decisions [320]. In IoT systems, many moving parts produce and consume related data in parallel, just like in data analytics operations in DBMS [93]. Additionally, in streaming systems, user-friendly SQL queries create different windows (views) of the datastream without specifying *how* this should be implemented, leaving room for transparent optimisations [287]. As such, to reduce data redundancy and improve application development productivity, automatic DBMS systems have been adopted by the industry (already in the 1990s [94]) despite the difficulty of managing the complexity of such dynamic systems [104] that can handle a large variety of relations [250].

Figure 2.1 shows a generic view of the back-end of a DBMS system with the following processing steps before a query gets executed:

1. At first, the parser translates the query into tokens with syntax parsing for identifying keywords and checking for correct grammar, after which semantic parsing extracts the intended meaning behind the request.

2. Then, the tokens are reorganised into internal intermediate representation (IR) data structures so that the optimisers can process the tokens more efficiently.

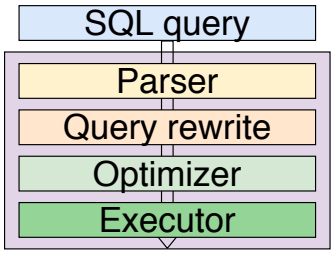

Figure 2.1: Any query gets processed in the steps shown in this figure. It is possible to extend or switch different process elements to work on different computing platforms (like FPGAs).

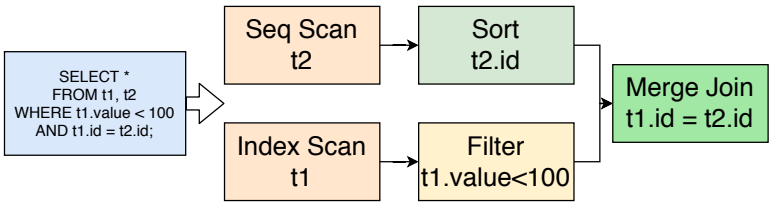

Figure 2.2: One or multiple dedicated HW modules can accelerate these database query plan nodes.

3. Finally, the optimiser selects the query plan (Figure 2.2) with the lowest cost for the executor to execute it.

There are two types of processing DBMSs perform. These are: 1) online transactional processing (OLTP) for frequent small-scale queries and 2) online analytical processing (OLAP) workloads for large-scale data analysis. With analytical workloads, the throughput is usually prioritised over latency and vice versa for online transactional processing to enable fast data insertions and deletions. However, the various industry standard benchmarks for different workloads (like TPC-C and TPC-E [48] for random transactional workloads and TPC-H [33] or more difficult examples like TPC-DS [247, 20] for analytical workloads) show that memory speed is the main bottleneck. Consequently, different data storing schemes (column-oriented, row-oriented, or a mix of the two) have been developed to improve data locality [4]. Storing table data in column-store fashion improves OLAP performance as demonstrated with MonetDB [119] and can be further improved with compression [350]. Compression can be done row-wise [248] or column-wise [258] with different approaches for different data types such as integer [113] and strings [310] while there are also approaches that allow operating on compressed data without decompression [343].

Recently memory speeds have also been increasing due to the broader adoption of solid-state drives, and with compression, memory has become more accessible to

facilitate in-memory databases and near-memory computing, alleviating the memory bottleneck [77]. Furthermore, DBMSs are starting to support more compute-intensive operations like built-in machine learning SQL functions [224], making DBMS systems more compute-hungry than ever. As a result, there is such a large variety of DBMSs, including non-relational databases (for example, for faster and easier document access and storage solutions) [55, 61, 63], such that surveys only list some examples from the actively developing field [105].

Specialist DBMS systems are designed for specific use cases. For example, in time series databases, fast data ingestion is prioritised, as is the case with systems such as QuestDB and Graphite [157]. Tools like Grafana and Elasticsearch (often used for data monitoring [27]) require standard interfaces for integration to effectively use these DBMS systems (providing support for a large enough selection of DBMSs). Standard interfaces like ODBC (Open Database Connectivity) and language-specific APIs like JDBC for Java and ADO.NET for .NET serve this purpose (although there is a performance cost when using these out-of-the-box [172]). Grafana leverages the ODBC data source plugin to connect to relational databases, providing a standardised interface for accessing and visualising data and enabling monitoring from multiple sources within a single platform, with technical capabilities including executing SQL queries, supporting a wide range of SQL operations, and creating custom SQL queries for data retrieval. These options to specialise, while having various integration options, lead to a rich and improved data processing world with a wide range of performance-enhancing frameworks for the target application.

Performance optimisations: DBMSs either target energy efficiency with mobile low-performance platforms [321] or performance with distributed database systems (through DBMS protocols like ODBC) [225] that require dynamic management [260]. MonetDB has also demonstrated performance improvements with a dynamic modular framework that interleaves optimisation and execution [32]. Therefore, the industry isolates different operations with their unique requirements to fitting platforms through higher-level DBMS (e.g., Presto in Facebook [276] or Spanner in Google [56]). However, the main contributor to these methodologies' effectiveness is the vast amount of work going into the query optimisers [169, 98]. In TPC-H, the most significant performance improvements, up to 30×, come from bringing filtering operations forward and flattening subqueries (while all other optimisation techniques have a combined improvement factor of less than three) [72].

CPU alternatives: After exhausting all of the CPU optimisation options in query optimisation work, which have resulted in very efficient SIMD instruction usage by the popular DBMSs [52], different GPU DBMSs are proposed for demonstrating additional acceleration [39, 344]. However, the GPU-accelerated DBMSs are still not flexible enough to present good enough performance speedups for larger adoption [53, 292, 49].

There are hundreds of SQL keywords in the SQL standard with varying functionalities other than the common SELECT, FROM, and WHERE keywords, like the latest JSON supporting operations [205]. FPGAs are used for accelerating the most compute-heavy DBMS operations associated with these keywords (e.g., machine learning operations in DoppioDB [7] or JSON filtering [100]) due to requiring less power while having comparable performance indicators [252]. The next step has been to use reconfigurable HW for more general-purpose query acceleration, either with static or dynamic solutions, while integrating with existing query optimisers [21] (like Apache Calcite in ReProVide [216]).

Static solution example: Most related work on DBMS query acceleration has employed operator over-provisioning due to static solutions like Q100 [325] or AxleDB on FPGAs [269] for serving varying runtime requirements. However, we can see that the dynamic nature of general-purpose DBMS systems drives static solutions to become inefficient or inapplicable. For instance, AMD supports an open-source database acceleration library with FPGAs consisting of static pipelines. Nevertheless, these pipelines cannot accelerate query 19 (Q19) from the industry standard benchmark TPC-H, and as of version 2020.2, AMD has dropped the direct support of TPC-H with its FPGA library for generalisation purposes [329]. The Q19 filtering consists of 12 disjunctive normal form (DNF) clauses. Any generic static pipeline supporting this query will require ample resources for this many filtering operations. However, the same filtering hardware would be wasteful for any other TPC-H query, and most queries use only a single DNF filtering clause. For instance, most other TPC-H queries use a single filtering clause. Ergo, there is no cheap one-fits-all static configuration.

A dynamic stream processing system that can adapt to different queries is the best approach to handle this complexity. Furthermore, such a dynamic system is more maintainable, allowing for the addition of more modules over time. Nonetheless, instead of focusing on building a rich library of modules adaptable to a wide array of

queries, the middleware introduced here addresses an orthogonal challenge that remains even with a comprehensive module library in place. This work lays the foundation by offering the necessary abstraction layers that facilitate the extension of the library without affecting existing accelerator modules and also ensures the seamless orchestration of these layers and modules' operations.

2.2 FPGAs

The topic of digital design falls outside the purview of this thesis, as there are ample resources available for studying FPGA designs (e.g., Serrano's "Introduction to FPGA Design" [275]). This study instead examines a more holistic view of how FPGAs can be integrated into larger systems.

FPGAs are integrated with a large variety of ways [31] (bump-in-the-wire, co-processor, network-attached) as they can connect to other external accelerators or data storage devices both directly or through a network. Nevertheless, the FPGA is almost always controlled by a CPU [47, 318]. For an approach with a tighter integration, often an AXI[2] interface is used in an MPSoC (like the AMD Zynq devices), dividing the CPU and FPGA into the Processing System (PS) and Programmable Logic (PL), respectively. Otherwise, a soft-core processor like the MicroBlaze can be added into the FPGA PL area as it can also serve as a control unit responsible for scheduling and reconfiguring the device [298] next to the other benefits of tighter integration like direct access to shared memory [268, 58, 257]. Instead of using the processor "bare-metal", using an operating system like PetaLinux that contains the necessary Linux kernels helps manage various communication protocols [283, 220, 227] without significant overheads (while also enabling using hypervisors [9]). For example, the FPGA Manager inside the Linux Kernel allows bitstreams to reconfigure the FPGA through the PCAP [170].

Bitstreams contain data [238] to program the FPGA resources that can be partitioned as shown in Figure 2.3. Digital circuits for FPGAs are designed with the help of electronic design automation (EDA) tools. The latest devices have become more heterogeneous given the large variety of hardened blocks available in the recent Versal

[2]The Advanced eXtensible Interface (AXI) is a standardised communication protocol commonly utilised in FPGAs and Multi-Processor System-on-Chips (MPSoCs) to facilitate the transfer of data between IP cores. The AXI protocol involves bus transactions between master and slave devices, with each transaction consisting of an address phase and a data phase. Among multiple alternatives, due to its scalability and high performance, it is an industry-standard protocol on ARM systems [244] and is the default bus interface in the IP integration frameworks of the vendors AMD and Intel.

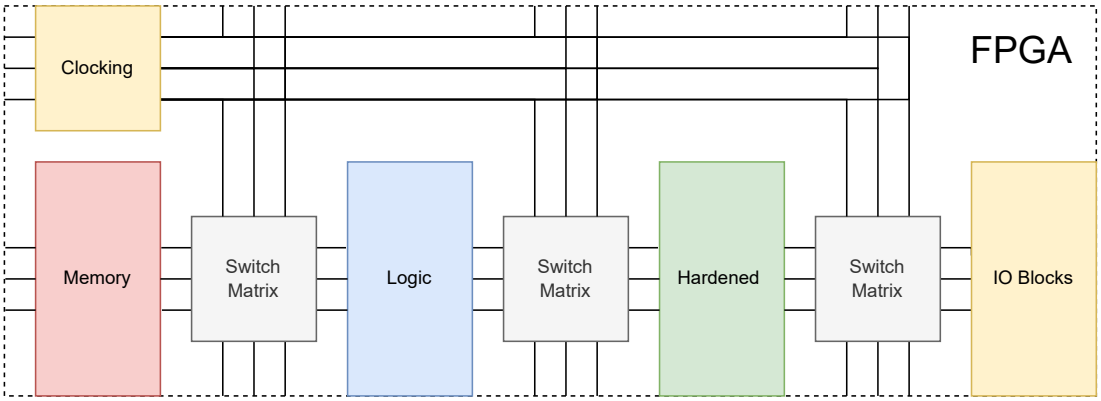

Figure 2.3: FPGAs are laid out as a grid of resources connected by programmable switch matrices for flexible routing.

FPGA family from AMD [82] to be more competitive against ASIC devices [162]. The synthesis process contains many steps (syntax parsing and translation, technology mapping the gate level netlist, place & route), gradually refining the IR to create the resulting bitstreams to program FPGAs optimally [277]. To make this process faster and more optimal, there are many different approaches [57] (that can even include ML [183] or approximate computing [271]). Nevertheless, synthesis is perilously slow (hours or even days) [167], leading to slow development turnaround times and, eventually, predominantly static designs.

We will look at an approach that circumvents slow synthesis times to improve productivity and flexibility. First, for many application domains where FPGAs are preferred over alternatives, the acceleration task could be modelled fully or at least partially as a dataflow operation [309]. There are many approaches to making dataflow programming a prioritised design concern for developers, including tools that extract IR for creating dataflow elements [136] and tools that turn IR-like Petri Nets into reconfigurable systems [323]. There are numerous examples built with MaxJ [202], a Java-based language where developers define dataflow graphs consisting of kernels and communication channels consisting of FIFO queues that eventually get compiled into optimised FPGA designs. Then, the virtualisation of all the different types of resources can be done at different system-level granularities using different-sized design blocks as demonstrated in by Zha et al. [341]. As a result, given a static system (e.g., Zucl [241]), the design blocks representing various dataflow processing pipeline stages can be synthesised independently and combined during runtime with *dynamic partial reconfiguration*, which is magnitudes faster than synthesising monolithic systems.

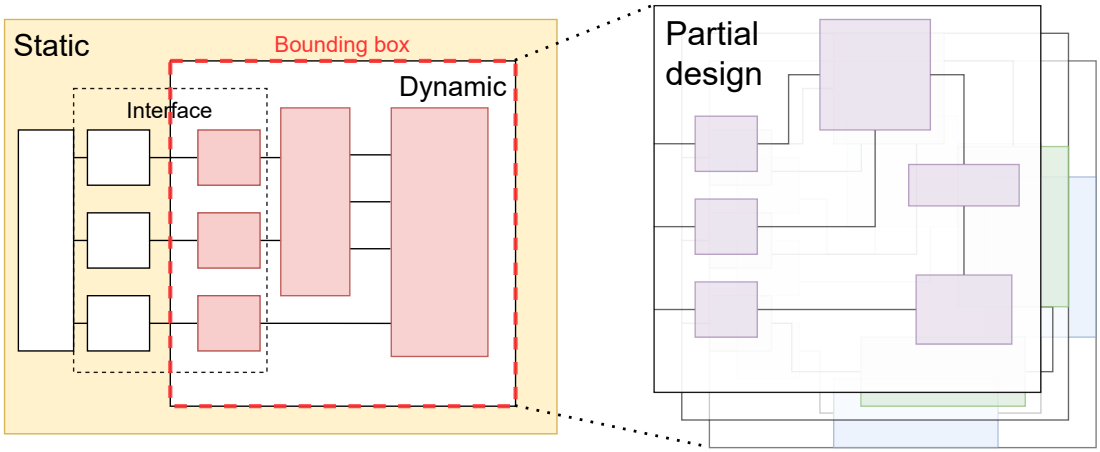

Figure 2.4: Various partial designs can be used on an FPGA, given that they are built into the confined region and have the required interface.

2.2.1 Dynamic Partial Reconfiguration

Dynamic partial reconfiguration is used in various applications [18] and is done with loading bitstreams containing partial designs as shown in Figure 2.4. These dynamic systems are built with different *shells* (static areas) and *roles* (partial designs) separated by various interfaces and bounding boxes [149] to avoid any "leaks" or erroneous signals reaching the rest of the system during the reconfiguration process as summarised by Vipin and Fahmy [312]. In the vendor workflow, the role is built incrementally on top of an already synthesised shell, which is useful for verification and fast compilation [231], but if there is an update to the shell, all of the dependent roles have to get resynthesised. Maintaining an extensive module library is unattractive with frequent updates to the static, and therefore tools like GoAhead create interfaces with "blocker macros" to be able to decouple roles and shells [24], which is also required for isolated designs [239]. This decoupling allows composing the whole system at the bitstream level without ever exposing any IP and seeing the netlist of the modules [140]. However, it comes with an increased verification cost overhead as timing verifications and floorplanning become challenging, which is why there are many proposals to automate floorplanning [336, 255, 73, 222, 25, 211].

Nevertheless, given the decoupling between role and shell, we can directly connect different independently synthesised partial modules, which has resulted in various approaches to floorplanning, as shown in Figure 2.5, that are implementable with current tools [153]. The first three approaches have all modules directly attached to the static system, while the last two approaches partition modules into smaller resource-elastic

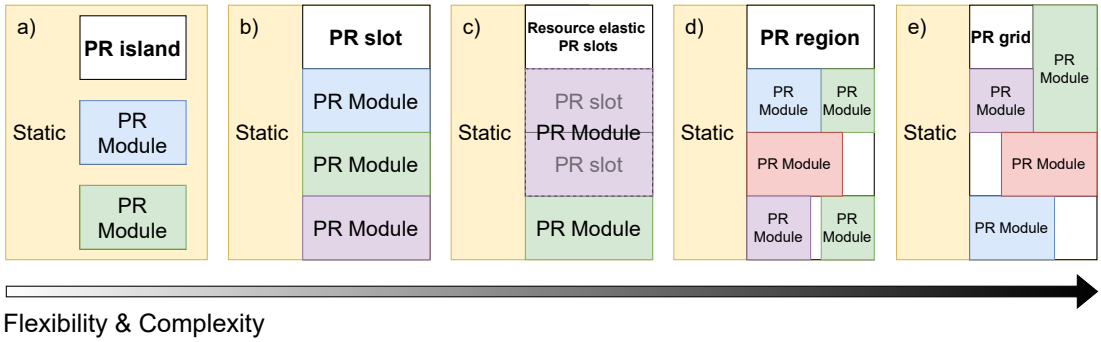

Figure 2.5: a) A common approach to designing PR systems is using PR islands where modules are designed for each location. b) With PR slots, modules could be relocated to other slots that have the same resource layout. c) With resource elasticity, larger modules can gain additional performance due to reduced fragmentation. d) The same resource elasticity can be used within a PR region, given direct module-to-module interfacing capabilities. e) All previous approaches can also be used in a grid where the interfaces between modules can be on all sides of a PR module.

PR slots within the PR region. As a result, module-to-module communication is required in approaches where there is no direct connection to the static system. As a next step, even hierarchical partial reconfiguration that puts PR modules inside other PR modules becomes a valid option for fine-grain changes like changing LUT functionalities [145] or for optimising CAD tool times [328]. This flexibility is necessary to increase the effective resource utilisation of the device by reducing PR module *internal* and *external fragmentation*, or in other words, limit resources becoming unused due to PR module boundaries and the heterogeneity of the FPGA.

Reconfiguration Performance

Despite the added complexity of floorplanning and designing matching interfaces between partial modules and static shells, the main common hurdle for using partial reconfiguration is the reconfiguration time [229]. A straightforward yet slow approach for loading configuration data into the chip is to use the PCAP reconfiguration controller through the FPGA Manager in the Linux Kernel (with which we achieved ≈270 MB/s). PR is slow due to going through a complex chain of software layers with the following steps:

1. The Linux kernel's generic FPGA manager scans the "/lib/firmware" directory for the bitstream file [296].

2. After detecting a new bitstream, the Xilinx Linux FPGA Manager driver takes

over, allocating RAM and loading the bitstream into it [332].

3. The Xilinx Linux firmware interface sends a signal to the PMU (Power Management Unit) Core Firmware, instructing it to execute the "fpga_load" command with the RAM address. The PMU Core Firmware is responsible for managing power states of the SoC and the programmable logic.

4. The PMU Core Firmware has a board specific Xilinx driver, which picks up the instruction, sets up the CSU (Configuration Security Unit) DMA with the RAM address, and transfers the bitstream in memory directly into the FPGA [333]. The CSU DMA is a high-speed direct memory access engine that enables data transfer between the system memory and programmable logic without involving the CPU.

In addition, reconfiguring an FPGA risks introducing new security vulnerabilities and potentially damaging (or even "bricking") the device if the process is not performed correctly. Validation of the new configuration bitstream and ensuring compatibility with existing hardware and software are used to minimise the risk of damage. To this end, a long chain of software processes is used to perform the reconfiguration process in a secure environment, such as the ARM trusted zone on the ZCU102 FPGA where the PMU firmware code is executed. This approach mitigates risks and ensure that the reconfiguration process is appropriately managed.

However, this reconfiguration could also be done on the FPGA itself, and then more lightweight OSs (e.g., FreeRTOS, a mature open-source OS alternative on embedded systems [95]) could be used for increased performance. Kamaleldin et al., [135] tested faster alternative approaches that also let the CPU work on other tasks during reconfiguration at the cost of using PL resources like the ZyCAP [311]. Using the ICAP has shown the highest reconfiguration speeds (even up to 2200 MB/s with older devices [103]) when loading bitstreams directly from DRAM [41, 74, 240].

Bitstream Manipulation

Another problem is that designs are position-dependent, and having an extensive module library can take too much space (one entire bitstream for AMD's VU19P is 0.2 GB [331]). However, bitstream relocation and compression reduces the module library space and speed up the reconfiguration [26]. Tools in the GNU Binutils like *objcopy* (e.g., for converting files from targeting one architecture to another through

endianness swapping) and *readelf* (e.g., extracting a symbol table for debugging) help read and manipulate binary code object files [80]; similarly, there are various tools for manipulating bitstreams. They enable bitstream manipulation (e.g., modifying configuration bit values for changing a frame's position) [263, 238, 192, 46] and various methods to compress and decompress bitstreams [253, 147, 161, 86] that can also be done on the FPGA for increased performance [134] and even integrated with soft-core CPUs for additional flexibility [117].

These tools help handle board-specific problems associated with relocating logic in the fabric, such as dealing with new clock regions that may be in new Super Logic Regions (SLRs). Given this progress in making partial reconfiguration more effective and productive, the next step is to provide a layer of abstraction over the low-level details and execute various operations transparently to the user as is done in general-purpose processing.

2.2.2 Resource Elasticity

Resource elasticity (changing the resource allocation of a task transparently to the user) with partial reconfiguration can be done after dividing the PL resources between different coarse-grain slots for a more straightforward approach where processes can be allocated to take more or fewer acceleration slots (as is demonstrated with FOS [305]). However, to allow direct module-to-module communications and to reduce fragmentation, we must use more fine-grain heterogeneous slots that provide a particular resource type. The presynthesised modules then use a combination of these slots. One way to define these regions filled with fine-grain slots is to use the fact that FPGA resources are homogeneous vertically inside a clock region and heterogeneous horizontally. We can confine modules into these resource columns with a bounding box like explained by Koch [144], and then the column resource footprint defines the module's placement constraints. With pattern matching between the resource footprint of the modules and the PR region, we have more fine-grain placement freedom as described by Grigore et al. [92] to reduce fragmentation.

A module library with modules that have different resource footprints enables resource elasticity and resource/performance trade-offs with the following features:

- **Module alternatives** implement the same logical function but with a different *functional capacity* (i.e., the maximum supported problem size or the maximum number of operands or operations supported by a specific module). For instance,

in Figure 2.6, the platform provides functional alternatives that match patterns of different sizes for a regular expression function. The choice between alternatives allows the runtime system to pick the optimal module for any given problem (minimising resource usage without compromising effective throughput).

- **Module composing** of accelerators provides the same function as the individual modules to feature a greater aggregated functional capacity. For instance, to serve a string matching request with 42 characters, we may use one match module for 32 characters and another for 16 characters together (for reduced overprovisioning or handling fragmentation as shown in Figure 2.7).

- **Module variants** implement a specific function but with different physical resource footprints (e.g., Figure 2.8 shows 3 module implementations for an arithmetic function that: 1) uses once a DSP column in the left, 2) a DSP in the right, and 3) uses logic only). Module variants allow for a tight module packing on heterogeneous FPGA resources (i.e., the columns of logic, memory, and DSP primitives) at the expense of multiple module bitstreams.

All these options with flexible routing between the modules enable **resource-elastic stream processing**, which aims to maximise the utilisation of available resources for a given runtime problem. Note that resource-elastic stream processing may use module variants, module composing, and module alternatives arbitrarily together.

2.3 Related Work

Stream processing MoCs fit FPGAs. For instance, Maxeler's MaxJ supports stream processing accelerators with orders of magnitude speed-up over software variants (e.g., [215, 155, 84, 300]). Consequently, Vesper studied various previously mentioned isolated parts required for a stream processing system such as having dedicated PR regions within a static shell (like ZUCL [241]) that is designed with floorplanning tools (with GoAhead [24]) that are to be used in conjunction with bitstream manipulation tools (like Bitman [238]). As a result, Vesper highlighted the benefits of resource-elasticity and the complexity of permuter modules that can reformat a stream in dataflow pipelines [308]. However, that work lacked a SW stack and a sufficient number of HW modules to evaluate accelerating complex applications, as is the case for most related work. Ergo, most stream processing systems are static, albeit with

some notable exceptions like the following three systems demonstrating video processing applications. Cattaneo et al., [44] swaps between statically compiled processing pipelines, while Kritikakis et al., [159] additionally supports the stitching of relocatable accelerator bitstreams, but is constrained to Maxeler's HW and MaxJ compilation. However, these and similar older works like the self-adapting system proposed by Oettken et al., [223] (which used the ReCoBus [148] tool for providing different sized interfaces for dynamic modules) use older more homogenous boards from the Virtex family, and more importantly lack SW integration capabilities.

In data analytics, many static compilers like Glacier [213] decompose the queries into different operator modules. Then it is possible to automatically schedule and compile queries to static pipelines (as is proposed by Müller et al. [212] and Sadoghi et al. [267]) including the solution from Minhas et al. [207] that deliberately uses overprovisioning to provide the flexibility to accelerate batched queries. In the following paragraph, we look into why overprovisioning is prominent in related work targeting dynamic stream processing applications and how it can be avoided.

Operations: Papaphilippou et al. [230] and Fang et al. [77] survey database acceleration using FPGAs and confirm the dominance of static database acceleration systems for queries known at design time. Therefore the industry provides many static FPGA solutions from the HLS accelerators provided by the Vitis Libraries [329] to a more limited subset of operations like (de)compression, join, and data filtering provided by the systems developed by Netezza [79], Accelize/Xelera [327], and Swarm64 [293]. For more flexibility, alternative approaches provide complex programmable processing elements [209] or even specialised CPU cores [110] defeating the purpose of using specialised accelerators. Further specialised data analytics problems that FPGAs have accelerated include operations for calculating the stochastic gradient descent and solving the skyline problem in doppioDB [280], accelerating the k-means algorithm [111], or OLTP processes like insertion and indexing operations in the BionicDB [142]. More extensive algorithms like linear regression can be broken down to atomic instructions to be accelerated individually [189].

Integration: Hoozemans et al. [114] touches on the rest of the aspects of DBMS acceleration missed by the other surveys, such as integration. For streaming pipelines, first, the data blocks have to get fetched from memory using specialised DMA infrastructure proposed with Fletcher [237] and Ibex [324]. Then in big data and in-memory

databases domains, various data encoding schemes exist for which FPGA conversion acceleration has been proposed [234, 235] including specialised streaming encondings [236]. Consequently, we see numerous module requirements and a vital engineering question for such rich libraries: *how these modules can be used interchangeably in a stream processing pipeline to avoid unnecessary overprovisioning*.

Overlays: Overlays provide a layer of abstraction over the highly heterogeneous resources on FPGAs to increase the productivity of developing solutions using the extensive customisability of FPGAs [284]. They can resemble virtual FPGAs, GPUs, CPUs, CGRAs, or DSPs, that are more customised towards the targetted application and can use time-multiplexing to switch between different configurations [179]. Time-multiplexing alleviates the inherent overhead cost of such virtualisation amongst other efforts [146]. Mbongue et al. [203] use application-specific dataflow overlays for better performance, which are derived from compiled algorithms while stitching optimised modules with RapidWright [171] (a tool allowing netlist manipulations for recent Xilinx FPGAs). However, instead of RapidWright, PR can be used to switch between various modules and the overheads of overlays can be avoided.

PR infrastructure: ReProVide [217] uses PR accelerator "islands" to switch between different operator modules. Becker et al. [22] proposed an automated approach to find the maximal number of such PR "islands" while minimising fragmentation. From another perspective, Koester et al. [154] measured the popularity of resources and placed modules such that the most popular regions are preserved in order to be able to get overall more placement requests served. Finding different performance/resource tradeoffs for designs can be quickly done with HLS as demonstrated in [345]. Consequently, there have been runtime systems proposed a long time ago already that use partial reconfiguration [313], and there also have been heuristics developed based on the generalisation of the problem by Danne et al. [62]. Nevertheless, the required crossbar connecting the wide datapaths between the PR "islands" and any memory or control subsystem is expensive and causes congestion. Another less expensive option is to allow direct streaming between exchangeable accelerators in PR regions as proposed by Ziener et al. [348]. However, these dynamic stream processing systems with direct connections [308, 195, 348] fail to propose a matching software stack [123].

Dynamic stream management: For GPUs, for example, the Ocelot/Hype platform provides transparent integration, which helped to create the CoGaDB [38] optimiser

as GPU programmability is more mature than for FPGAs. Following such a virtualisation approach on FPGAs, Vaishnav et al. [304] showed better performance in FOS (the FPGA Operating System) by using multiple adjacent regions combined with a scheduler that can utilise several instances spanning over different regions depending on the load and resources available. Meanwhile, DeHon et al. [66] developed SCORE to schedule stream computations for reconfigurable execution with time multiplexing in a hybrid architecture consisting of microprocessors and grids of reconfigurable regions. However, as opposed to systems with direct module-to-module communications ([195, 348, 134, 338, 152, 148]), both [304] and [66] lack the fine-grain placement freedom as defined by Grigore et al. [92] to reduce fragmentation. Combining the time-multiplexed scheduling for stream computations from SCORE and the resource-elastic scheduling from FOS while using fine-grain resource partitioning, our runtime system can further optimise resource usage for a large variety of *data-dependent problems only known at runtime*.

Potential: There is an open question on what the system architecture using FPGAs should look like for DBMS (and general stream processing) use cases. For instance, Gustavo et al. [8] proposed Enzian with a 30 GB/s cache-coherent datapath between the CPU and the FPGA, while both of them have access to DDR4 memory and PCIe and Ethernet connections. This architectural layout allows experimenting with different designs like the Intel Xeon + FPGA system (proven to successfully accelerate CNN workloads [51]), but it still has lower bandwidth with the main memory than OpenCAPI-enabled FPGAs with Power9 CPUs from Nallatech [219]. With systems like these, an in-memory DBMS speculatively could be accelerated with a throughput of 100 GB/s using our system if scaled up correctly using techniques like those proposed by Manev et al. [191, 194]

2.4 Chapter Conclusion

To conclude, we introduce a high-level discussion summarising the main points of this chapter. First, in Section 2.1, we looked at the underlying theoretical Dataflow MoC. We concluded that given a suitable set of dataflow operations and corresponding modules, heterogeneous computing platforms can be naturally employed to optimise memory usage, especially in dynamic dataflow systems. We further highlighted how image processing and data analytic workflows suit the Dataflow MoC and how the

optimisation aspects available on CPUs have been explored thoroughly, whereas the field of using alternative computing platforms has not matured as much.

In order to explore the optimisation avenues offered by FPGAs, we delve into their capabilities in Section 2.2. We can conclude that FPGAs can be used in a variety of ways after highlighting that FPGAs can be used standalone or attached to a CPU, and the PL can be used statically or modified with PR (through either PCAP or faster configuration ports like the ICAP). When using the FPGA statically, commonly an exclusive design is synthesised to the targeted problem, and when another input problem is targeted a whole new design is synthesised, which suits workflows where the requirements rarely change. For more dynamic workflows, either more generic static designs are used (introducing overprovisioning and therefore requiring larger devices), or multiple static designs are considered that are swapped with whole board reconfiguration (introducing large reconfiguration times and increasing latency).

For use cases demanding even greater dynamism, we propose leveraging partial reconfiguration. Reconfiguring a smaller area of the device reduces the reconfiguration penalty. However, the increased complexity of creating systems that use PR, creates both operational and design-time constraints that are not present in the more static approaches. When partitioning the FPGA device into shell and role areas, HW engineers face challenges such as congestion and fragmentation overheads. Furthermore any interfacing also introduces overheads. As a result, there is a scale for how dynamic the targeted problem is, which should match the scale of how fine-grain the partially reconfigurable areas are on the system under design that is optimised against the targeted problem. These scales are analogous to mapping problems to their suitable computing platforms that become more specialised in the order of CPUs, GPUs, CGRAs, FPGAs, and finally ASICs. Various optimisation methods are available on all levels on the FPGA (from using static designs, overlays, PR islands, to using PR regions with fine-grain slots). Consequently, this thesis investigates where the break even points reside between using the fine-grain approach of daisy-chaining PR modules within a PR region and reconfiguring the whole coarse-grain PR region in Chapter 4.

Lastly, in Section 2.3, we investigated the related work that leverages partial reconfiguration. At a higher level, we conclude that no existing system simultaneously 1) provides an abstraction layer similar to a software integration framework, and 2) is capable of scheduling and orchestrating multiple incoming requests while reconfiguring a daisy-chained set of resource-elastic modules in PR regions on modern FPGA devices, as outlined in DBMS-specific Table 2.1. Resource-elasticity can be gained with

Table 2.1: Comparison of DBMS-specific related work on FPGA acceleration for core data analytics operations. Few works utilize FPGA's PR capabilities, and none offer both SW DBMS integration and resource-elasticity.

Related Work	Direct Module-to-Module Interfacing	SW Integration	Swappable Modules	Resource-Elasticity
doppioDB [280]	×	✓	×	×
Ibex [324]	×	✓	×	×
ReProVide [216]	×	✓	✓	×
Wang et al., work [315]	✓	✓	×	×
SQL2FPGA [184]	✓	✓	×	×
Glacier [212]	✓	✓	×	×
AxleDB [269]	✓	✓	×	×
Manev's work [195]	✓	×	✓	✓
Ziener et al., work [348]	✓	✓	✓	×

synthesised static pipelines, however in Table 2.1 runtime resource-elasticity (using prebuilt solutions as introduced in Section 2.2.2) is provided only by Manev's work. The same conclusion can be reached outside the scope of DBMS acceleration systems. In the DBMS-specific works, even if a SW integration approach is described and provided it is often not generalisable as it is DBMS specific. As a result, next we look at the design challenges of creating such an abstraction layer and its subsystems for *OrkhestraFPGAStream* in Chapter 3 (to tick all boxes in Table 2.1).

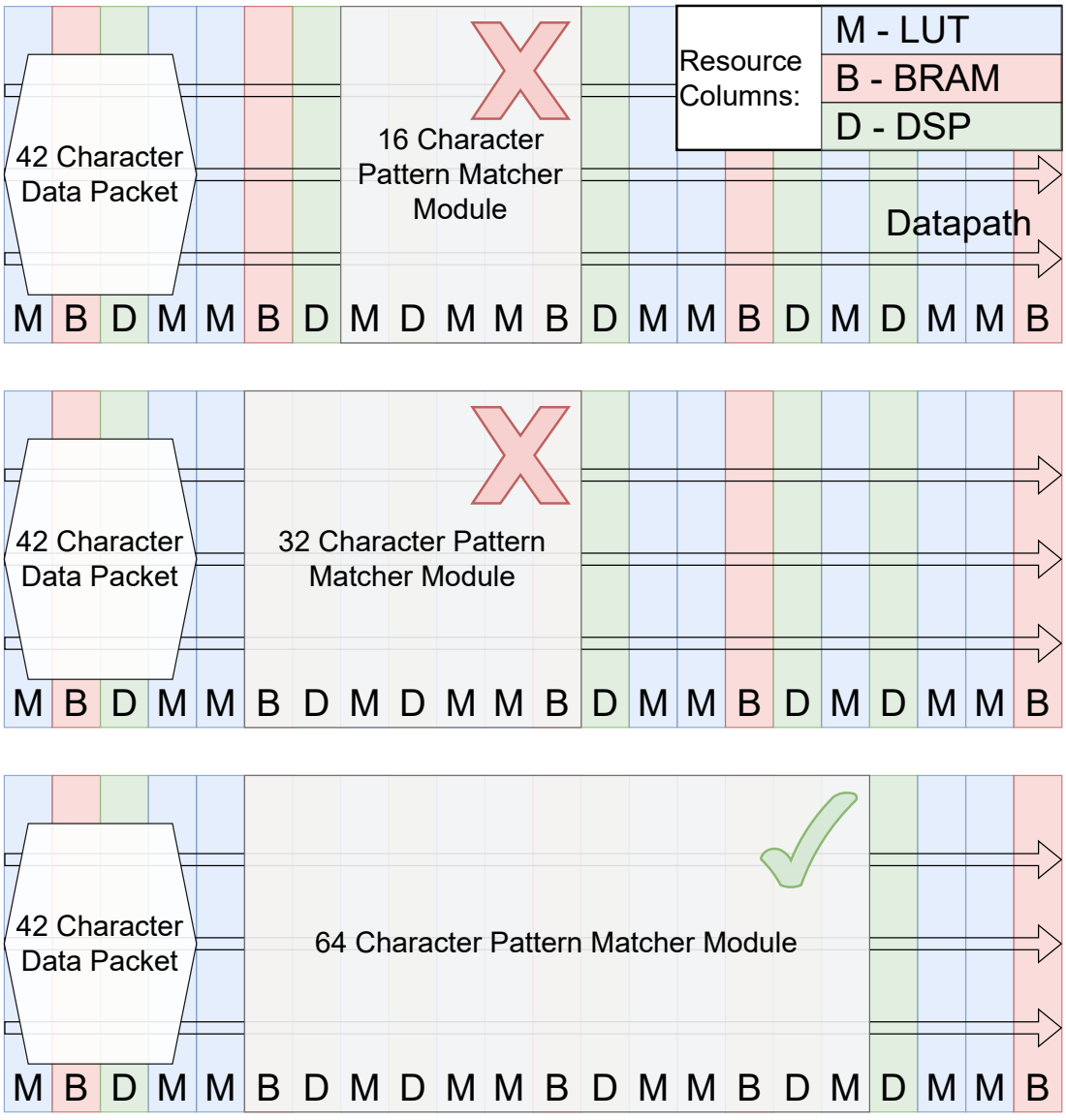

Figure 2.6: Only the largest pattern matcher module with sufficiently large functional capacity can execute the required operation on the incoming data packet.

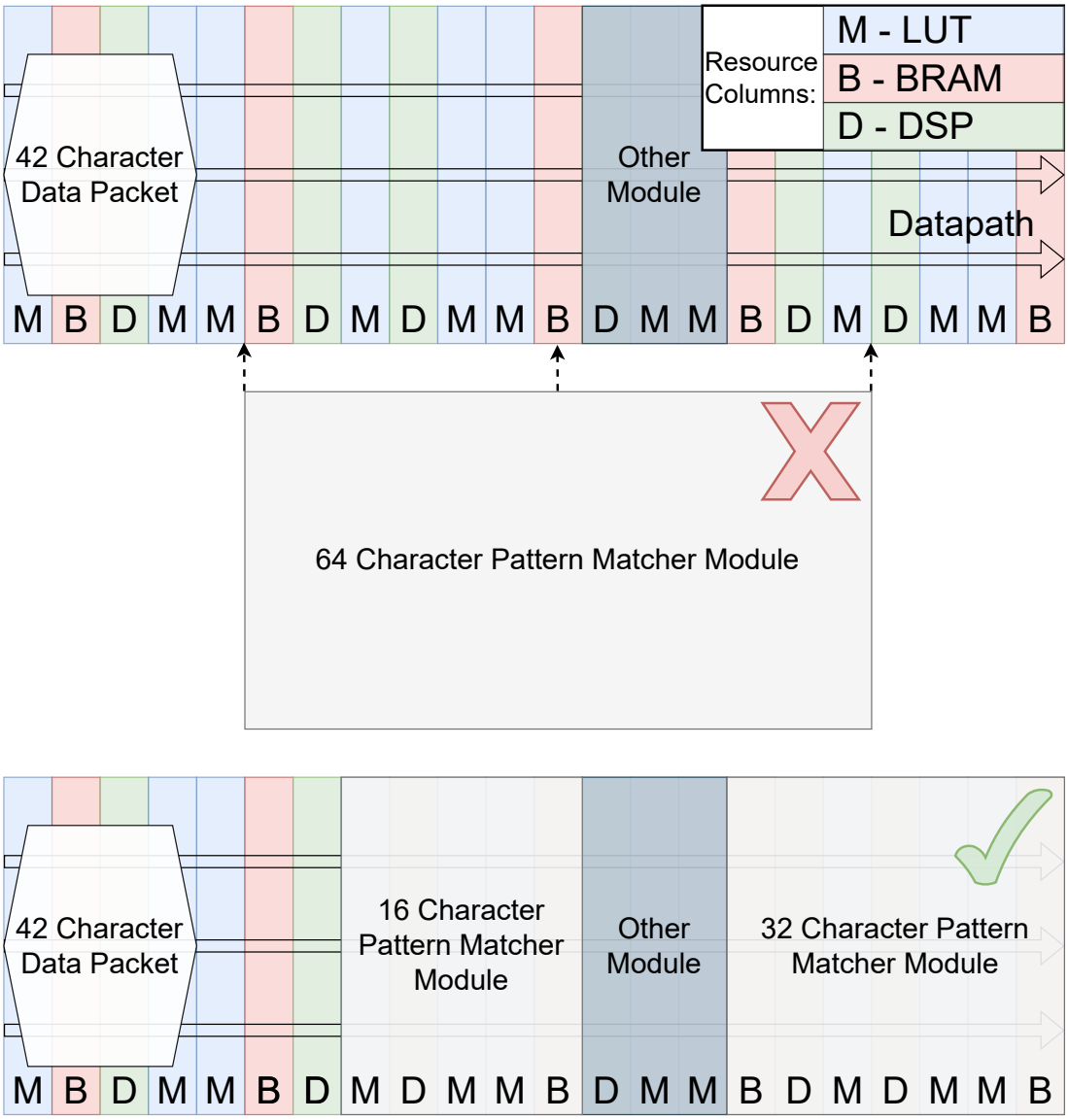

Figure 2.7: If the largest pattern matcher does not fit with the already placed modules then smaller modules can be composed to meet the requirements set by the incoming data.

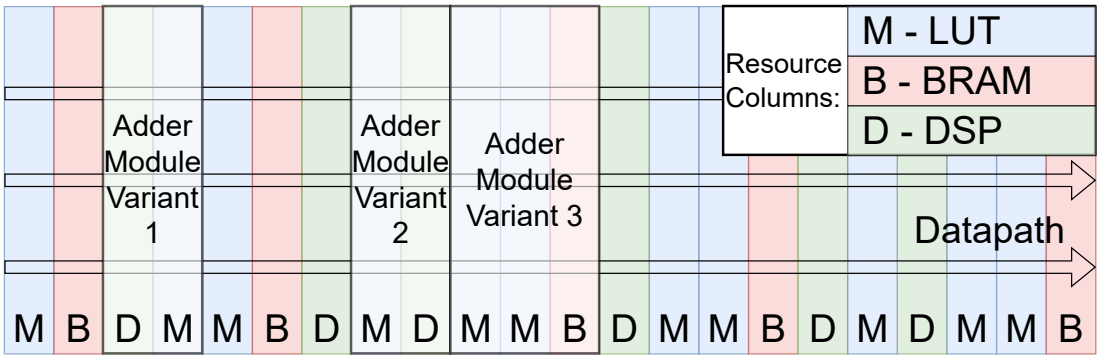

Figure 2.8: A module library can have multiple adder module variants to have more placement options.

Chapter 3

Runtime Management

This chapter introduces all of the parts necessary to manage dynamic pipelines in Section 3.1. After this, we will concentrate more on general data management and scheduling in Sections 3.2 and 3.3 respectively.

3.1 System Overview

In distributed systems, complex systems monitor workloads and adapt the networking and resource allocation to maximise performance. Likewise, to maximise resource utilisation for a wide range of queries on a single device (and in the future on multiple configurable devices), we need a runtime manager that monitors potentially non-deterministic operations to choose and set up the ideal accelerator pipelines. Such a system, as depicted in Figure 3.1, has an extensive degree of freedom to place any operator accelerators spatially and even temporally, if required, on the FPGA. Additionally, to further improve resource utilisation, the system uses resource elasticity that results in significant complexity and requires studying viable ways to manage the added complexity effectively.

As the scope of the system is general stream processing, the first step is to enable various application-specific customisations by designing a modular system shown in Figure 3.2. In general, such a system can be partitioned to the following subsystems:

- **Memory management** - To allocate and keep track of reads and writes to all data blocks in the main memory.
- **Data management** - To encode or decode various data types in SW or HW, which involves resolving conflicts when placing data on the datapath.

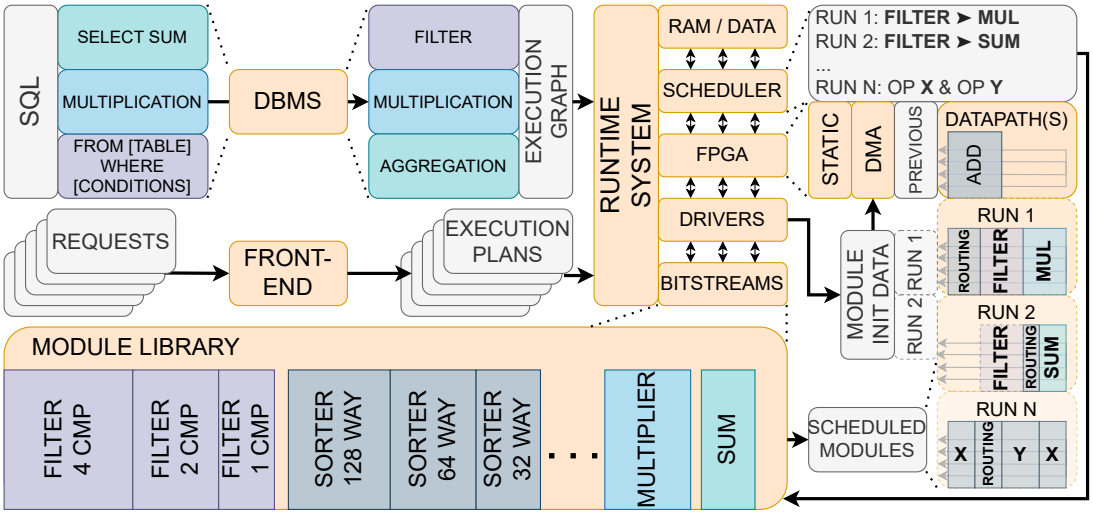

Figure 3.1: Acceleration ecosystem reusing physically implemented modules. The system must support various front-ends (e.g., DBMS) that handle parsing acceleration requests and map the required operations to available modules. The runtime system manages data and memory while scheduling requested operations and loading bitstreams, fitting modules and routing blocks accordingly. Furthermore, to limit reconfiguration, the modules are reusable and programmable for various use cases, which requires a corresponding set of drivers.

- **Scheduling** - To find the optimal set of operator pipelines and minimise end-to-end execution runtime while considering the limited set of resources that enforces scheduling both in time and space.
- **Input parsing** - To parse acceleration requests in any form and map them to available modules.
- **Execution management** - To program and operate all of the HW modules in the correct order.

Note that we could switch any of these parts during runtime when providing different services or use pre-computed services for more static use cases. In this context, a 'service' is a higher-level concept, decoupled from specific subsystems and modules. It can be predefined or dynamically compiled during runtime, catering to both static and dynamic environments. In our later evaluation, we primarily address module swapping, leaving subsystem changes for discussion in Chapter 5.

Efficient management of different memory access patterns is crucial for minimising I/O costs [214]. However optimising memory management subsystems is dependent on the platform, and is out of the scope of this thesis as this has been explored by related

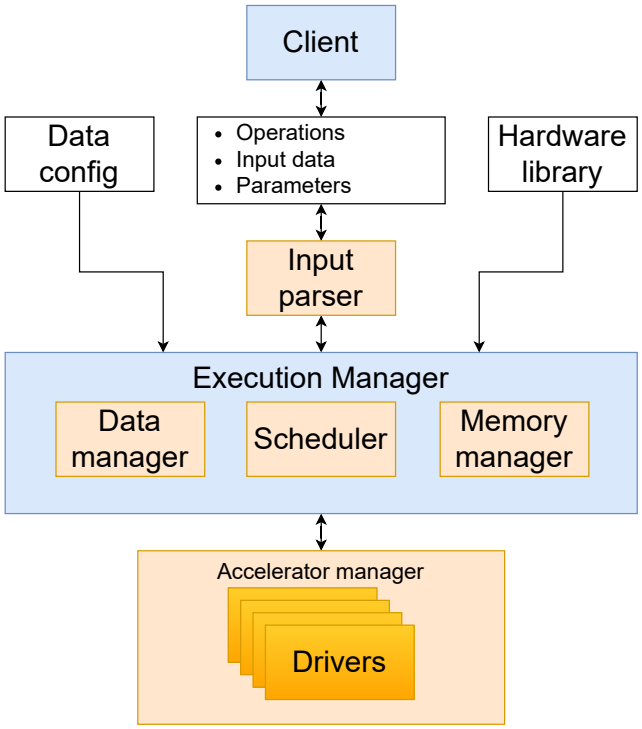

Figure 3.2: A modular system is required to handle the large degree of freedom to facilitate various optimisations for application-specific use cases. All system parts shown in yellow can be exchanged for optimised scheduling, memory management, or execution strategies that may fit different target applications or devices.

work (e.g., for AI, an application-specific context [109], and FPGAs, a device-specific context [193]). Therefore we will look at how data can be managed on a wide datapath, define the scheduling problem next in this chapter, and leave the implementation-specific details of input parsing and execution to the following evaluation chapter.

3.2 Data Managing

Given their slow clock rates, FPGAs require wide datapaths to maintain high throughput. In high-performance stream processing, automated management is necessary to achieve transparent optimisation and exploit the wider datapaths. On-the-fly data management requires considering the following:

- **Data encoding** - How to support a variety of encoding approaches, both in terms of data types and how to place them onto the datapath?

- **Large data packets** - How to deal with large packets given limited HW resources?
- **Control signals** - How does flow-control have to be managed?

3.2.1 Datapath Layout

The primary concern of data management is how the HW and SW interpret the data, as it can be encoded in various ways - optionally with metadata and potentially in a compressed format. Depending on the module library (just like with any other operations), the (de)encoding and (de)compression can be done in HW or SW or even while using both domains. The crux of the idea behind this choice is: *depending on the modules in the HW library, the scheduler must be able to reach optimal solutions in any way possible*. Therefore on the data management side of things, the resulting system has to be able to describe data on a higher level abstraction while using different data types to provide the required flexibility. These custom data types can be application specific, like large blocks of pixel data, as is the case for image processing.

However, the datapath is not infinitely wide - whereas data packets can be (depending on the data type). Problems that cannot be resolved in *space* must be resolved in *time*. Usually, given the ubiquitous I/O bottleneck, any practical processing elements are likely to operate at a higher throughput than the arrival rate of new data. Therefore, the system has some leeway to process data packets using multiple clock cycles without sacrificing overall performance.

Chunks

Stream processing modules that can operate on larger data packets (contrary to on-the-fly updating of smaller values) have to have buffer space in dataflow systems to process batches of data packets that can arrive over multiple clock cycles. In order to differentiate between multiple data packets and simply larger data packets arriving one after another, Dennl et al., [68] coined the term *chunks* for a similar dataflow system that is built on-the-fly with reconfigurable modules (however without resource elasticity and an integrated runtime). A data packet consists of one or multiple chunks, and modules are initialised with chunk counts to know how many clock cycles are required to read the targeted packet with a given datapath width. If a module is only required to process a small subset of a large packet, then enumerating each chunk is required to target a specific offset on the datapath marked with the targeted chunk ID.

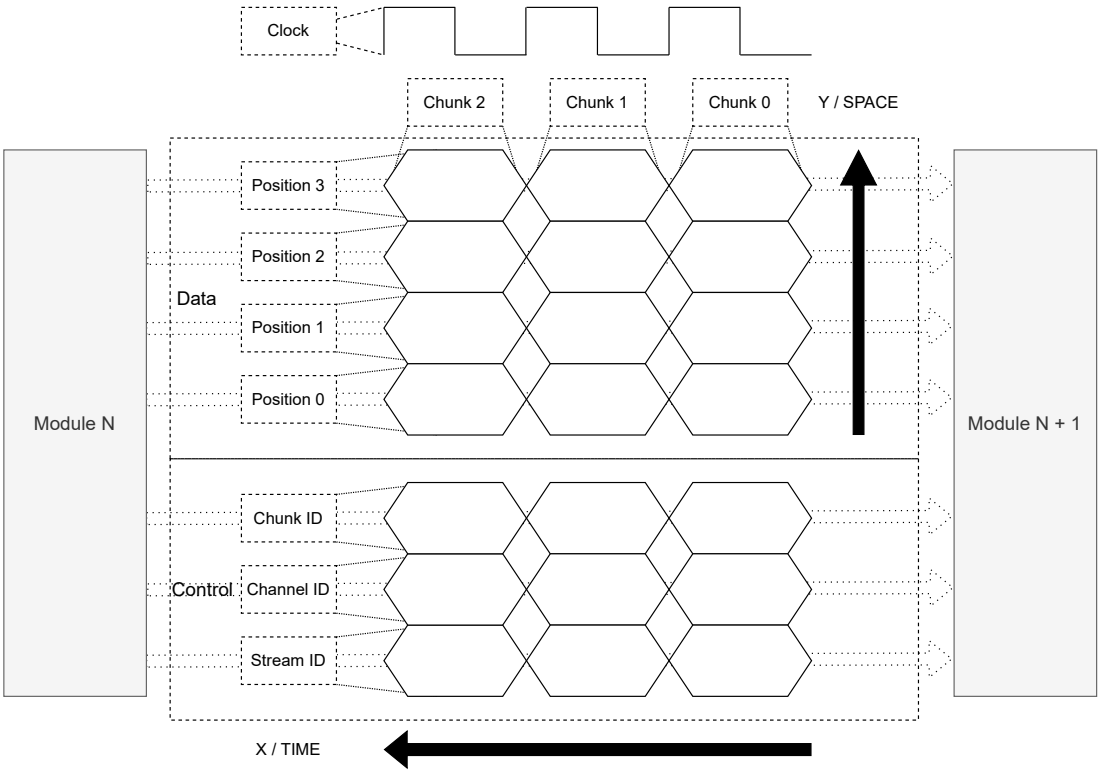

Figure 3.3: The different positions are for selecting different wires on the wide databus, and chunks help select correct data packets arriving at some specific clock cycles. The control signals show which chunk and stream that data packet is from to allow out-of-order data transfers. The channel IDs are for further substream indexing when virtual streams are required within a stream.

Virtual channels and streams

The same is true for parallel streams. When multiple modules share the same datapath but process independent streams, they must target specific streams while passing through other streams unaltered. Therefore, streams must also be labelled with stream IDs. All labelling data required to be sent each clock cycle require parallel wires on the datapath as shown in Figure 3.3. To achieve additional optimisations in the HW modules, an additional label is needed for indexing virtual streams within streams (particularly important for large merging operations [191]).

In conclusion, dataflow systems need a way to label data packets arriving at specific clock cycles and also how the various data packets relate to each other with additional stream and channel labels. This labelling allows the incoming service request to be decoupled from the physical acceleration implementation details. The HW modules do not need to have any conceptual knowledge about the type of work they are doing

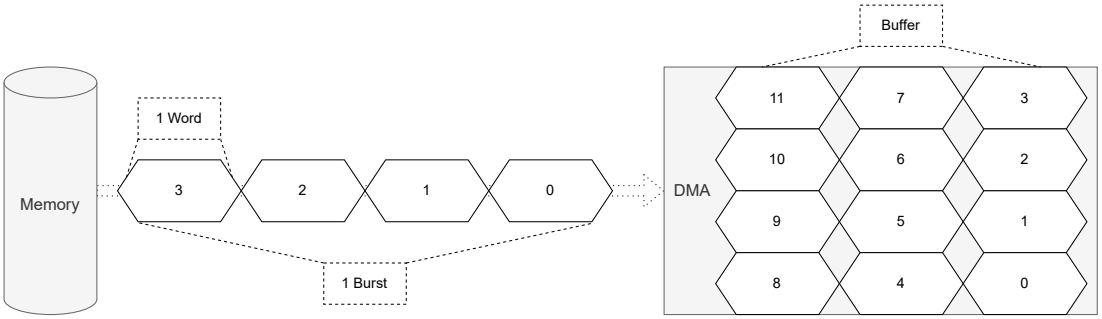

Figure 3.4: Data bursts received from memory get sequentially packed into buffers inside the DMA module.

when they are only targeting data at certain offsets in the datastream labelled with the correct ID labels. Likewise the incoming service request does not have to know anything about the underlying HW constraints (e.g., how wide are the datapaths). It is the middleware's job to map the application specific requests to match the physical constraints of the available accelerators with these labels to virtualise the data streams and channels from the physical connections. The drivers paired with the modules in the module library do this mapping. A module library example is introduced in the next Chapter. Before looking at implementation details, we need to also understand 1) how the data payload streams are laid out into packets transferred each clock cycle by the physical data mover module; and 2) how the modules that receive these are initialised and scheduled.

3.2.2 DMA Module

In order to rearrange data blocks from a linear stream to a wide datapath with the required labels, the dataflow system requires an initial data mover module as shown in Figure 3.4. This data mover module with direct memory access (DMA) requests data from specific memory addresses to be written to free available buffer space while streaming data to the modules in the datapath from other buffers already filled up. The exact process is happening the other way around when writing results back to main memory, where data arriving from the modules is written to free buffers while data from filled-up buffers are being written to given memory addresses. Consequently, the DMA module must be specifically configured to use optimal data burst sizes given data packet size specifications to maximise performance against the given memory subsystem and buffer sizes as shown in Figure 3.5. These configurations and buffer partitioning between different streams must consider the ongoing processes on the device that

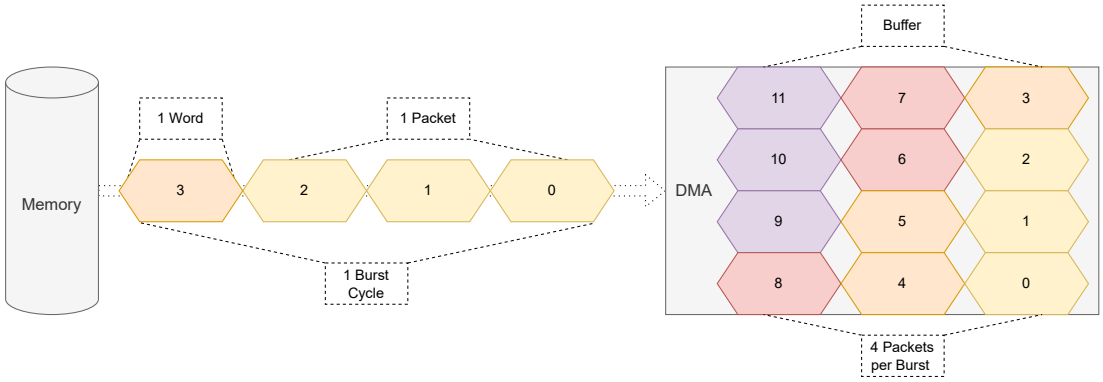

Figure 3.5: The DMA is configured to use optimal packet sizes and burst sizes to minimise data alignment issues (e.g., a stream of 3-word packets must be fit into buffer without alignment issues). This configuration considers the available buffer sizes, memory subsystems, and other potential systems interacting with the main memory to optimise data flow. Consequently, it is possible to read and write to multiple buffers transparently with data pre-fetching.

might affect the I/O performance.

In order to enable further HW module optimisations and to exploit the fact that all modules have buffering capabilities - data can be placed on the datapath *out of order* as shown in Figure 3.6. However, as the datapath has limited width, this out-of-order placement entails placing words of data in the time and space domain. Figures 3.8 and 3.7 show this flexibility in data placement. Additionally, the ability to buffer and place data allows data duplication, as shown in Figure 3.9.

It is again the job of the middleware to coordinate the data streaming between the memory and available buffers. The datamover module itself also must have a corresponding driver where the aforementioned labelling can be programmed into the data mover module during its initialisation process to prepare for the subsequent streams to be processed. In order to differentiate between transferring payload data and module initialisation instruction data, there are control signals the modules must recognise, as described next.

3.2.3 Control Data

The modules must have buffer space to also handle varying producer-consumer rates in a dynamic dataflow system. There are various flow control signal wires parallel to the primary data wires to keep the dataflow operational. In order to avoid buffer overflows and deadlocks, the modules use a **credit system** where each module has a

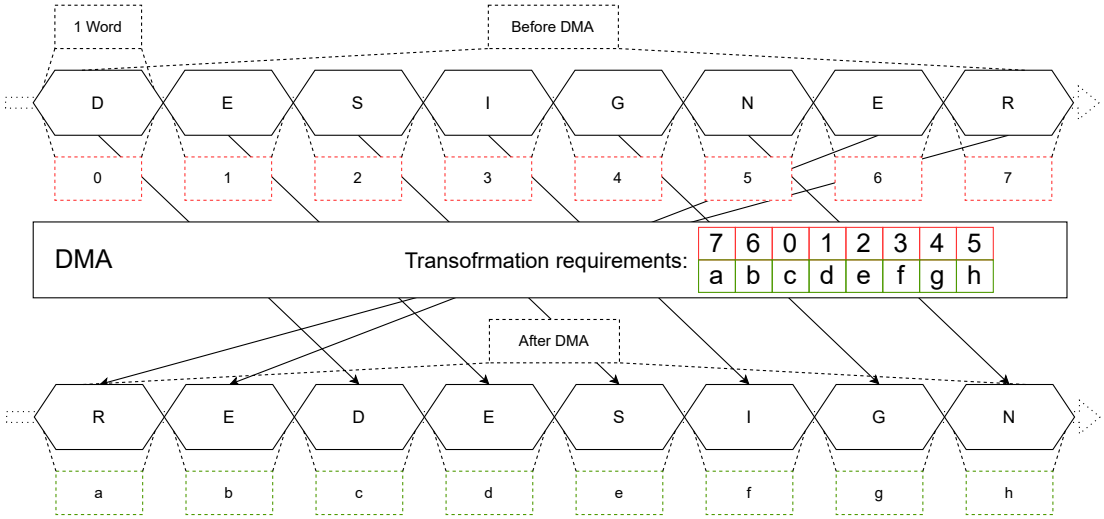

Figure 3.6: The DMA must be able to place data out of order to facilitate data format constraints formed by highly-optimised modules.

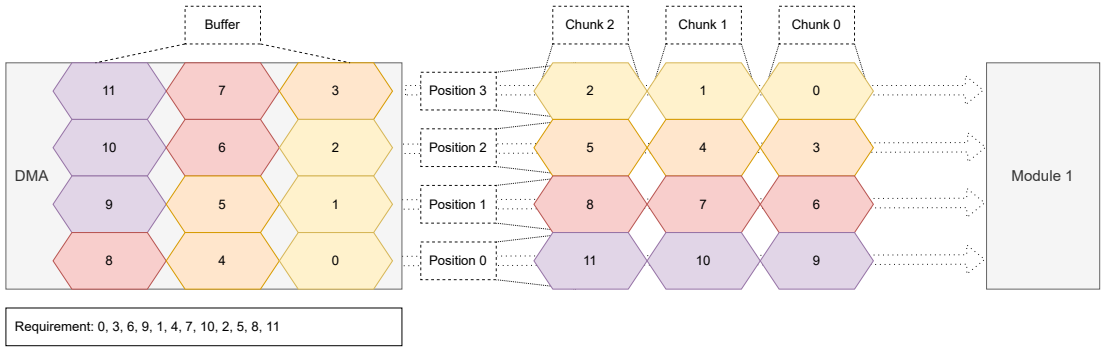

Figure 3.7: The data is packed into different packets arriving at modules each clock cycle. The requirements show how the data must be placed into the packets sequentially, given the original positions in the buffer.

certain amount of credits that symbolise how much additional data they can request from the previous modules to continue operating and getting additional credits. Manev describes full details of this dynamic stream processing interface (DSPI) in his thesis [195].

Incidentally, this credit (token) system also supports prefetching. Once the credits have been sent to request more data, the query processing can begin. We observed in our case study that the DMA module had already prefetched data and started some execution before the entire system was initialised. This early execution is possible due to the self-scheduled flow control (like in a Kahn process network) that is transparent to the runtime system as long as the sequential order of module initialisation follows

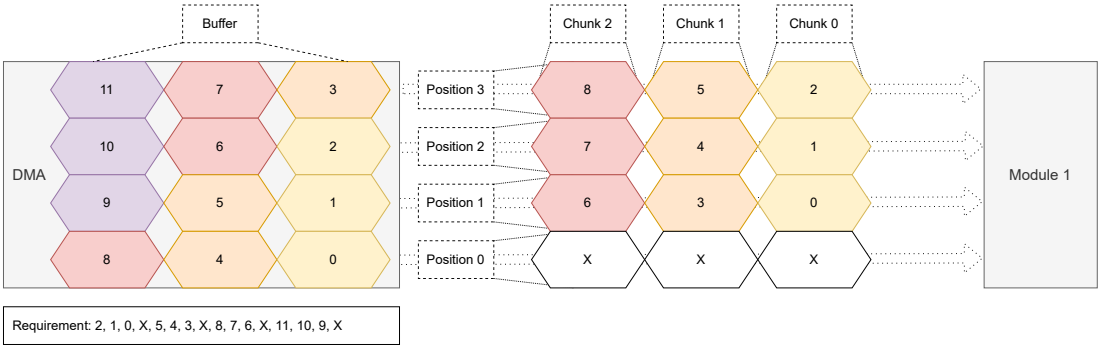

Figure 3.8: The crossbar inside the DMA provides the freedom to set data as required and is most optimal for the modules. This allows us to change the endianness or remove unnecessary data on-the-fly as we do not care about data marked as *X* (more useful when writing data from the datapath to memory).

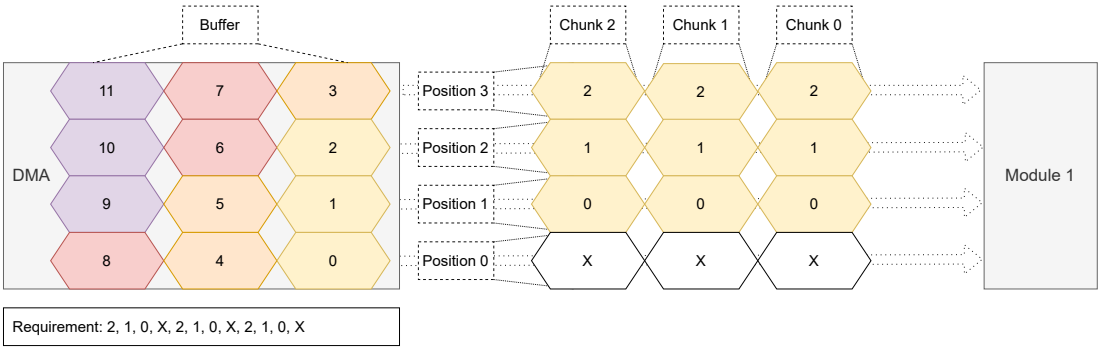

Figure 3.9: We can set the crossbar to keep selecting data from the same data location in its buffers during multiple clock cycles to duplicate any data if necessary.

the dependencies of the pipeline.

The decentralised approach is necessary to enable ad-hoc swapping out any necessary modules with varying flow-altering characteristics. Contrary to the flow control signals that the runtime system does not have to manage, there are initialisation signals that the runtime system uses directly to notify the modules in an optimal order when to start, after initialising the modules with the next operation parameters (to enable data pre-fetching). Given the control and data wires, the system can send instructions and instruction data to the modules to initialise or reprogram them (including the data placement and enumeration inside the DMA). Swapping out modules is expensive, and as many different streams, data formats, and constant operational values are used for each operation, the modules must be flexible to limit the number of required module alternatives.

With programmability, the system provides a significant degree of freedom and is

enabled by the static system that routes memory-mapped data writes to correct modules. The added initialisation latency is compensated by increased throughput over long-running operations. The concrete values written to these modules are determined by their drivers, described in the implementation chapter.

As a culmination of this complex data management, we can see that the runtime system becomes more vital for productivity as it handles the operational details automatically. Furthermore, often these operational details are tightly coupled to the application-specific implementation of the system, which is intentionally virtualised as part of this thesis as we will look at the drivers created for evaluation in the next Chapter separately from the ideas introduced in this Chapter. Besides the operational signals and the initialisation data calculations, the runtime system comprises a complex scheduling problem with first finding the most optimal modules in the first place to use this programmability effectively that we also have to investigate before delving into the implementation.

3.3 Scheduling

Now that we have specified the datapath and flow control, the next step is to specify the scheduling and module placement problem. After describing the problem in general we will then define the general-purpose constraints and objectives.

3.3.1 Problem Description

Scheduling can have a range of objectives, often pursued simultaneously with varying priorities. Common goals aim to either minimise makespan, tardiness, waiting time, turnaround time, response time, context switching, or to maximise factors such as throughput, workload distribution, reliability, adaptability, resource utilisation, energy efficiency, and fairness. All of these are covered extensively in literature.

In this thesis, our focus is on a specific scheduling challenge that involves parallel task scheduling. The primary objective is to minimise the makespan, which is similar to achieving minimal average latency and maximal average throughput. Minimal makespan ensures a swift initiation of the next scheduling spin (phase) and the efficient processing of requests in each data burst. This strategy is designed to address the main memory throughput bottleneck while maintaining adaptability to evolving requirements as new requests may be arriving during the execution of previous requests.

Each batch of requests is scheduled during its respective scheduling spin and this section will describe each spin. In the example scheduler we present in the next chapter, these spins occur after each execution phase. More frequent scheduling spins are beyond the scope of this thesis. Furthermore, we do not currently consider prioritisation (and having multiple scheduling queues) that would align our approach more with traditional CPU and OS schedulers. As a result, we do not consider resource starvation, assuming tasks are, on average, processed faster than their queuing rate, and that the scheduler does not have to decline any tasks that it supports with the given module library.

In order to adapt to changing runtime requirements, we need to use resource elasticity to optimise the resource allocation dynamically. Partitioning the available resources to fine-grain slots allows us to build a resource elastic module library consisting of multiple module bitstreams for each supported operation. As a side note, the more fine-grain the slots, the more flexible the resulting system can be, as is demonstrated by Koch et al., [148] with high-slot count systems. However, the additional flexibility comes with increased complexity, and each additional module induces an additional interface overhead. As is the case with the module library and its drivers, the implementation details of the example scheduler containing all concepts introduced here will be discussed in the next chapter, while this section offers a theoretical overview of the system.

As a reminder, each resource elastic module has multiple bitstreams for two reasons. First, different **variants** provide multiple placement options given heterogenous slots and their combined patterns. Second, different **alternatives** provide the ability for the modules to take as many slots as required to have the necessary **functional capacity** for any given workload demands (e.g., sorting more streams in a single run with modules that take more slots).

Resource elastic stream processing

Resource elasticity excels in stream processing workloads as these require either: 1) batch-processing with blocking operations due to limited buffering space, or 2) partial execution due to limited logic resources. Blocking operations (e.g., sorting) require looking at all input data packets before starting to output any resulting packets - requiring partitioning the blocking operation into smaller subproblems where the results can be later merged for the final aggregated (fully sorted) result. Similarly, other operations

(e.g., filtering) may require a large number of computational steps (like filter comparisons) and, therefore, can be done partially with smaller atomic runs that process only a subset of these required steps when there are not enough resources available. Later, given enough runs (for executing all required comparisons in the requested expression) - the total result is eventually computed.

We can process larger or smaller batches of data in a single run depending on if a larger or smaller number of slots are dedicated to a module. For larger datasets or more complex operations, multiple runs are required where the same dataset may be streamed through the module multiple times, leading to drastically increased runtimes. However, by having the capability to reconfigure and use more runs, the system can guarantee that any query can be executed under resource constraints, given a sufficient pool of HW modules. To differentiate streaming the same dataset multiple times as a whole or with smaller partitions, we use the terms major and minor runs (not directly correlated to size).

Major run: A major run consists of at least one or multiple minor runs where the whole dataset gets streamed through the configured pipeline.

Minor run: A minor run streams a fractional amount of data in a single pass through the modules.

In a hypothetical scenario, the module library includes merge sorting units that can merge only two streams at a time into one sorted stream. Sorting four streams using one of these units would require two major runs: the first major run would consist of two minor runs, and the last major run, which produces the final sorted output, would consist of one minor run. The scheduler's first goal is to minimise the number of major runs, and then second, minimise minor runs to reduce operational overheads (initialisation costs), leading to better end-to-end performance - minimised makespan.

Therefore combinations of multiple physical modules can be used in parallel to act as one larger logical module by programming them all to work on the same stream. Such **module composing** enables processing larger batches of data and reducing the number of required consecutive merging operations (when supported by the module). By composing the two merge sorter units, that can sort only two streams at a time, the new combination of modules sorts four streams at a time, and instead of 2 major runs consisting of 3 minor runs, the whole task is finished with one major consisting of one minor run. Given enough module variants, it is easier to minimise external fragmentation - reduce the number of resources in the PR region not allocated to any module

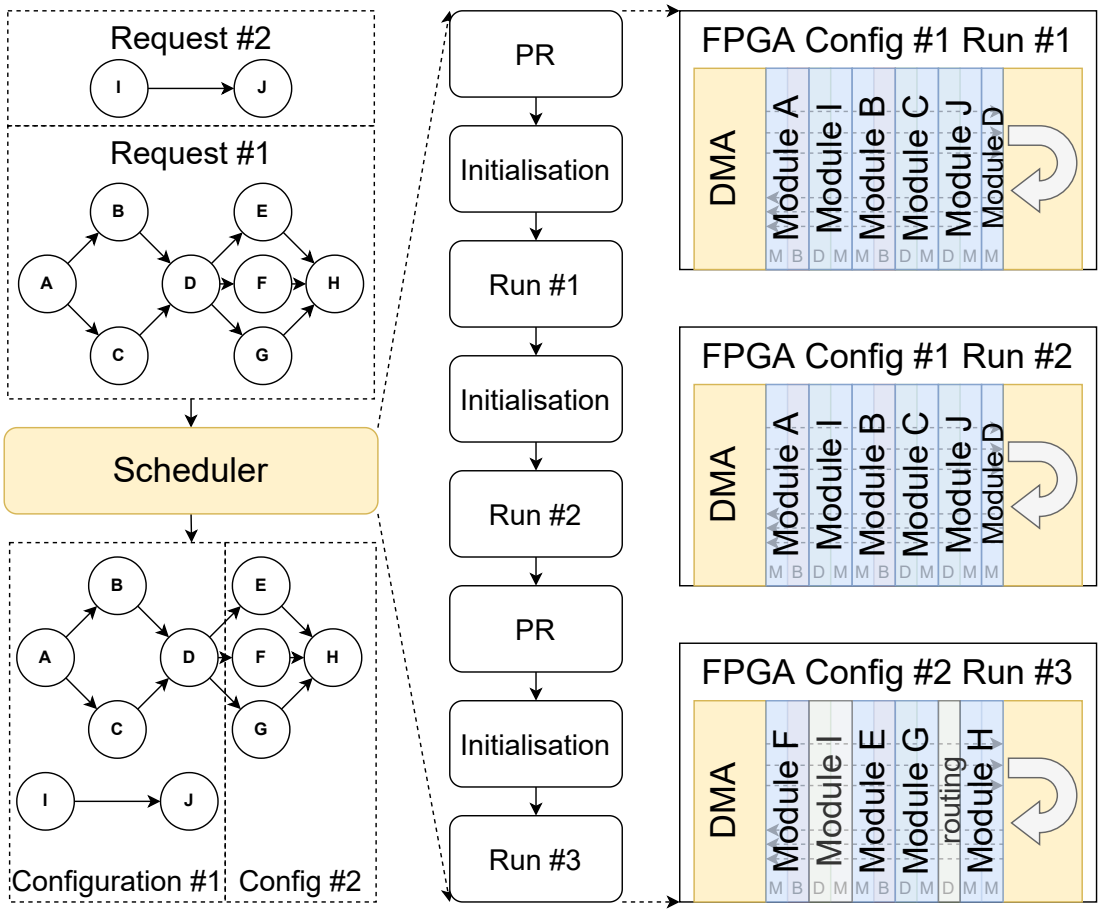

Figure 3.10: The scheduler partitions incoming requests #1 and #2 to multiple configurations and runs while mapping operations to corresponding modules. Consequently, request #2 gets processed in parallel to #1 in the first configuration and the overall makespan gets minimised.

due to not finding a suitable set of modules that collectively cover the whole PR region without any overlaps. Composing smaller modules can also reduce this fragmentation because they can be split up and even separated by placing other modules between them due to all modules working on their designated streams. The scheduler's job is to select the best (or close to) fitting set of modules using *any* combination of different module variants and alternatives while composing them when appropriate.

Optimisation targets

Nevertheless, when the incoming acceleration service requests become complex and large enough (with parallel requests), re-initialisation (reprogramming modules) and partial reconfiguration (swapping modules out), as shown in Figure 3.10, becomes

viable or even necessary despite the steep initial cost. The incoming batch of requests can be partitioned into multiple time-multiplexed runs - allocating more resources to more modules for faster execution. If we denote the number of runs as I, the number of elements (data count) in the data stream D_c, the size of each element D_s, the number of different parallel streams in a run J and the I/O throughput v, then data streaming time T_s is:

$$T_s = \sum_{i=1}^{I} \frac{\sum_{j=1}^{J_i} D_{sij} \times D_{cij}}{v} \tag{3.1}$$

In other words, T_s is equal to the time it takes to stream $D_{cij} \times D_{sij}$ elements through the system at speed v in each run i (up to I runs in total), for each parallel stream j in the run i. In this data streaming time evaluation, we assume that the execution time is bottlenecked by the time it takes to stream input data and that the input and output streaming overlap each other (which is more likely the case for large streams that we target the system for).

Consequently, due to the constant I/O throughput and the modules likely operating on the clock speed equal to the slowest module's speed, the performance of the different resource elastic pipeline configurations is determined by the **utility** of the modules. The utility is defined by the amount of valuable work (data processed) a specific acceleration pipeline delivers per I/O operation. Plans with less utility than required for the given data loads require additional runs through the FPGA, streaming the same data multiple times, while ideally streaming processing needs only one run through the FPGA.

When to use partial reconfiguration?

PR can be used to switch out modules from different runs that can be used to increase the overall utility or to adapt to new acceleration requests. However, the configuration overhead scales with the FPGA resources to be reconfigured. Therefore, the system can reprogram modules to meet new requirements and avoid reconfiguration if the module is reused from the previous run. These optimisation opportunities are more likely to arise when the scheduler has to schedule multiple batched requests simultaneously with repeating operations, such that modules are in the correct location in multiple runs without being swapped out with PR. Nevertheless, balancing placing modules in such locations to enable programmability and minimising fragmentation and maximising utility is a challenging scheduling problem.

It is necessary to remember that configuration speeds, initialisation speeds, I/O

throughput, and module throughputs are different on different devices, yet our proposed ecosystem enabled by this scheduler is device-agnostic. Therefore the scheduler must be given all relevant system parameters beforehand, as we can see two relations: 1) with faster configuration speed, more configuration data can be used to increase utility, 2) with faster I/O throughput speeds, more utility can be sacrificed to reduce configuration times.

Assumptions

Before looking at the scheduling problem we define our assumptions about the initialisation costs and about the modules in our module library.

With more general-purpose module libraries (with choices between the different levels of generality in addition to the resource-elasticity-induced choices) where the initialisation times are significant enough, the initialisation cost also has to be explicitly minimised by the scheduler. However, the initialisation cost is a mandatory cost in each run that is much faster than reconfiguration (even more so when the modules have limited programmability). Therefore, as we do not have general-purpose module alternatives for the scheduler to choose from in our application-specific case studies, the scheduler's ability to consider initialisation costs is out of this thesis's scope.

Furthermore, to simplify formally defining the placement problem, we assume all modules to have the same clock speed. Additionally, they are constrained into rectangular bounding boxes (multiple PR slots) covering the full height of the configurable region. Each PR slot is a thin and tall (to facilitate wide datapaths) resource column that can be characterised by the resources it occupies. Giving each slot a resource character allows interpreting the module placement problem as a string matching problem, as module resource requirements can be represented as a pattern of required resource columns. As the modules are prebuilt - finding correct placements on a particular device can be done offline.

Let us also assume that the modules have input interfaces on the left and output interfaces on the right and that adjacent modules can stream directly through prebuilt routing paths. If the space next to a module is unoccupied, a routing block (or a turnaround block to route the resulting data back towards the DMA) can be placed instead to connect to modules farther away. Streams have identifiers for implementing virtual channels that allows them to bypass modules if needed for implementing complex topologies made up from various communication dependencies. The placement problem thus uses a 1D model where modules are placed in one horizontal plane into a

single configurable regions in a single run. This pragmatic model will be implemented and evaluated in the next chapter as it is flexible enough to support arbitrarily complex problems given the appropriate module library.

Scheduling problem's overview

The rest of this section describes the spatial and temporal dependencies that must be considered for the packing problem that placing modules on an FPGA represents. Any runtime system optimising FPGA resource usage with dataflow operators in a configurable region needs to consider the following main constraints and optimisation objectives:

- Constraints:
  - τ_1: Ensure correct task execution
    - All tasks must be executed correctly in the correct order according to a partial ordering of the tasks.
    - Each data packet (or even a smaller unit given appropriate module support) must be fully processed in the previous task before starting any subsequent tasks.
    - Uncertain data loads from non-deterministic operations (i.e., how much data passes filtering) must be accounted for full execution (which may result in scheduling multiple times or placing enough modules to cover the upper bound of the potential resulting data size).
  - τ_2: Ensure correct accelerator placement
    - Each accelerator module must be placed in a location with a matching resource string.
    - Modules must also be in the correct order to execute the tasks in the correct order.
    - No two modules can overlap.
    - Data formats between two consecutive modules working on the same task or on two tasks with a shared edge must match.
    - Modules must be placed within the correct distance of each other for timing purposes.
- Objectives:

  - φ_1 - Minimize configuration cost.
  - φ_2 - Minimize streaming cost.

As a reminder, in order to make the following subsections more clear that describe this packing problem, we use the following terms with the symbols in Table 3.1:

- **Input service request** - A request of some computational service on the provided data. Requests can contain multiple orthogonal and concurrent (batched) subrequests - representing a DAG with multiple strongly connected components (SCCs).
- **Datastream** - A stream of data flowing through the system consisting of multiple data packets. Multiple datastreams can be used in the same request.
- **Workload** - The *amount* of work required to perform an abstract computational job, such as counting or sorting. This amount is measured in terms of the number of individual microtasks, which could involve comparing the fields of n data elements, moving m data elements, or performing other operations.
- **Task** - An atomic job *type* (to be executed on a datastream) with workload values that define the needed compute capacity. In other words, a task consists of a single or multiple microtasks of given workload sizes.
- **Operation** - An input request can be defined with multiple high-level operations (such as sorting) that can be translated to a combination of tasks (linear sorting and merge sorting phases) on each datastream. In other words, a high-level operation could consist of multiple low-level tasks.
- **Resource column** - A single character in a *resource string* mapping a certain resource type (e.g. a "D" stands for a column of DSPs with the associated routing).
- **Module** - A module is a physically implemented hardware accelerator (part of a larger design) providing processing or storage capabilities. The resource columns used form the module's resource string.
- **Configurable region** - A container where different modules can be placed simultaneously (called PR region in a system using partial reconfiguration). The resources of a reconfigurable region are modelled as a string of resource columns.

Table 3.1: In order to formulate the scheduling problem, we define the constraint and objective functions with the elements introduced here.

Symbol	Meaning in the here presented scheduling problem context
T	The current set of tasks to be accelerated
t	A particular task which is built from a vector of workloads sizes
Ω	The current vector of PR modules executing the current set of tasks
W	A vector of all possible workload types
w	A particular workload size of a task
M	A set of modules representing the current module library
m	A particular module
E	A vector of tuples for a particular module that define its functionality and capabilities
e	A particular effect function tuple a module manipulates a particular workload with
v_e	An effect value of a module on a particular workload, corresponding to its capacity
f_e	A workload modifying function given the original workload value and the applied effect
$\oplus$	A relation showing how a set of modules update all requested workloads
P	A vector of priority values for each task
p	A particular priority value for a task
l	A location tuple defining where a specific module is physically located
l_s	A starting point index value for defining where a dedicated module area begins
l_e	An ending point index value for defining where a dedicated module area ends
Φ	An overlap verification function that returns the distance between any two modules
f_p	An order verification function for checking the correct execution order of two tasks
C	A set of module adjacency constraint verification tuples
c	A particularly indexed constraint tuple describing module placement constraints
f_c	A function for finding out an adjacency constraint verification value
v_c	A particular adjacency constraint verification value calculation constant
R	A vector of reconfiguration costs of all modules in the module library
r	A particular reconfiguration cost associated with a module in the module library
U	A set of reconfiguration costs of all modules used in the current run
u	A particular reconfiguration cost associated with a module in the current run
S	A set of slots that can be used to calculate the routing configuration cost
D	A set of data streams required to execute a specified task
d	An index to a datastream whose size gets updated according to the modules

3.3.2 Motivating Example

Before formally defining the scheduling problem, we will present an example showcasing why this is a challenging scheduling problem that also requires runtime monitoring.

Table 3.2: The merge sort's functional capacity notes how many sequences it can merge into one. For each module it is listed how many CLB (M), DSP (D) and BRAM (B) resources it needs.

Module	Functional capacity	Resource footprint
Merge sort	32	MBDMDMM
Merge sort	64	BDMMBDMDMM
Merge sort	128	BDMMBDMDMMBD

As mentioned before, module composing increases utility and reduces *external* fragmentation (i.e., fitting modules tightly together). However, larger monolithic modules have reduced *internal* fragmentation (e.g., maximising the use of available BRAMs inside a module bounding box for sorting) due to tighter integration and better scaling as they provide more utility per resource. To minimise reconfiguration costs, we must examine the tradeoffs between using larger modules or multiple smaller ones. This optimisation involves studying the relationship between the use of more resources and the resulting increase in utility.

This mostly device-agnostic example (except for the module resource requirements from our implementations) examines the utility-to-configuration cost ratio of one data size-sensitive resource elastic module - the merge sorter. Table 3.2 shows the resource footprints of the sorting modules (adapted from [191]) in our HW library. The number of major runs of an E-way merge sorter to process X sequences is $M = \lceil log_E X \rceil$. Consequently, the number of minor runs required is:

$$m = \sum_{i=1}^{M} \left\lceil \frac{X}{E^i} \right\rceil \qquad or \qquad m_{min} = \left\lceil \frac{X-1}{E-1} \right\rceil \tag{3.2}$$

m_{min} is the minimum number of minor runs if using an overlapping merger strategy which fits better smaller datasets; however, it leads to a larger major run count in the general case. The number of minor and major runs is crucial for cost modelling and measuring utility for all resource elastic modules (e.g., filter modules).

Figure 3.11 shows examples where different sorter modules had been composed for delivering a larger functional sorting capacity (the number of sequences merged per single minor run). Figure 3.11 also shows a *nonlinear relationship between resources and utility* in the merge sort example. More resources allow us to cut down the number

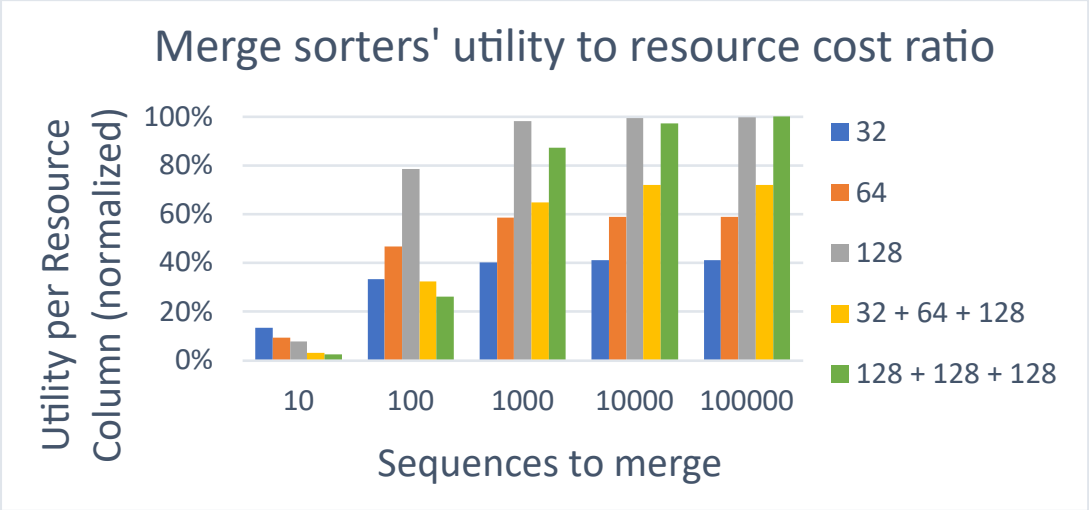

Figure 3.11: Different modules are suitable for different scenarios, depending on the data-dependent ratio of utility to resource cost. Module composing gives more freedom in module placement and is more suited for increasing capacity despite larger resource costs, while larger monolithic modules fit more resource-constrained scenarios.

of major runs (e.g., in two major runs with the same I/O cost, a 32-way sorter could merge 1024 streams while a 32+64+128=224-way sorter could merge 50K streams respectively).

However, in some applications, the sorting problem size is data-dependent. For instance, we likely have filtering operations before sorting operations in database acceleration problems - making the sorting problem size unknown beforehand. There is no overall best strategy, so such a system requires runtime monitoring with a dynamic scheduler that can adapt to find the best approach for each particular workload.

3.3.3 Constraints

In order to give a formal definition of this scheduling problem, we can start from a *basic placement problem definition* as explained by Stoyan et al., [289] where we have to place a set of tasks T from the input service request DAG into a configurable region (container) building up a vector of modules Ω while optimising based on our objective functions. However, before looking at the objective functions, we must understand how to define the current configuration Ω and what constraints must be met to make it valid after defining tasks and modules.

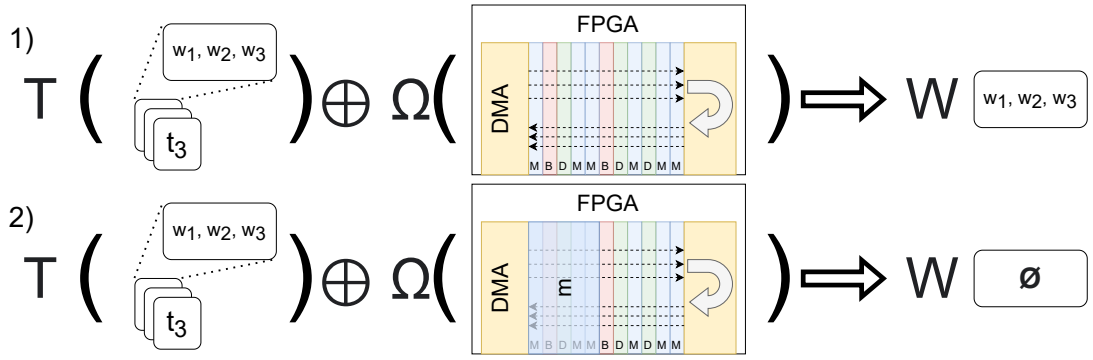

Figure 3.12: At first, we have a request consisting of tasks T to be accelerated with an empty FPGA, resulting in the required workload equalling the sum of workloads defined in T. After configuring a module m that executes the jobs defined by the tasks in T, we have no remaining requirements.

Task and module definition

First, we look at tasks in T, representing a DAG with nodes and edges. Tasks are generated to serve a set of operations, where all workloads of a specific type (enumerated until i) from the tasks in T are in the set W. That is, for each $t \in T, t = (w_0, w_1, ..., w_i)$ and t has values for all $w_i \in W : i = \mathbb{N}$ where each w marks how much work (in terms of problem size) has to be performed to execute t. For example, a simple counting task has only the value $w_j > 0$ (here: for the counting task, equal to one as we cannot know how many data elements we want to count) where j is indexing the counting task while all other w values are equal to 0.

Then second, we have to look at modules in our module library M. All tasks can be fulfilled by different vectors of modules Ω scheduled to the configurable region, which for now, we can assume is infinitely long. In addition to placement values and constraints we will look at later this section, each physical module $m \in M$ contains a set of effect pairs $E = \{e_0, e_1, ..., e_i\}$, where $e_i \in E = (f_e, v_e)$ given each i indexes both the workload type w and necessary compute capacity in m. In the effect pairs model, the value v_e updates the workload value (how much compute capacity is required after being processed by the module m) with the corresponding paired function f_e such that $w_i' \leftarrow f_e(w_i, v_e)$. For example, modules can have a subtractive/additive effect on the workload value (due to limited buffer sizes that necessitate more runs), while other modules just set the value to 0 regardless of what workload value was set before for the executed task (i.e., non-blocking, 1-in 1-out modules), for which f_e information is required. In other words, modules may only execute tasks up to a certain problem size, defined by v_e (e.g. an insertion sorter module).

In order to define how each module works on a task, we can see each module with its effect tuples set as a function modifying the task it is working on in the DAG. Figuratively, $t' \leftarrow m(t)$ means that when a module executes a task, it adds a new task to the input DAG on the edges between this node and the subsequent nodes. For a task to be fully executed, all microtasks must be completed by an arbitrary number of modules $m_0 \circ m_1 \circ ... \circ m_j(t) = (0,...,0)$, such that all workloads are equal to 0. A naive assumption from the scheduler would be to make this stack as short as possible to find the fastest execution time. In certain scenarios, a longer chain of modules may have lower configuration costs due to module reuse and can allow allocating more modules for other operations, resulting in faster execution times. However, this requires context that will get defined in the following subsections for mathematically understanding the scheduler's objective of optimising module selection for all tasks in T.

Hence we can represent the scheduled modules Ω update the workloads of tasks in T with the symbol $\Omega \oplus T$ (Figure 3.12), or in other words: define Ω as a vector of module and task tuples $\omega = (m,t)$ such that $\Omega = ((m_0,t_0),(m_1,t_0),(m_1,t_1))$, for example. The whole set of tasks will finish processing if the sum of the workload values of the modules is larger than the sum of workload values of the tasks as shown in Equations 3.3 and 3.4. Intuitively, if there is a corresponding module set placed in the configurable region for all tasks, then the total workload of all different work types must be equal to 0 for the placement to be satisfying.

$$\forall t_i \in T \exists M_{ti} \in \Omega : M_{ti} = (\forall m(\omega) : \omega \in \Omega \wedge t(\omega) = t_i) \wedge M_{ti}(t_i) = (0,...,0) \tag{3.3}$$

$$W' = (0,...,0) = \varnothing \Leftrightarrow W' = \Omega \oplus T \tag{3.4}$$

$W = \varnothing$ is true when the $\sum_{j=0}^{J_m} w_j = 0$. We define this first primary constraint function in Equation 3.5 abstracting the constraint in Equation 3.4 which means that *given a set of tasks and a set of placed modules, all workloads must be executed.*

$$\tau_1(T,\Omega) = 0 \implies W = \varnothing \tag{3.5}$$

Partial order of tasks

Now the modules and the tasks in Ω must be arranged in the correct order. Execution dependencies of tasks can define a partial order for the modules executing the workloads of these tasks. However, not all tasks are related; therefore, this provides some

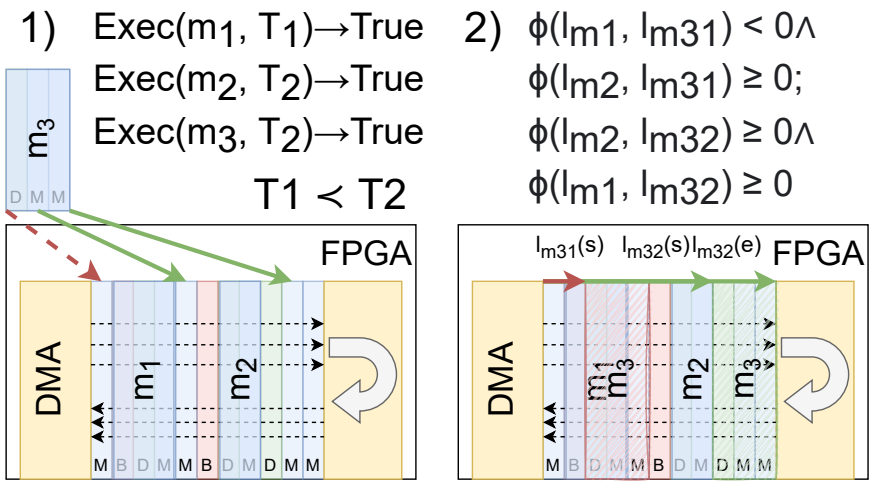

Figure 3.13: In 1), we see how a task order defines possible modules placement positions in relation to other modules. 2) shows how modules are prevented from overlapping. We also see how the module placement data l is defined.

freedom in the scheduling (there is some mobility between the earliest possible module placement slot and the latest possible slot) that can be used for optimisation. We can create a vector of priority values $P_t = (p_1, p_2, ..., p_n)$ for each task t. Then we need to order the tasks $t_i, i \in \{1,2,...,n\} = I_n$ in the execution order for each priority value $p_{tj}, j \in \{1,2,...,n\} = J_m$ in each t_i. In other words, to order tasks in space (and in time), they have to obey the partial order constraint such that all priority values for a task in position i are smaller than all corresponding priority values for tasks that are executed later - formulated in the task ordering constraint in Equation 3.6.

$$\forall t \in T \wedge \forall p \in P_t \wedge j \in J_m \wedge i \in I_n \mid t_i \prec t_{i+1} \Rightarrow p_{ji} \leq p_{ji+1} \tag{3.6}$$

The same partial ordering constraint must also be set on the modules processing the tasks themselves (Figure 3.13.1). However, next, we also need to consider how modules get placed physically next to each other in a configurable region Ω. As such, each module m in Ω is also paired with location information tuple $l(l_s, l_e)$, which are the starting and end position x coordinate values equal to the distance from the data source (l value arrows in Figure 3.13.2). With the location parameters, we can make sure all modules placed in the configurable region obey the task order. This results in the physical module ordering constraint in Equation 3.7.

$$\forall m_i \in M_{ti} \mid t_i \prec t_{i+1} \Rightarrow l_i \leq l_{i+1} \tag{3.7}$$

Intuitively this partial ordering means that a module can start executing a task if all of the preceding tasks are finished, and that it can only be placed after the modules that finish processing the prerequisite tasks (in space but also in time if multiple runs are required). Prerequisite tasks can be completed partially in stream processing pipelines, and therefore data packets must be processed fully by the previous task.

Nevertheless, as a task can be executed with multiple modules, the modules of two consecutive tasks cannot interleave. This constraint can be defined with the following that must also be incorporated into τ_1 (Equation 3.5) by returning a negative number with incorrect ordering:

$$\begin{aligned} &\forall \omega, \widetilde{\omega} : \omega < \widetilde{\omega} \\ \wedge &EDGE(t(\omega), t(\widetilde{\omega})) = 1 \\ \wedge \neg &\exists \omega' > \omega : t(\omega') = t(\omega) \\ \wedge \neg &\exists \omega'' < \widetilde{\omega} : t(\omega'') = t(\widetilde{\omega}) \end{aligned} \tag{3.8}$$

Placement of modules to a container with no overlaps

Next, the physical module placement has to be computed. The first constraint here is where each module's location's resource string must match the module's own resource string (Equation 3.9). However, overlapping and the partial ordering of the modules constraints also have to get defined.

$$\forall \omega \in \Omega : RESOURCES(l(\omega)) = RESOURCES(m(\omega)) \tag{3.9}$$

We describe the overlapping of two modules A and B with their position and size values (in our case encoded in the $l(l_s, l_e)$ tuple) with the *phi*-function $\Phi(l_A, l_B)$ that returns a value 0 or larger if the objects A and B do not overlap (as used in [289]). The overlap function could be extended by composing with another function f_p that validates any two sets of priorities P_t for task t such that the corresponding two modules m_A and m_B in Ω work on tasks noted as t_A and t_B in the correct order. We define this f_p in Equation 3.10 to abstract the constraints in Equations 3.6 and 3.7.

$$f_p(A,B) = \mathbb{N} : f_p(A,B) \geq 0 \Rightarrow A \prec B \tag{3.10}$$

Consequently, the composition of precedence function f_p and Φ from [289] can express the validity of module placements such that $f_p(A,B) \circ \Phi(A,B) \geq 0$ is true if the partial order of the tasks is obeyed and the modules do not physically overlap. We

define this second primary constraint function as a composition of Equation 3.10 with the Φ function and the valid module positioning constraint in Equation 3.9 for defining *valid module placements* in Equation 3.11.

$$\tau_2(T,\Omega) \geq 0 \tag{3.11}$$

Multi PR region module placement

Now we remove the assumption that we have unlimited resources as we need to find a way to fit all modules into a limited configurable region to successfully process all of the workloads of the desired tasks. This requirement means that with all pairs of modules m_A and m_B in a possible set of modules M, we can define a potential problem in Equation 3.12 if no combination of modules can meet all of the constraints given finite configuration region boundaries.

$$\neg\exists M \in \Omega \wedge \forall m_A, m_B \in M . m_A \neq m_B \wedge f_p(m_A, m_B) \geq 0 \wedge \Phi(m_A, m_B) \geq 0 \wedge M \oplus T = \varnothing \tag{3.12}$$

This problem implies that for any set of modules in M to execute all tasks in set T, the set of modules Ω must consist of modules in multiple configurable regions as illustrated with Equation 3.13. When combining the available resources of systems consisting of multiple regions, the individual resource strings can be joined with region breaker symbols as explained by Grigore et al., [92].

$$\Omega \oplus T = \varnothing \Rightarrow \Omega = (\Omega_1, \Omega_2, ..., \Omega_n) \tag{3.13}$$

Assuming all regions used have the same resources and hence length noted with x_{max} we can split the infinite region configuration Ω into sub configurations (for different regions the length value must be made into a vector based on the regions ordering). However, with splitting Ω, we must ensure that no module is placed in the middle of a split (configuring half a module in a run is not permitted) either by avoiding region breaker symbols in resource strings or with the Equation 3.14 that must also be incorporated into τ_2.

$$\forall \omega \in \Omega : INT(x/x_{max}) = INT((x + SIZE(l(\omega))/x_{max}) \tag{3.14}$$

The additional regions can be either a different subsequent configurable region, different FPGA, or the same region but in a different time slot through PR (becoming a

PR region). Chaining of modules in different PR regions can be modelled with resource strings using wildcarding, as shown by Grigore et al., in [92]. The regions and all of the modules in the regions are also ordered to obey the tasks' and modules' partial order - meaning we need to extend the partial ordering constraint with Equation 3.15.

$$\forall m_i \in \Omega_i \wedge \forall m_j \in \Omega_j : i < j \Rightarrow f_p(m_i, m_j) \geq 0 \tag{3.15}$$

Module specific placement constraints

So far we know that constraints can be expressed with the partial order defining priority values P over all of the PR regions in the set Ω as shown with Equation 3.15. However, individual modules have additional placement constraints. These module adjacency constraints can be summarized as follows:

- Modules may *prohibit* or *require* placing other certain modules directly (or within some distance) before or after.
- Modules may *prohibit* or *require* their placement in a certain position (i.e., first or last) in terms of partial ordering precedence values in the PR region Ω_i.

To model the additional constraints between different modules, we introduce another set of values $C = \{c_1, c_2, ..., c_n\}$ for each m. As each module consists of a set of effect pairs defined in E and its position values with l as mentioned before, we can add an additional set C into the construct where each $c_i \in C = (f_c, v_c)$ (not to be mistaken with the functions and values in E used in Equation 3.4). These values eventually are summed up after being evaluated with their paired functions to ensure that all module-specific constraints are in balance after their placement, as defined in Equation 3.16.

$$\forall m_i \in \Omega_i \wedge \forall c_i \in C_{mi} : C_i = \sum f_c(v_c) \Leftrightarrow C_i = 0 \tag{3.16}$$

In order to define these inter-module placement constraints in more detail, first, for constraining placing modules within a certain distance from each other, we can redefine the non-overlapping constraint to include a required parameter ε as shown in the Equation 3.17.

$$\begin{aligned} &\forall \omega_i, \omega_j \in \Omega : i < j \\ \wedge &\varepsilon_{ij} + START(l(\omega_i)) + SIZE(l(\omega_i)) \leq START(l(\omega_j)) \end{aligned} \tag{3.17}$$

Second, to constrain placing modules into a certain relative position form other modules or within a sub configuration requires considering the input and output datastream formats. Matching datastream formats can be defined with d:

$$d_{ij} = \begin{cases} 0, & \text{if } m_i \text{ cannot be followed by } m_j \\ 1, & \text{otherwise} \end{cases} \tag{3.18}$$

Using d allows us to constrain two consecutive modules to have the same data formats (given that they share a datastream instead of just passing data through). For two modules working on different tasks with an edge between them, the constraint $d = 1$ can be added to Equation 3.8. However, this is also true for two modules working on the same task where we have to define a separate constraint with Equation 3.19.

$$\begin{aligned} \forall \omega, \widetilde{\omega} \in \Omega : \omega < \widetilde{\omega} \wedge t(\omega) = t(\widetilde{\omega}) \\ \wedge \neg \exists \omega' \in \Omega : \omega < \omega' \wedge \omega' < \widetilde{\omega} \wedge t(\omega') = t(\omega) \\ \wedge d_{m(\omega), m(\widetilde{\omega})} = 1 \end{aligned} \tag{3.19}$$

Module capacity and tasks with unknown workloads

Lastly, we look at how the modules can change the task workload value with the functions and values in E_i for each $m_i \in M$. In the case where any particular workload value is empty (equal to 0 or empty set), the modules have to be able to generate a default value. The default value can be one of the following:

1. A positive integer noting further work that other modules have to do.
2. A 0 value noting that the workload of that specific type has been exhausted
3. A negative integer possibly noting that using this module breaks the flow and does not belong to a valid set of M for the set of tasks T.

Modules can have various combinations of these effect pairings. For example, basic modules can have a single effect on the set of all required workload types (e.g., a simple counter counts the data elements and sets the workload to 0). Meanwhile, more complex combined modules can process multiple tasks with multiple workload types at once (e.g., a composite module that sorts and counts simultaneously sets multiple workloads to 0 regardless of whether the end-user request requires them or not).

How the remaining workload gets affected can be data-dependent for specific work types. This data-dependent behaviour can add an unknown workload x to the workload

while updating the workload with the corresponding effect function f_e of the executing module m. This unknown workload is data specific and is formulated in Equation 3.20

$$\forall e_i(f_e, v_e) \in E_i, E_i \subset M_i : w_i \in t_i \wedge w_i = f_e(w_i, v_e) + x_i \tag{3.20}$$

However, the unknown workload x can be bound such that it can still be overcome by preemptively scheduling (during the scheduling stage before execution) more modules to handle the unknown workloads. The scheduler will know the size of the unknown workload (e.g. after a filtering operation) once the module m has finished executing with the rest of the modules in Ω. It is important to note that, given the option to employ overprovisioning to cover these upper bounds, preemption during execution would allow adapting the pipeline accordingly after scheduling, but these features must be supported by the modules - which is missing in our case and hence not supported. Therefore the scheduler will adapt to the new workload sizes while scheduling the next run in the following scheduling spin (as examined in the next subsection).

Constraints – in conclusion

The module-specific placement constraints (Equation 3.16) and module capacity constraint definitions (Equation 3.20) give more freedom to use various composed (requiring multiple modules to execute a task) and resource elastic modules (providing resource and performance or functionality tradeoffs). These two constraints are included respectively in the two main constraint functions τ_1 and τ_2, which allow building dynamically reconfigurable dataflows in a way that has not been considered before (while considering multiple PR regions as shown in Equation 3.15)

3.3.4 Objectives

With an understanding of the placement problem and defined constraints, the scheduler must optimise the objective function. Intuitively, the fastest scenario would involve fitting all modules into a single run and fully executing the requested tasks without repeated I/O operations. However, the objective is to execute the tasks as fast as possible, which may entail reconfiguration steps, even if it is avoidable (e.g., if smaller modules shared over multiple runs are faster than larger monolithic modules).

Configuration cost

Each module in the module library has a specific cost (the time it takes to configure the FPGA) for placing it. Therefore we model each $m \in M$ to have a specific reconfiguration cost r. These costs are collected to a list $R = (r_1, r_2, ..., r_n)$ of length n - equal to the size of the module library M.Next, we can have a copy list $U = (u_1, u_2, ..., u_n)$ for new modules configured in a particular run where each value is initially zeroed but then after scheduling each module $m_i \in \Omega$, the value $u_i \in U$ is set to be equal to $r_i \in R$ as is shown in Equation 3.21.

$$\forall i \in I_n . u_i \in U = r_i \in R \Leftrightarrow m_i \in \Omega \tag{3.21}$$

After placing all modules into the PR region Ω, we can sum all values in U together to get a configuration cost function defined in Equation 3.22 with a set M to process tasks T.

$$\varphi_1(\Omega(R, M)) = \sum_{i=0}^{n} u_i \tag{3.22}$$

One of the objective functions to optimise is to find the PR region configuration(s) Ω_{min} with a set of placed modules with minimal configuration cost. Ω_{min} is from the set of all possible and *valid* PR region configurations Ω_{all} that fulfil the required workloads defined by the set of tasks T, as shown in Equation 3.23.

$$\forall \Omega \in \Omega_{all} : \begin{cases} \varphi_1(\Omega) \geq \varphi_1(\Omega_{min}) \\ \tau_1(T, \Omega) = 0 \\ \tau_2(T, \Omega) \geq 0 \end{cases} \tag{3.23}$$

Reusing modules

We also need to mind the unoccupied slots within the PR region. These empty slots still must have routing wires to keep the physical circuit complete at all times. If no operator modules fit the unused slots, then special routing bitstreams (containing only the datapath wiring, contrary to using modules in bypass mode) can be used instead with a reconfiguration cost marked in R.

However, operator modules can also be reused for routing data and repeating workload types. First, in a scenario where the PR region Ω_i already has a set of modules M_i configured, each module $m \in M_i$ can be used for routing because of the requirement (described at the beginning Section 3.3) to pass data streams through that are

not flagged for the module m. Second, physically identical modules in M_i can be set up differently with initialisation data written to memory-mapped registers to configure the set of effects E and the datastream it is addressed to process. Therefore if $\Omega_i \neq \varnothing$, then U is not always equal to all of the modules used in the PR region as not all points covered by modules in Ω_i have to be configured due to reused modules from previous runs. The previous configuration and the next planned configuration of modules will then be combined as shown in Figure 3.14.

We find the set of slots that need to be configured with routing bitstreams $S_{routing}$, and the set of modules that must be reconfigured onto the board Ω_{new} given the set of modules in Ω_i already and the set of modules we want in Ω_{want} with the following algorithm:

Algorithm 1 Finding set of slots $S_{routing}$ and modules Ω_{new} in Ω to configure

Input: Ω_i, Ω_{want}

Output: $\Omega_{new}, S_{routing}$

1: $\Omega_{common} \leftarrow \Omega_i \cap \Omega_{want}$

2: $\Omega_{routing} \leftarrow \{\forall m \in \Omega_i \wedge \forall m_{want} \in \Omega_{want} : f_c(m, m_{want}) = 0 \wedge \Phi(m, m_{want}) \geq 0\}$

3: $\Omega_{reuse} \leftarrow \Omega_{routing} \cup \Omega_{common}$

4: $\Omega_{new} \leftarrow \Omega_{want} \setminus \Omega_{reuse}$

5: $S_{reuse} \leftarrow \bigcup \forall m(l(s,e)) \in \Omega_{reuse}\{s...e\}$

6: $S_i \leftarrow \bigcup \forall m(l(s,e)) \in \Omega_i\{s...e\}$

7: $S_{want} \leftarrow \bigcup \forall m(l(s,e)) \in \Omega_{want}\{s...e\}$

8: $S_{remove} \leftarrow S_i \setminus S_{reuse}$

9: $S_{routing} \leftarrow S_{remove} \setminus S_{want}$

In other words, first, we have to find reusable modules, after which we can determine the slots which need to be configured with new operator and routing modules to get the desired set of modules M configured to Ω_{i+1}. Then U can be built from $S_{routing} \cup \Omega_{new}$ rather than from the modules in Ω_{want}. Therefore we have to extend the configuration cost objective function in Equation 3.22 to take the previous configuration of the PR region also into consideration as shown in Equation 3.24. Then these costs are summed together for all runs in a plan to compare with other valid plans.

$$\varphi_1(\Omega(R, M_{old}, M_{new})) \tag{3.24}$$

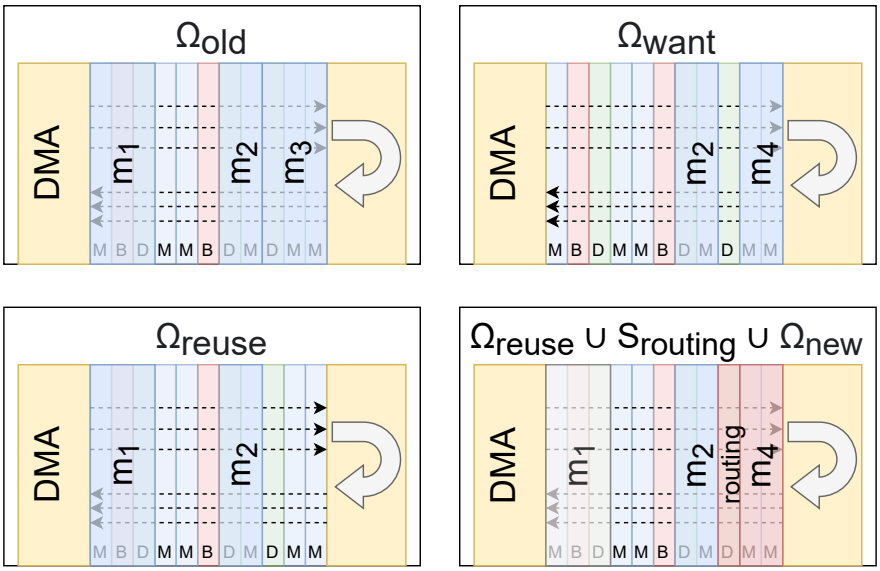

Figure 3.14: By first finding which modules can be reused for execution or routing, we only have to reconfigure columns needed for m_4 and the additional routing column for clearing resources from m_3.

Additionally, to cut down on the cost of writing routing bitstreams, we can use a single turnaround module instead of using multiple consecutive trailing slots on routing. Instead of adding multiple routing bitstreams or bypass modules to fill the PRR region, a smaller bitstream can be used that routes the datapath back towards the DMA, shrinking the used configuration region (albeit when overwriting this U-turn wiring in subsequent runs the full length of the PRR region must be connected again or cut with a U-turn in a different location). This optimisation requires finding the slot with the largest index S_{last} covered by an operational module in Ω_{want}. Then with S_{last}, we can reduce $S_{routing}$ by replacing all regular routing bitstreams with a larger index with a single turnaround routing block. However, this opportunity raises rarely with large enough module libraries that minimise external fragmentation.

Streaming cost

The other optimisation objective to model is to minimise the time it takes to process all the data necessary to execute the desired tasks. As defined earlier each task $t_i \in T$ is a vector of workloads $(w_0, w_1, ..., w_n)$ such that $\forall w_i \in t_i : w_i \in W$. Workload w of a workload type in W is an abstract value and does not necessarily represent the amount of data required to finish the task. Moreover, the same data stream can be used to complete the processing of multiple workload types. Meanwhile, multiple data sources

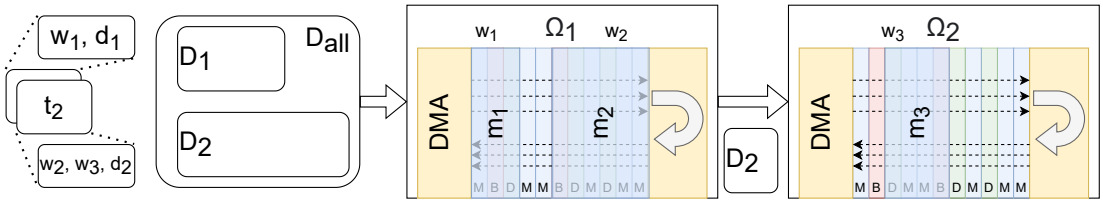

Figure 3.15: Example how different modules are placed into two PR regions define which (and how large) datastreams are streamed to them.

with different formats can also be required to finish a single task. As such, we need to model different data amounts in the set D_{all} as required for all tasks in T where for each $t_i = (W_i, D_i)$ such that $D_i = (d_1, d_2, ..., d_n)$. $d_i \in D_i$ (one of the required datastreams for task t_i) is an index pointing to a value in D_{all} that gets updated after being streamed through the modules M_i in PR region Ω_j (Figure 3.15).

Let the time it takes to read, modify (with the modules in Ω), and write the data in D_{all} be classified as the runtime $\varphi_2(D_{all}, \Omega)$. Additionally, it takes time for a module $m \in \Omega$ to process data. Moving data from one position in Ω to another also takes time. We can model the time that it takes to move the data through the PR region(s) Ω as $\varphi_{execution}(D_{all}, \Omega)$. However, in practice, it takes substantially more time to stream data to the FPGA in the first place. The time it takes to stream data to the PR region(s) is magnitudes larger than execution time given that already the physical distance between DDR and the FPGA is larger than the width of the FPGA and as such $\varphi_{io}(D_{all}, \Omega) \gg \varphi_{execution}(D_{all}, \Omega)$. Given that the latency $\varphi_{execution}(D_{all}, \Omega)$ is so small for I/O bound processes, then we can assume that the process of streaming data to the FPGA and streaming data from the FPGA is mostly overlapping and equal in magnitude on average. For compute bound problems the latency of the operators must be considered as well. However, for the modules described in the next section, the execution latency is negligible, and thus the scheduler aims to optimise I/O costs, assuming $\varphi_2(D_{all}, \Omega) \approx \varphi_{io}(D_{all}, \Omega)$.

To calculate the I/O time, we need the data amounts D_i, that are streamed into each PR region Ω_i in all of the runs Ω with the PR region count being equal to I_n as $\varphi_{io}(D_{all}, \Omega) = \sum_{i=0}^{I_n} D_i$. The modules working on the data sources in each container define the data cost for that particular stream, as all of the data does not necessarily need to be streamed to every module. Consequently, the second primary objective function is to find the set of PR regions Ω_{min} with the shortest streaming time with

valid configurations, as shown in Equation 3.25.

$$\forall \Omega \in \Omega_{all} : \begin{cases} \varphi_2(D_{all}, \Omega) \geq \varphi_2(D_{all}, \Omega_{min}) \\ \tau_1(T, \Omega) = 0 \\ \tau_2(T, \Omega) \geq 0 \end{cases} \tag{3.25}$$

3.3.5 Partial Scheduling vs Full Scheduling

When the modules produce resulting workloads with unknown amounts, there can be an option to schedule more modules such that the module *compute capacities* are larger than the unknown (but bounded) workloads ($M_W > \sum x_i, \forall x_i \in W$) for full scheduling. There are cases when this is not possible, and then by ignoring the task fulfilment constraint τ_1, partial scheduling can be used as long as it is possible to finish processing with subsequent module scheduling plans. In a system, where the scheduling process and FPGA execution processes are parallel to additional input query parsing processes, new independent tasks may be added ad-hoc to the set of tasks T that also must be scheduled in the next scheduling spins, making partial scheduling more suitable.

Alternatively, there is a third option of using heuristics, such as scheduling no additional modules with unknown workloads unless the worst-case scenario amount of unknown workloads can be executed by additional modules that fit in the PR region while using leftover resources - resources that are still available after scheduling everything else not involved with the unknown workloads.

3.4 Chapter Conclusion

This chapter introduced the parts required for runtime management for partially reconfigurable pipelines on FPGAs. The following aspects have to be done automatically: 1) memory management, 2) data management, 3) scheduling, 4) input parsing, and 5) execution management.

First, for the system to be flexible enough to work with any memory subsystems and dataflow pipelines, we need a swappable data mover module. This data mover module and the modules themselves must adhere to a standard interface where modules communicate through a token system (indicating available buffer space) to avoid deadlocks in an acyclic execution sequence.

Second, the middleware parsing service requests and building the most optimal

accelerator setups also handles execution management as the modules in the module library provide enough flexibility to require scheduling data packets to be in a particular layout on the datapath. This way, the modules can be optimised to expect certain words of data in a subset of all possible offsets while being flexible enough to be programmed for multiple closely related problems. As a result, we have a system capable of adapting to various workloads with various set of module libraries - a necessary step to build a larger ecosystem for code-sharing within the FPGA community.

However, this flexibility eventually results in a complicated scheduling problem. The complexity rises even more with resource-elastic module properties defined in the background chapter. Furthermore, given the non-linear relationship between resource usage and performance, we need to examine the scheduling problem more in-depth as traditional constraint solvers become non-optimal with such problems.

Therefore at the end of this chapter, we formally defined the two main constraints of 1) fully executing all requested tasks and 2) placing accelerators correctly without overlaps and breaking implementation-specific constraints. Then we defined the two main objectives of 1) minimizing configuration cost by finding the cheapest set of modules and routing blocks and 2) minimizing streaming cost by using modules that reduce the required workloads the most. Both of these objectives combined allow the scheduler to minimise the total makespan. Next, a system is presented that implements optimised dataflow pipelines given these constraints.

Chapter 4

Implementation & Evaluation

First, this chapter explains implementation details for our targeted workloads in Section 4.1. After that, we show how the system is capable of running a Sobel filtering operation on any image in Section 4.2. Then in Section 4.3 we look at how resource elastic dynamic pipelines compare with static pipelines and CPU alternatives while executing SQL queries. This work resulted in our proof of concept platform, *OrkhestraFPGAStream*, that can manage both image processing and execute SQL queries and is publicly available on GitHub [197].

This chapter discusses design factors and design decisions. The exact implementation details and documented code are available in the GitHub repository.

4.1 System Implementation

In order to implement our target workflows for image processing and data analytics, we made design decisions to create general and modular subsystems. The following paragraphs will introduce these subsystem implementations and provide context for the evaluation presented later in this chapter.

Architectural layout: Fang et al. [77] discussed three ways of integrating an FPGA with a software DBMS: 1) the FPGA has a copy of the main memory separately on the device to alleviate the CPU bottleneck, 2) the FPGA is located between the memory and the CPU to alleviate the I/O bottleneck (e.g., Netezza did compression and filtering operations in this manner [79]), and 3) a co-processor model, where the FPGA and CPU share the main memory directly (e.g., the Xilinx Zynq or Intel Stratix 10 SX chips have this layout). Our system can support any of

these configurations - but for now, we use an UltraScale+ Zynq ZCU102, where the ARM core and FPGA are tightly coupled while sharing DDR memory.[1]

Middleware environment: As the middleware interfaces with various clients (e.g., DBMS applications), instead of running the middleware "bare-metal", we use an official Ubuntu (20.04.3 LTS) environment [42] where we run our application that uses Xilinx's FPGA Manager inside the Linux Kernel to reconfigure the FPGA through PCAP [42]. This official image already includes all of the board setup configurations for the ZCU102 while still providing enough flexibility for future changes to the system (e.g., architecture-wise changes or even simply changing the DDR memory size that involves changing parameters in the OS image and the first stage bootloader - FSBL). For reference, this Linux environment is built similarly to PetaLinux [330] that was used in FOS. The OS image, booted from an SD card, contains the following:

1. An initial FPGA design (to avoid sending erroneous signals during the start-up process).
2. Binaries for setting up the various stages of the boot process, trust zones on the CPU, and the PMU (which sets voltages for all devices on the board)
3. The Linux kernel and the device tree to tell the kernel about the board it is running on.

Using full Linux distributions on embedded systems has several advantages over more lightweight or bare-metal approaches, such as better third-party SW and HW support and, more importantly, increased security and networking tools which enable the potential to expand this system to be a small part of a more extensive distributed system in the future. There exist tutorials on the various boot steps to bring up Linux on an embedded platform [128], as this provides a unified yet flexible approach to running applications on different devices effectively.

Acceleration service requests: In order to provide a similarly unified yet flexible approach to interfacing with various clients, our middleware maps any incoming requests from client applications to an intermediate representation (IR). For data

[1] Even if the CPU and the FPGA have separate physical memories, resizable BAR techniques over PCIe [5] can help devices access all available memory directly, which is gaining support with the latest CPUs, and treat it as shared memory.

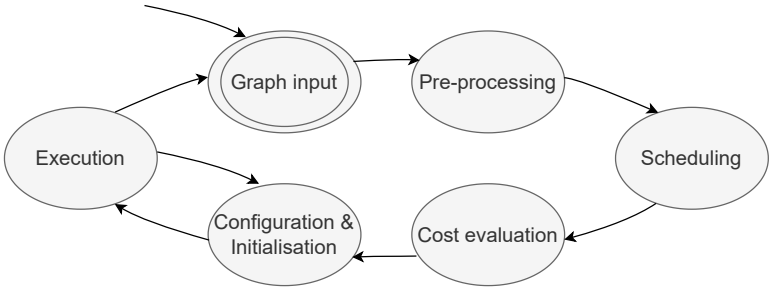

Figure 4.1: Middleware FSM to execute queries given as a graph IR. Front-end parsers (e.g., DBMS systems parsing SQL, as explained later in this chapter) locate dependencies in received requests and then compile IR data structures that will be passed to the scheduler. Then the best plan is fully executed before restarting the main loop.

analytics workflows, we parse the supported DBMS systems EXPLAIN[2] command output given an SQL query, which will then be rewritten to obey constraints induced by the heterogeneous FPGA. This IR represents a dataflow graph where the nodes (i.e., accelerator modules) are defined with the following information:

- Operation - The scheduler has to be aware of operator types, FPGA resources, and performance numbers.
- Operation parameters - Information about how operations are initialised and how the input/output streams must be parsed and created.
- Node edges - The number of node connections (including streams that are streamed to multiple nodes)
- Graph Input/Output - Additional information on data formats and how to handle I/O data.

While parsing incoming requests, the system maps streams with appropriate stream IDs and embeds the fine-grain data layout information into the IR.

Middleware states: The middleware operates in modular states, as depicted in Figure 4.1, which provides an overview of the middleware's finite-state-machine (FSM). The process begins by gathering and parsing input requests, initiating

[2]DBMS systems often use SQL, a declarative language, to describe *what* operation the DBMS must perform on which data. The query planner determines the computational details of *how* these results should be calculated. The details of this plan can be extracted using the typical EXPLAIN command in the absence of tighter integration.

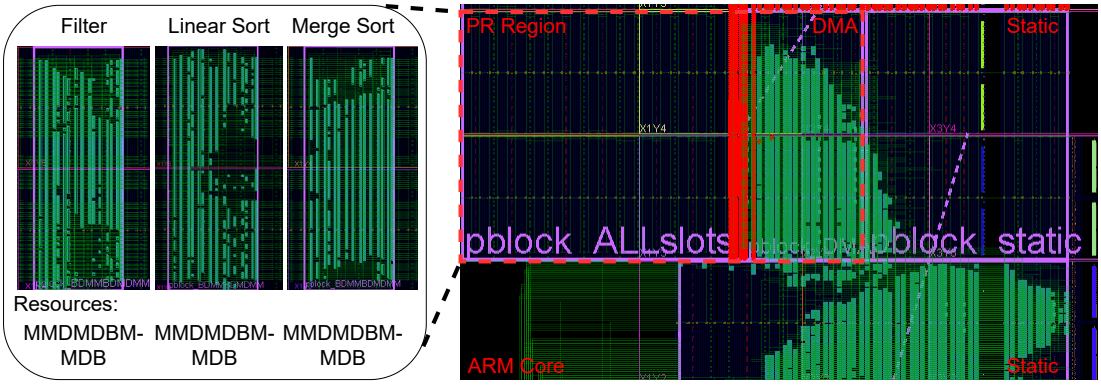

Figure 4.2: Different modules use the same resource string that appears multiple times in the PR region, enabling placing each of these modules into three different locations concurrently (on the top left of the static system). Additional PR regions can be built above the shown static system.

the next scheduling spin. Once all generated plans have been evaluated, the optimal one is selected, and the middleware proceeds to configure and initialise the FPGA. Subsequently, data processing occurs using the new FPGA accelerator pipelines until all planned runs are executed. It's important to note that new requests can be received at any point during this cycle. Therefore, the main loop restarts by collecting new requests and the results from the previous execution cycle, which in turn unblocks any leftover existing tasks. While this loop can be modified or parallelised, for the purposes of this evaluation, the middleware operates on a single thread to ensure the Linux system remains available for other processes that generate and collect new incoming requests.

Manipulating the FPGA: Meanwhile, the CPU is connected to a static system with multiple coarse-grain slots that are all memory-mapped to specific memory regions[3], similar to the ZUCL framework [241] (we built our system using a single such slot in Figure 4.2, however this can be extended when using simultaneous multi-PR region scheduling). FOS uses this framework (thus has been similarly partitioned) and has added libraries which suit our use, first for enabling writing to memory-mapped registers inside these regions and second for using the FPGA Manager for PR. The u-dma-buf [139] device driver helps with allocating continuous memory blocks in the Linux user space that the DMA module can access - for transferring large blocks of data from either CSV files (table data) or

[3]In more detail, each region has a corresponding AXI slave port that is mapped into the global address space such that these coarse-grain slots can be accessed individually by the middleware using the AXI master port.

BMP files (pixel data). Therefore, we can write data to DDR memory from disk using virtual pointers and then pass the corresponding physical pointers (memory addresses) for the allocated memory block to the DMA. The middleware organizes these data blocks as a simple in-memory file system and keeps track of the different streams' seek pointers given to the DMA engine as required to start executing on the FPGA after configuring the correct pipelines.

Partitioning FPGA resources: In order to configure the correct pipelines, the reconfigurable region is physically partitioned into different atomic resource columns (i.e., CLB, DSP, or BRAM columns, in our case). For our targeted UltraScale+ XCZU9EG, we define each resource column as 120 CLBs high (2 clock region or 2 ZUCL slot heights) for providing a 512-bit wide datapath and two resource blocks wide to include their shared switch matrix for routing. The smallest module in our library (pixel converter) takes two resource columns and the largest one (sorter) uses 14 columns, while the PR region consists of 31 resource columns. The fine-grain partitioning allows us to model module placement requirements through resource string matching without significant overheads. We omit using more fine-grain slots (when considering using smaller slots vertically) for now as they would require additional checks against breaking the timing constraints of the modules when moving a module from the initial location where it was built (synthesised) to the desired location, given that it has matching resources. Bitstream relocation is possible with bitstream manipulation tools like byteman [192] offline or online to limit the module library size, given a negligible PR latency tradeoff.

Modules: In order to keep such a system practical and flexible, it needs an extensive module library (how it could be expanded is explained in the next chapter). For each operation we support, the modules' bitstreams are stored with information about their possible placement locations, length and resource capacity. However, to run the modules, drivers must be created to calculate the initialisation values written to memory-mapped registers for each operation. The drivers also convey module-specific constraint information to the scheduler and help set stream IDs, and in the case of the DMA, also calculate the data layout and read sizes according to various packet sizes to make the flow (shown in Figure 4.3) operational. Consequently, the following subsection describes the modules used in our evaluation and their corresponding drivers.

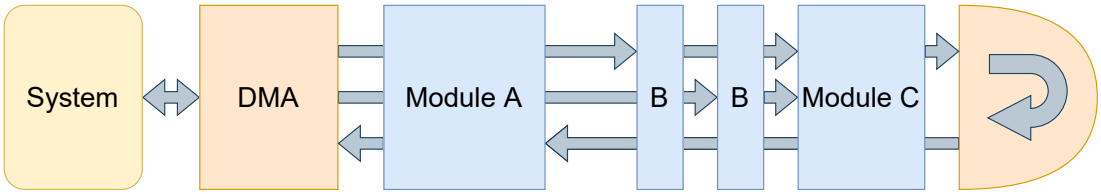

Figure 4.3: Data flows from the DMA through the modules back to the DMA, which communicates with the rest of the system.

Scheduling: All of this flexibility enables the scheduler to integrate the various module options together. Nevertheless, as the module library size grows with resource-elastic module alternatives and variants, the complexity of the scheduling problem rises exponentially, given the larger search space for finding valid module combinations. Therefore, we use heuristics to limit the exponential growth of the scheduling runtime. After introducing the module library and its drivers, we examine the heuristics to determine their efficacy in managing this complexity without adding significant computational costs or compromising quality. Our findings from synthetic benchmark simulations are presented in Section 4.1.3. Later, we conduct a comprehensive evaluation of the entire system on an FPGA, as discussed in Sections 4.2 and 4.3.

4.1.1 Data Mover

The most important module in the module library and the heart of the system is the data mover module, or as it is called in this thesis - the DMA module. The hardware specifications and its functionality are described by Manev et al., [194]. The DMA module is generally responsible for interconnecting the acceleration pipeline and standard AXI ports for memory access. However, besides address management, the DMA module also provides, as previously discussed, the necessary means for data prefetching, placing, projection, and duplication as needed by the currently executed queries.

Nevertheless, to make it work automatically in a dynamic system, it needs a supporting driver to guide the system (different DMA modules come with different characteristics and, ergo, new drivers) and operate the modules. The module drivers' primary responsibility is compiling the query node parameters into initialisation data written to the memory-mapped registers inside the modules. Each module has memory space reserved for memory-mapped registers, and the drivers and the static system know the address offsets of the different registers.

The runtime system configures the DMA by allocating more or fewer buffers for different streams (tables or images) with pointers to the data blocks in the main memory, after which it can start data prefetching. To explain it more in-depth, first, the DMA input and output channels can independently read and write concurrent streams. According to the initialisation order protocol (to utilise the self-scheduled flow control), these two channels have to be initialised in order, like any other module, which means that the input controller gets initialised first and the output controller gets initialised last. Therefore second, the DMA driver calculates the DDR burst counts and burst lengths to make them as large as possible without misalignment issues with the given buffer sizes, datapath widths and data packet sizes. Packets that have non-aligning sizes (when compared to the width of the datapath) get packed into bursts that contain enough packets to make the whole burst fit into the buffer without alignment issues (the DMA gets initialised how to handle these bursts and given the number of packets the DMA module handles the data amount pulled from RAM by itself). Then last, the drivers will send the start signals to the DMA channels and all modules in the pipeline that actively pull data and affect the data flow. After the input controller in the DMA fetches data from RAM, the active modules with tokens (and the output controller in the DMA for retrieving results start requesting more data from previous pipeline stages when they have space in their buffers. Meanwhile, the system keeps polling the busy flags for each stream through the driver logic to know when they finish and how large the resulting outputs are, allowing each stream to be processed independently and concurrently.

In order to enable the data formatting on the datapath, the DMA module features a crossbar between the buffers and the reconfigurable stream processing modules to reformat the data layout as needed. With one set of multiplexers, we can choose data words (in our case, 4 bytes or 32 bits) to be placed into a temporary buffer for intra-chunk placement (time-domain), which is then followed by another set of multiplexers, with which we can do the final inter-chunk word placement (space-domain), as shown in Figure 4.4. Therefore, as the DMA is initialised to use a set number of buffer blocks for each stream, the crossbars choose data from one already full buffer, while the DMA fills in other empty buffers with new data packets. This double-buffering is synchronised by the module itself, while the middleware divides the buffer space evenly between different concurrent streams. The same is happening concurrently in the output channels for writing back results.

This buffer usage allows the runtime manager to configure these crossbars (one in

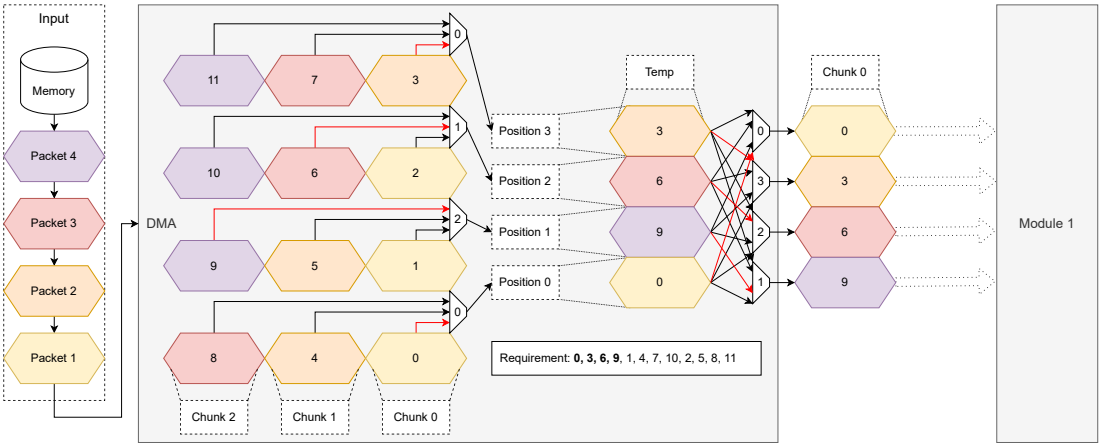

Figure 4.4: The crossbar consists of multiple MUXs that choose the source chunk and position each clock cycle allowing shuffling of data however necessary. Here, the term chunk is used to refer to the "x" coordinate of the targeted word and position is used to refer to the "y" coordinate, similar to the datapath, as when it comes to addressing specific words, the buffer has the same width as the datapath. Figure 4.5 illustrates how the high-level data sequencing requirements are broken down into these 'x' and 'y' coordinates.

each direction) with a sequence of multiplexer control settings derived from high-level projection requirements and module constraints. As data words may be shuffled both in time and space, we need the driver to translate the one-dimensional requirements into two different sets of MUX initialisation values, as shown in Figure 4.5. The MUX values for placing data words into correct chunks can be calculated by dividing the datapath width by the current placement requirement value, and the datapath position MUX value can be found similarly with the modulo function.

However, a temporary buffer holding only a single chunk of data is not enough to enable complete data placement freedom. This setup runs into a problem when we need to place two different words of data, initially from different chunks inside the buffer but at the same offset (position) within their original chunks, into the same chunk on the datapath, as shown in Figure 4.6. This clash can be detected when different initial requirement values result in the same modulo-function outputs. In order to solve this, more resources must be used on a larger DMA module or the system must adapt to the added latency and solve the problem by placing the clashing words of data into different chunks. Additional crossbar modules can be also used in the datapath before a module if necessary to meet any strict data formatting formats. The same problem can occur on the output side, but then the potential resulting gaps in the data can be removed through either post-processing in the CPU or with consequent runs reshuffling

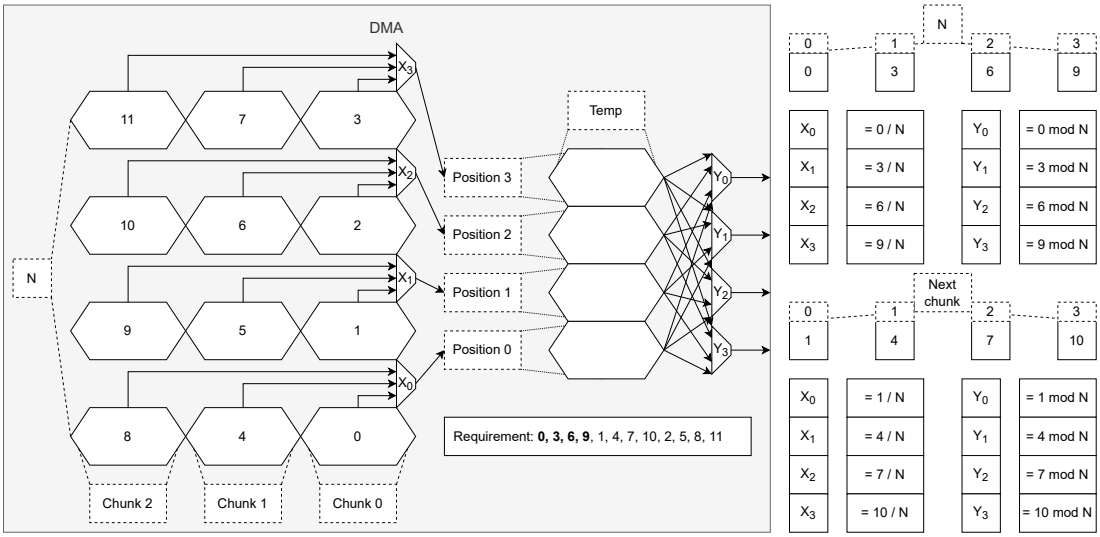

Figure 4.5: The high-level requirements get mapped to low-level MUX configuration values for each chunk on the datapath. If each chunk contains four words then we see how the requirements are broken down in steps of four calculating four MUX values choosing data in the "x" and "y" directions. How exactly the mapping corresponds to the requirements is shown in Figure 4.6.

the data.

The modules necessitate this shuffling of data; thus, the clash-solving has to be coordinated with the other module drivers, which we will look at next.

4.1.2 Module Library

Each operation accelerator module has a driver coordinating with the scheduler to ensure that the data packets are organised such that the corresponding module knows where to expect certain words while the format obeys all constraints. In this subsection, we will briefly look at the driver functionality of each module before looking at the scheduler to better understand the currently supported workloads. In addition to the active modules (these pull data through the pipeline and request for more, given enough credits), our library of modules also contains modules that are passive (operate on passing streams while not actively affecting the flow):

Filter: The filter module is a passive module, where a DNF boolean expression conveys the requirements set in the SQL query's WHERE clause. The expressions are divided first into comparison functions for calculating literal values with given constant values and secondly evaluated into DNF clauses that combine

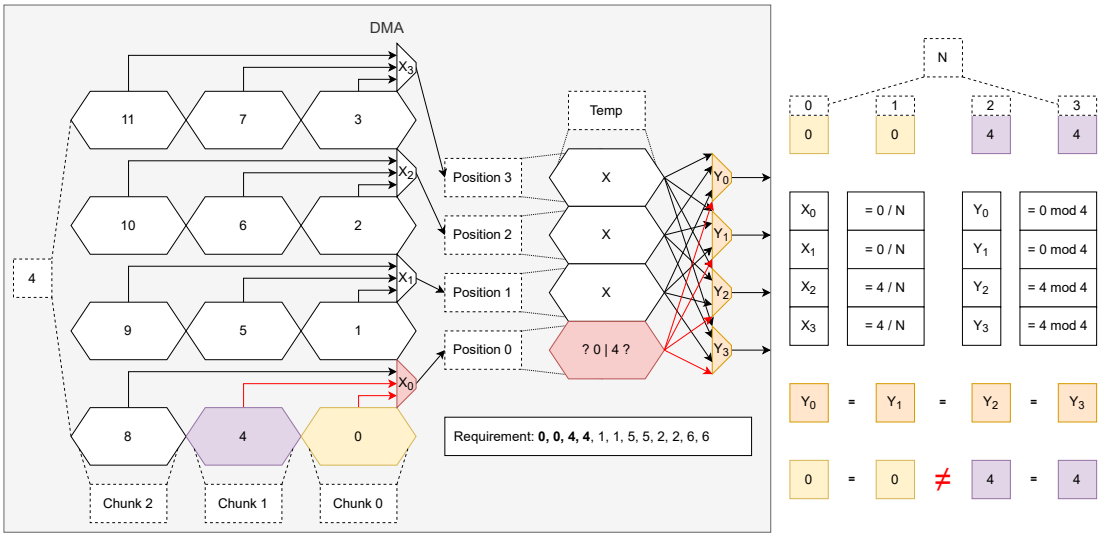

Figure 4.6: Data placement requirements may result in a conflict. When two words of data come from the same position but from different chunks within the buffer, then the crossbar cannot route both of these words onto the datapath in the same chunk. This issue can be resolved using more resources on additional buffers and crossbars or by placing these words into different chunks and resolving the conflict in time.

the literal values (all literal values are ANDed within the clause, and the clause values are ORed). The driver sets corresponding stream IDs for records with either a positive match or a negative match (i.e., for splitting the input stream into two resulting streams). The resulting stream consisting of negative matches may be explicitly required by the query or when using filters in a resource-elastic manner (when a single module may not have enough comparison units or enough resources for the required number of clauses). When multiple filters act as a single logical unit, records between the two streams can be reallocated to another physical resource-elastic filter module. A negatively matching record is turned into a positively matching record when a record matches another comparison in another ORed clause and vice versa with a negative comparison value match in an ANDed clause.

Join: An active module for joining two streams, acting as an inner join (equi-join) operation. The current system only provides a single join module which combines two sorted streams by the value in the first word. Additionally, the module contains a small crossbar for shuffling words in time to merge the two streams. Therefore the driver sets constraints to the scheduler that first, constrain the

matching words on which the join operation is done to be in the specified position, and second, the second input stream has to have some flexibility to shuffle data words without losing important data - which may require duplicating data beforehand to make the necessary space.

Linear sort: An active module for sorting smaller streams or starting to partially sort larger ones. Different modules can make short sequences of sorted elements of various lengths while sorting based on the value in the first word. As it is a stable sort, a crossbar (either a separate crossbar module or the one inside the DMA) must be used when sorting with multiple keys to reshuffle the values after sorting with one key before sorting again with another key.

Merge sort: An active module for sorting more significant streams through multiple partial runs. Multiple modules can cooperate to merge all sorted input streams into one combined sorted stream. Both merge and linear sort module drivers have to keep track of the size of these sequences to know how to mark the channel IDs of these sequences and how many further sorting runs are required. In addition, as the merge sort module has a variable size of buffer space, the driver has to calculate how many records get fetched per read burst and how these fit into the buffers on the CPU instead of these values being calculated in the module to save module resources.

Addition/Subtraction: A passive module for arithmetic operations. As the constant value can be added to the data value on the datapath, the requested result may also consist of the given constant value after the data value on the datapath was subtracted from it (meaning the module has to be able to calculate $x - y$ and also $y - x$). In addition, as we support adding or subtracting multiple decimal values concurrently that are 64-bit values, the pairs of 32-bit words have to be aligned correctly on the datapath - setting another constraint to the scheduler, which may result in an additional shuffling of data in the DMA.

Multiplication: A passive module for multiplying multiple 64-bit values in the datapath and writing the result back into the resulting stream. Handling 64-bit values can add additional constraints to the data formatting on the datapath as the multiplier module does not spend additional resources for internal data shuffling.

Global aggregation sum: A module that can be both passive and active based on the

requirements. An aggregation module that can delete a stream if required (active module) and otherwise output the input stream unchanged (passive module). Deleting the stream can make more space for other streams, albeit the stream cannot be deleted when it is still an input of another module (while aggregation is usually the final operation, the stream can be still used in a parallel query if they share the same data). Either way, the aggregation value is read from a memory-mapped register that must be manually zero-ed with the driver.

Black and white converter: An active module for image processing that converts pixel data encoded in three colours to a single intensity value. This module demonstrates the ability to convert data encoding schemes with modules.

Sobel: An active module for accelerating an operation that is used in edge detection, where the sharp edges of the image are highlighted using a convolution window. Here the driver checks image dimensions and passes them on to the module.

4.1.3 Scheduling Implementation

Now that we know all of the implementation-specific details for further context, we can devise a practical approach to optimise our processing pipelines. Fundamentally, if we can execute all of the operations in the correct order within a single run (all modules fit into the PR region), the scheduling problem is straightforward - the combination of modules with the most limited configuration cost is the most optimal. This remark is accurate, assuming the modules operate at the same speed while the primary bottleneck is memory throughput. As a result, we can use the simplified model for calculating the streaming cost explained in Section 3.3.4.

Scheduling approach

In practice we cannot always fit all required operations into a single run. With limited resources we commonly have to use time multiplexing while processing large datasets (which may involve using PR as shown in Figure 4.7), which results in forcing the scheduler to use various resource elastic modules. Therefore, this forms a non-linear programming problem (NLP), given the multiple objective functions (albeit they can be combined under the minimising overall execution time objective), numerous module placement and task ordering constraints, and most significantly, the non-linear relationship between resources and gained performance for larger and smaller modules. Such

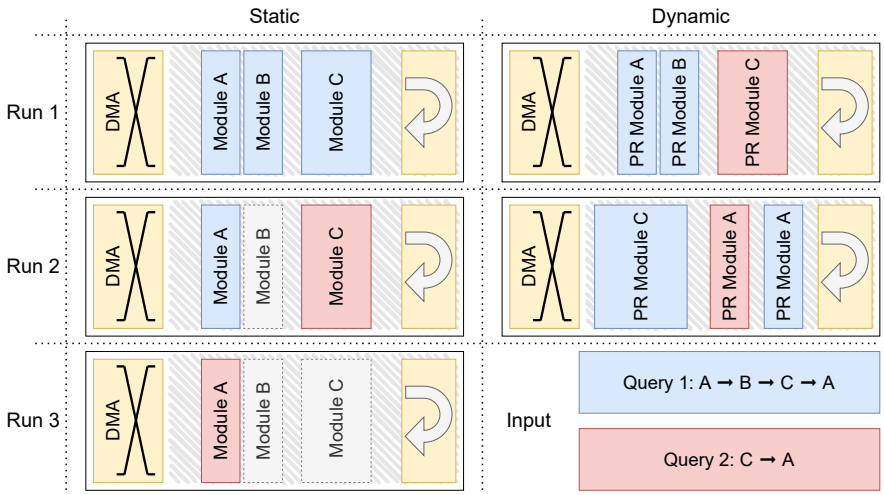

Figure 4.7: Operators of two concurrent acceleration requests can be scheduled with static and dynamic pipelines when multiple runs are required. Over-provisioning hurts performance when the static system needs to use an additional run to finish processing, and the cost of the additional I/O transactions is greater than the cost of PR required for the dynamic approach.

an optimisation problem consisting of scheduling modules both in time and space with functionality-performance-cost tradeoffs on different tasks with various dependencies is classified as a *resource-constrained project scheduling problem* (RCPSP).

RCPSP type of problems appear in many fields, and finding appropriate optimal scheduling approaches is a long-withstanding, decades-old, and active research field as surveyed by Hartmann and Briskorn [106, 107]. As the modules can be reprogrammed, this problem can be further generalised as an (Multi-Skilled RCPSP) MSRCPSP problem. Furthermore, as our tasks have partial-ordering constraints due to their dependencies and we have a non-preemptive execution model (a run lasts until the modules terminate execution), Blazewicz et al., [29] showed such MSRCPSP problems to be *NP-hard*. Consequently, related research shows that there are no fast general-purpose approaches to solve this problem [306, 274].

Other extensions of the RCPSP-type problems also apply to our scheduling problem and are similarly proven to be NP-hard. For example, another aspect of our resource-elastic modules is that the tasks can be completed in different ways (modes) that have different resource requirements and effectiveness levels which resembles a multi-mode RCPSP (MRCPSP - not to be confused with multi-skilled problems) as surveyed by Weglarz et al. [319]. These different project planning problem aspects can be combined, as done by Maghsoudlou et al. [188] who investigated a multi-skill

and a multi-mode problem. Similarly, our system can schedule multiple problems (multiple queries or executing data analytics and image processing dataflows concurrently) simultaneously. Consequently, related work shows how to combine multi-project RCPSP-type problems with a multi-mode type problem [16].

We need real-time adaptive scheduling for these problems, primarily because of resource requirement uncertainties, due to the data-dependent and non-deterministic execution. For example, no universal scheduling algorithms fit this RCPSP-type problem, and thus Rahman et al., [256] used a real-time meta-heuristic[4] approach based on probability distributions from historical data. However, as we currently target ad-hoc execution, we lack the necessary historical data and the optimal parameters to make a similar meta-heuristic approach effective. Furthermore, when reformulating the problem as a linear integer programming problem by enumerating all possible module selection combinations to decision boolean variables or when using non-trivial branch-and-bound and plane cutting techniques to reduce the runtime the constraint solver approaches are still too slow [60]. Consequently, most related work use heuristics to varying degrees for real-time scheduling [108, 156, 233]. Nevertheless, the system should be capable of adopting new scheduling strategies or algorithms when enough historical data is gathered. However, determining which parameters to monitor and how to formulate the problem to fit various meta-heuristic approaches falls outside the scope of this project.

Enumeration

First, the scheduler must solve the feasibility problem by finding all suitable modules for each node among the input requests and placing them in a valid order. This search entails branching search paths at each choice between different nodes, modules, module execution graph alternatives, module variants, and module placements (both in time and space). Therefore, after completing the input parsing, the scheduler starts finding suitable execution plans with a recursive (the recursion can be replaced with explicit

[4]Meta-heuristic approaches use learning to figure out optimal solutions from an existing history of solutions, as opposed to heuristics that use a trial-and-error approach, as summarised by Singh et al., [281]. Singh et al. also summarises most modern approaches, such as Ant or Bee colony algorithms, that often mimic life-like behaviour that solve complex problems appearing in nature. Meta-heuristic scheduling is a considerably thriving research field where different meta-heuristics and heuristics can be combined for hybrid solutions to solve state-of-the-art scheduling problems. As neural network workloads fit the dataflow model superbly with a plethora of effective FPGA accelerators, similar meta-heuristic approaches, like genetic algorithms, also have numerous FPGA accelerator implementations ([307, 299]) including HLS implementations [10] and automatic generators [97].

stack tracking) depth-first search algorithm shown in Algorithm 2:

Algorithm 2 Scheduling main recursive loop

Input: $Lib, AvailNodes, G, D, CurRun, CurPlan, AllPlan, BlockedNodes$
CurRun - Current run modules; CurPlan - Planned runs;
AllPlan - All valid plans; BlockedNodes - Cannot be executed;

Output: $AllPlan$

1: $CurAvailNodes \leftarrow AvailNodes - BlockedNodes$
2: **if** $CurAvailNodes \neq \varnothing$ **then**
3: $AvailModules \leftarrow \varnothing$
4: **for** each N in $CurAvailNodes$ **do**
5: $AvailModules \leftarrow AvailModules + GetAvailableModules(G, N, Lib, CurRun)$
6: **end for**
7: **if** $AvailModules \neq \varnothing$ **then**
8: **for** each M in $AvailModules$ **do**
9: $NewCurRun \leftarrow CurRun + M$
10: *Update variables and call this function again*
11: **end for**
12: **end if**
13: **if** $CurRun \neq \varnothing$ **then**
14: $CurPlan \leftarrow CurPlan + CurRun$
15: $NewCurRun \leftarrow \varnothing$
16: *Update variables and call this function again*
17: **end if**
18: **else**
19: **if** $CurRun \neq \varnothing$ **then**
20: $CurPlan \leftarrow CurPlan + CurRun$
21: **end if**
22: **if** $CurPlan \neq \varnothing$ **then**
23: $AllPlan \leftarrow AllPlan + CurPlan$
24: **end if**
25: **end if**

The algorithm requires the following input:

- A **HW library** that provides different module alternatives and variants. The library contains information about the locations, capacities, and sizes of the modules while mapping bitstreams to supported operations.
- A set of **available nodes** from which the scheduler can start the scheduling process (the input nodes).
- The input **service request** defining a stream processing problem (modelled as a graph). The input can contain multiple requests.
- The input **data parameters** that contain information about data sizes and data characteristics (e.g., if a specific field will be used for sorting).

The main idea of the algorithm is to enumerate all valid plans consisting of various runs (that, in turn, consist of different module pipelines at different positions that their respective drivers have sanctioned). In more detail, all nodes that have their parent (producer) nodes already scheduled or work on data streamed directly from the DDR are marked *available* (the available nodes are marked in the beginning of Algorithm 2). Then, in order to eventually examine all possible placements of modules in time and space, all modules that fit the set of available nodes are recursively placed to all positions in the current run (as shown in the for loop in lines 8-12 in Algorithm 2). Alternatively a next run is started (lines 13-17) where the whole process repeats until all nodes have been scheduled.

These different combinations of modules, organised into runs, create a set of possible global plans (which in Algorithm 2 is depicted as the output *AllPlan* that is populated in lines 18-25). Multiple modules in potentially multiple different runs may be required to execute an operation in one global plan, while only a single module (e.g., a static combination of multiple relocatable modules) may be required in another global plan. Therefore available nodes are updated separately in each recursive call for all global plans generated within a single scheduling spin. This set of plans will be evaluated in the subsequent cost evaluation state based on the given system-specific data streaming throughput and configuration writing throughput speeds to determine the fastest plan for execution. The configuration speed of each module is measured offline to estimate configuration times. Execution time estimates are based on measured streaming throughput, as it's the bottleneck; module throughputs are sufficiently fast to be omitted from calculations, as detailed in Section 3.3.4.

The rest of the inputs passed to Algorithm 2 are used to track the following in each recursive call: 1) the modules placed in the current run, 2) scheduled runs in the current

plan, 3) all plans generated so far, and 4) *blocked* nodes. The blocked nodes set is for marking nodes that are available for execution as their dependencies are scheduled, but the scheduler lacks information about how many resources should be allocated for these nodes. For example, streaming operations can output more or fewer data records than received, changing the data size unpredictably (e.g., filtering). Data-dependent consumer nodes with resource-elastic modules that do not fit into the same run as their producer nodes become blocked.[5] Blocked nodes become unblocked after analysing the intermediate data and updating the requirements, as in they can only be scheduled in subsequent scheduling spins. Therefore, the execution of more complex jobs that are split into multiple runs is possibly computed in multiple scheduling spins to adapt to changing runtime conditions.

When multiple scheduling spins are needed, the approach explained above will no longer generate a complete enumeration of all potential plans. With blocked nodes the scheduler does not start speculating or enumerating all possible outcomes of how it will become unblocked and instead will leave it to be scheduled in the next spin. This incomplete solution-space search may result in situations where by choosing a less optimal plan in initial scheduling spins a more optimal plan may become available in later spins which we will discuss when evaluating this approach after discussing available optimisations.

To conclude, if a fitting module in the module library exists for each node in the input request, this algorithm will always find a feasible plan, which may include multiple runs through the FPGA. A SW fallback must be used if there is a missing module implementation. While this enumeration problem will result in unpractical long execution times for online scheduling, it delivers the best solution and serves therefore as a baseline for heuristics that trade scheduling execution time for scheduling quality.

Heuristics

With such a vast amount of branching options, the scheduling runtime is exponentially growing with more complex requests to take a significant enough amount of time such that it hampers the overall performance of the system, regardless of the speedup

[5]Blocking nodes can be avoided by deliberately accepting over-provisioning and using modules large enough to cover the upper bound of the unknown workloads. The scheduler does this when there are unused resources, and no other module can be placed. Then instead of wasting these resources on routing, the resources can be allocated to the otherwise blocked node. If enough resources are allocated to cover the upper bound, the node never gets blocked, and the scheduler continues placing subsequent nodes.

gained with the chosen plan. When interpreting the well-known Amdahl's Law [13], we know that most performance is gained by optimising execution steps that take the longest, meaning we have to find the optimal balance between scheduling, execution, and configuration. However, we can create heuristics that aggressively choose plans with fewer major runs as they are the highest cost factor in performance with larger dataset sizes. Consequently, these heuristics allow us to reduce the cost of scheduling and execution, assuming the configuration time gains we can get through optimisation are smaller than the potential gains from optimising the other two aspects (given that most related works use static systems). We will confirm this assumption through experiments in Section 4.3.2.

However, first, we need to evaluate the effectiveness of the following heuristics in making the scheduler sufficiently fast to adapt to new requirements during runtime after a brief overview:

1. **H1** - Stop continuing scheduling plans that use more runs than the current best.
2. **H2** - Schedule modules first that already have producer nodes scheduled in the current run to avoid additional I/O operations.
3. **H3** - Schedule the smallest modules that can execute the required workloads. When the workloads are too large for any single fitting module, use the biggest modules possible with the leftover resources and process the operation with multiple modules or runs.
4. **H4** - Schedule as many modules into the current run as possible before scheduling modules into consecutive runs - minimising unused resources.
5. **H5** - Schedule modules as close to the DMA as possible for dense module packing.

The main idea we design our heuristics with is to pack everything as tightly as possible - as a consequence, we may miss opportunities to reuse modules over multiple runs as we try to replace all unused modules without considering the possible subsequent runs.

As the heuristics cut branching decisions that are not likely optimal, our first heuristic (H1) uses the branch-and-bound approach (as it is common in such scheduling problems [116]). We can cut down on our search space by calculating each option's lower bound of potential runs and comparing it to the current upper bound. Meanwhile, H2 is for minimising I/O costs by avoiding repeated streaming of parallel streams.

In order to use H3, the first step is to find all modules that can complete the job with one pass. This preprocessing step is to find modules in the HW library that can support the problem size even in the worst-case scenarios, as shown in Algorithm 3. As a result, the scheduler has a divided hardware library from which first it tries to find the smallest modules that cover the expected workload amounts, or, if no such modules fit the current run, then the largest possible module is used instead to do as much as possible with the given resources for partial results. Consequently, the final stream is calculated with fewer resources in the subsequent runs. Currently, algorithm 3 is reused every time with potential runtime requirement changes during scheduling for more accurate upper bounds. Alternatively, in the future, when there is enough historical data, this could be tweaked

Algorithm 3 Scheduling pre-processing

Input: $Lib, AvailNodes, G, D$

Lib - HW module library; AvailNodes - available nodes;
G - input graph; D - stream data

1: $MinACap \leftarrow GetMinAvailableCapacity(Lib)$
2: **while** $AvailNodes \neq \varnothing$ **do**
3: $N \leftarrow GetNextAvailNode(AvailNodes)$
4: $AvailNodes \leftarrow UpdateAvailNodes(G, N)$
5: $MinRCap \leftarrow GetMinReqCapacity(N, Lib, D)$
6: $FitModules \leftarrow FindFits(MinRCap, N, Lib)$
7: **if** $FitModules \neq \varnothing$ **then**
8: $G \leftarrow SetFitModules(G, N, FittingModules)$
9: **end if**
10: $WD \leftarrow GetWorstCaseD(G, N, Lib, D, MinACap)$
11: $G \leftarrow UpdateNextNodes(G, N, WD)$
12: **end while**

Lastly, the heuristics H4 and H5 follow a greedy scheduling strategy of packing modules as tightly as possible while avoiding leaving unused modules for routing and potential reuse in later runs.

Outside of using heuristics, when the scheduler spots a set of conditions it has solved before while finding available modules, it uses a dynamic programming approach and retrieves the previously chosen set of modules from memory instead of

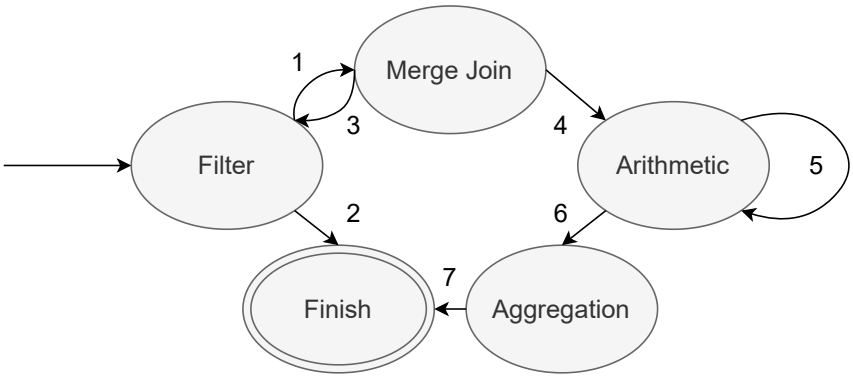

Figure 4.8: Benchmark generation finite-state-machine (FSM), where each state has a chance to generate the corresponding node into the benchmark query set.

recalculating them. Therefore, we exchange increased memory usage for faster execution as the middleware is likely the only application on the CPU running with high memory requirements. Additionally, we can minimise module library size when required and already have dedicated memory regions for the data blocks to leave more memory for the scheduler. Furthermore, the additional memoisation overhead is negligible, given that long search paths also take substantial memory.

Synthethic benchmark for heuristics evaluation

To stress test our scheduling methods, we implemented a data analytics query generator that creates requests with various characteristics. As a result, we have a dynamic benchmark with ad-hoc requests that uses supported operations from the module library, similar to the "SELECT FROM WHERE"-type SQL queries in the well-established TPC-H benchmark. This approach can generate more complex queries with more operations like in the more extensive TPC-DS benchmark. The queries are generated with a Markov Chain-like FSM (similar to the unstructured aspects that are generated for the queries in another data analytics benchmark BigBench [88]), as shown in Figure 4.8. There is a chance with a parameterised probability that a corresponding operation is added in each state to the generated query before transitioning into the next state. Each numbered state transition takes place with the corresponding conditions:

1. Commonly **joins** (and the prerequisite **sorts**) come after **filters**.

2. There is a chance to use an existing **join** (if there is an empty input) and stop.

3. If a **join** gets generated then further **filters** are added (creating an arbitrary sequence of joins and filters).
4. After **filters** and **joins** come **arithmetic** operations.
5. There can be multiple **arithmetic** nodes.
6. After **arithmetic** nodes comes the **aggregation** operation.
7. Lastly the query is finalised.

These steps generate desired non-nested queries (nested queries can be simplified in SW beforehand[6]) using modules with different characteristics. The modules we have are for filtering, join, and sorting operations, which have data-dependent behaviour with different numbers of inputs and outputs, in addition to having both resource elastic modules and non-scaling modules.

In summary, the custom benchmark generator serves as a framework for validating the scheduler's performance across a wide range of scenarios - allowing to change and rapidly test the scheduler without targetting specific queries. The framework also enables monitoring of the scheduler's performance with varying data loads while providing insight (with automated performance illustration shown in the following subsection) about the system's efficiency under different conditions. Moreover, the platform created by the framework can be utilized to generate historical data for developing meta-heuristic approaches or for training DNN models in future work.

The randomly generated test sets were used to evaluate our heuristics on an Ubuntu (20.04.5 LTS) desktop machine with an Intel Core i7-4930K (22 nm) CPU with 64 GB with 4 DDR3-1333 memory modules. While using different scheduling parameters with query sets of varying complexity, we compared the resulting scheduler runtimes and execution plan quality. As we will evaluate the system with TPC-H workloads on a ZCU102 with an XCZU9EG chip (16 nm), we measured the configuration speed for each bitstream available in our library. For reference, the FPGA Manager in the Linux Kernel loads bitstreams through PCAP with a small constant overhead cost, and as such, small bitstreams get configured with $\approx$115 MB/s and a larger module with $\approx$270 MB/s. To improve configuration speed, we use our DMA module's maximum streaming speed on the ZCU102, which we were also able to reach in practice, where

[6]When one query depends on the result of another, the scheduler realises this constraint and can generate the prerequisite result before scheduling following queries. When multiple queries or multiple operation nodes depend on another query, then if it is not possible to directly stream the resulting stream to all required consumer nodes, the results get temporarily saved in memory for sharing.

Table 4.1: Specification of generated queries for measuring scheduler's performance in Figure 4.9.

Data Set Parameter	Average	Std Dev	Min	Max
Query count	1.37	0.67	1.00	3.00
Avg node count per query	5.10	1.55	1.67	6.00
Avg table count per query	2.65	0.57	1.33	3.00
Avg table size	10041.17	3549.41	1466.00	19901.00

our AXI interfaces run with 300MHz clocks to achieve an aggregated 4.8 GB/s read and 4.8 GB/s write throughput.

Quality of results

In order to measure the effectiveness of the heuristics presented earlier, we generated queries with rows of 40 bytes of data (to stay comparable with row sizes in TPC-H queries) and compared them with two additional scheduler configurations:

- **H0** - Schedule without heuristics for most optimised plans.
- **H6** - Schedule with all heuristics to reduce search space.

Figure 4.9 shows that our heuristics individually do not have a significant enough impact on reducing the scheduling runtime, but as we add additional heuristics, the runtime eventually gets reduced from seconds to less than a millisecond.[7] These relatively small runtimes are achieved with small datasets as we compare results from 434 runs where the scheduler ran to completion and scheduled, on average, six nodes in a request with 3500 rows of data, with a minimal number of requests consisting of more than 1 query, as seen in Table 4.1.

Nevertheless, when looking at the quality of the resulting execution plans in Table 4.2, we see that the execution times were kept low as intended. The execution time gets reduced from 0.43 milliseconds to 0.40 milliseconds at the cost of increasing the configuration time from 17.9 milliseconds to 25.6 milliseconds with this small dataset scenario. Consequently, we see that the tradeoff of using the heuristics is increased configuration cost, which dominates the overall processing times with such

[7] Similarly to using multiple heuristics in our scheduling, in face recognition applications, the Viola-Jones algorithm is a well-established approach used on low-power devices to detect faces through numerous simple filters [314]. A single filter alone is not accurate enough to detect faces, but collectively the accumulated effect is substantial and reliable enough to warrant use over more computationally intensive machine learning models.

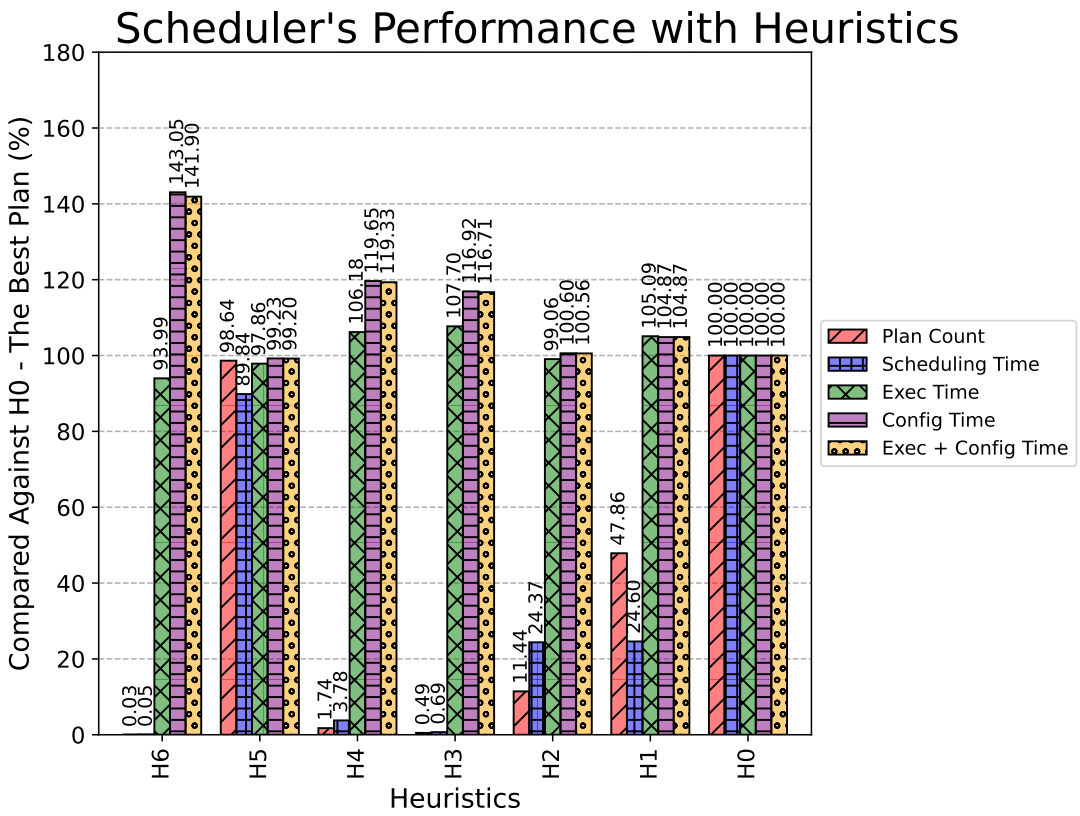

Figure 4.9: Using heuristics, execution plans are generated faster yet maintain the speed of plans from exhaustive searches. H6 prunes much of the search space compared to H0. The drawback here with small datasets is longer configuration, but for larger datasets, the overall makespan is unaffected as execution time dominates.

small datasets (as expected, but this is not an issue with larger datasets, which we will look at next). As a side note, we also see that H5 configuration can find better query plans than the exhaustive search can find because of a few exceptions when multiple scheduling spins are required, where overall better performance is achieved by choosing "worse" plans in the first spin that can enable finding overall faster query plans in later scheduling spins. The exhaustive search does not try to predict unknown data load amounts after scheduling non-deterministic modules and hence cannot remove the use of multiple scheduling spins, where the second execution plan can be different based on the previous plan.

Larger datasets runtime

In order to investigate scheduling times and execution times of more complicated requests, we generated a new set of queries with parameters shown in Table 4.3 to show

Table 4.2: Comparison of heuristics effects while scheduling with the dataset parameters from Table 4.1.

Performance Parameter	H0	H1	H2	H3	H4	H5	H6
Scheduling Runtime	0.0005	0.8776	0.0369	0.0067	0.2381	0.2403	0.9769
Plan Count	2.91	11051.38	195.36	55.14	1281.96	5362.28	11203.37
HW Configuration Time	0.0256	0.0177	0.0214	0.0209	0.0180	0.0188	0.0179
HW Execution Time	0.00040	0.00042	0.00046	0.00046	0.00043	0.00045	0.00043
HW Exec + PR Conf	0.0260	0.0182	0.0219	0.0214	0.0184	0.0192	0.0183

Table 4.3: Specification of generated queries for measuring scheduler's performance in Figure 4.10.

Data Set Parameter	Average	Std Dev	Min	Max
Query count	1.84	0.91	1.00	5.00
Avg node count per query	10.95	4.79	2.00	18.00
Avg table count per query	3.73	1.11	1.40	5.00
Avg table size	54860.63	12904.37	17746.25	91404.50

how parallelism, input size and operation count increase the complexity of the scheduling task. We see that the scheduling runtime can still explode as only 78% of the 1202 queries generated are still fast to schedule (below half a second), as shown in Figure 4.10. However, because of both the freedom to execute parallel queries in any order and the additional minor and major runs required with increased data set sizes, there exist still a lot of suitable execution plans that have little difference in quality. Furthermore, we can see a classical performance bimodal distribution featuring two distinct peaks in the histogram of scheduling performances in Figure 4.10, where the second peak could be attributed to cache misses.

Therefore, to handle these few exceptions consistently, the last way to prune the search space is to limit the search time after finding the first valid plan. For the last scheduling test, we created another set of 654 tasks that are sufficiently complex such that for all of them, the scheduling process takes longer than 3 seconds, even with the heuristics. These tasks, on average, have 20 nodes between three and a half parallel queries with tables having 300 000 entries on average, as shown in Table 4.4. In Figure 4.11, we set different time limits to the schedulers with different heuristics and see that giving more time to the scheduling process only provides diminishing returns. Thanks to the heuristics, finding a set of initial plans with reasonable quality is fast, and consequently, we can use a time limit to avoid looking at a more extensive set of plans. Based on these evaluations, we have set a time limit for practical experiments in the next section at 0.1 seconds for each scheduling spin, after which the best plan is

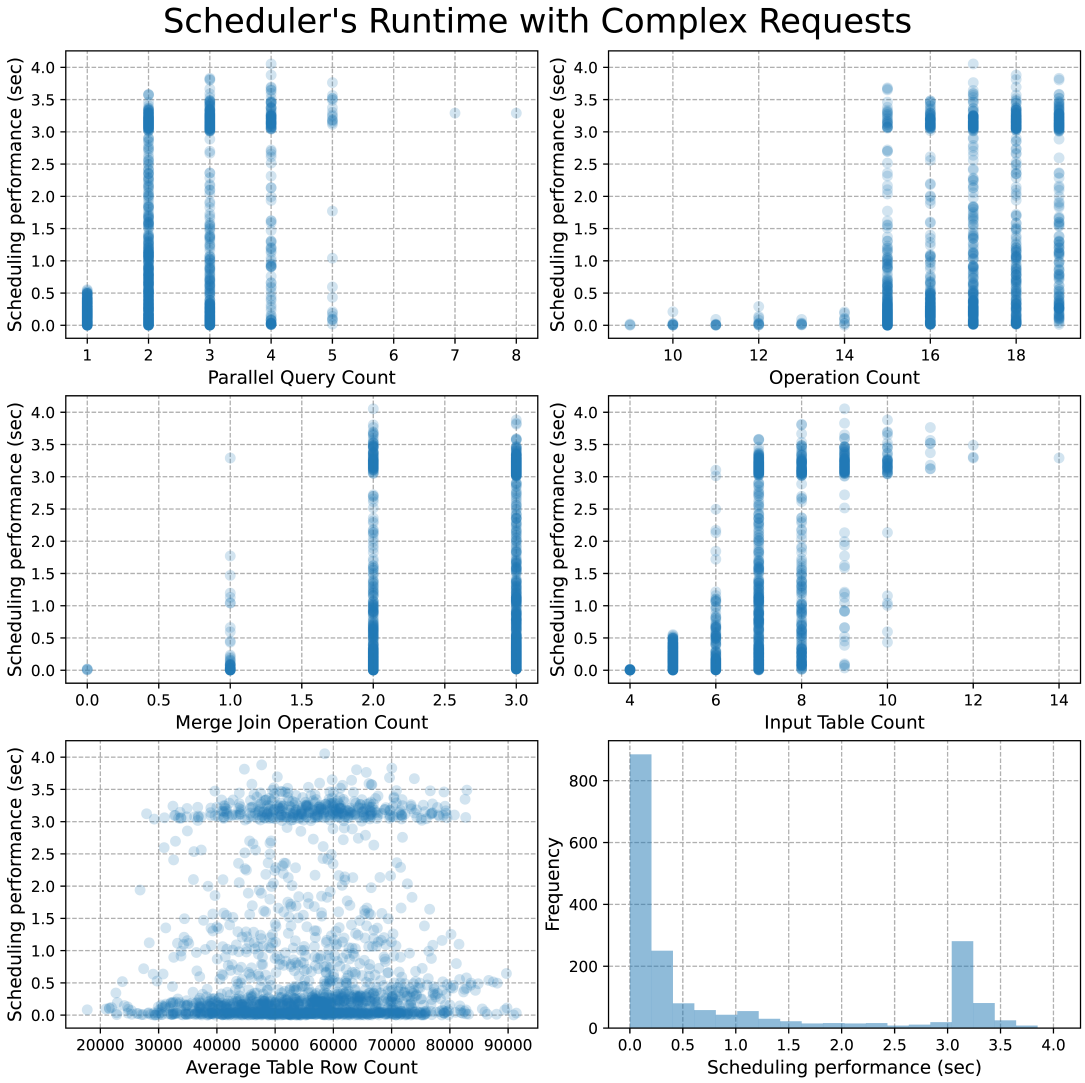

Figure 4.10: Many factors can increase the complexity of finding the best execution plan. Most prevalently, a high operation count implies higher parallelism and more chances to have more merge sort and join operations.

Table 4.4: Specification of generated queries for measuring scheduler's performance in Figure 4.11.

Data Set Parameter	Average	Std Dev	Min	Max
Query count	3.50	1.41	1.00	10.00
Avg node count per query	7.04	3.94	2.17	24.00
Avg table count per query	2.81	0.87	1.50	6.00
Avg table size	300095.54	46815.38	159272.25	434203.17

chosen.

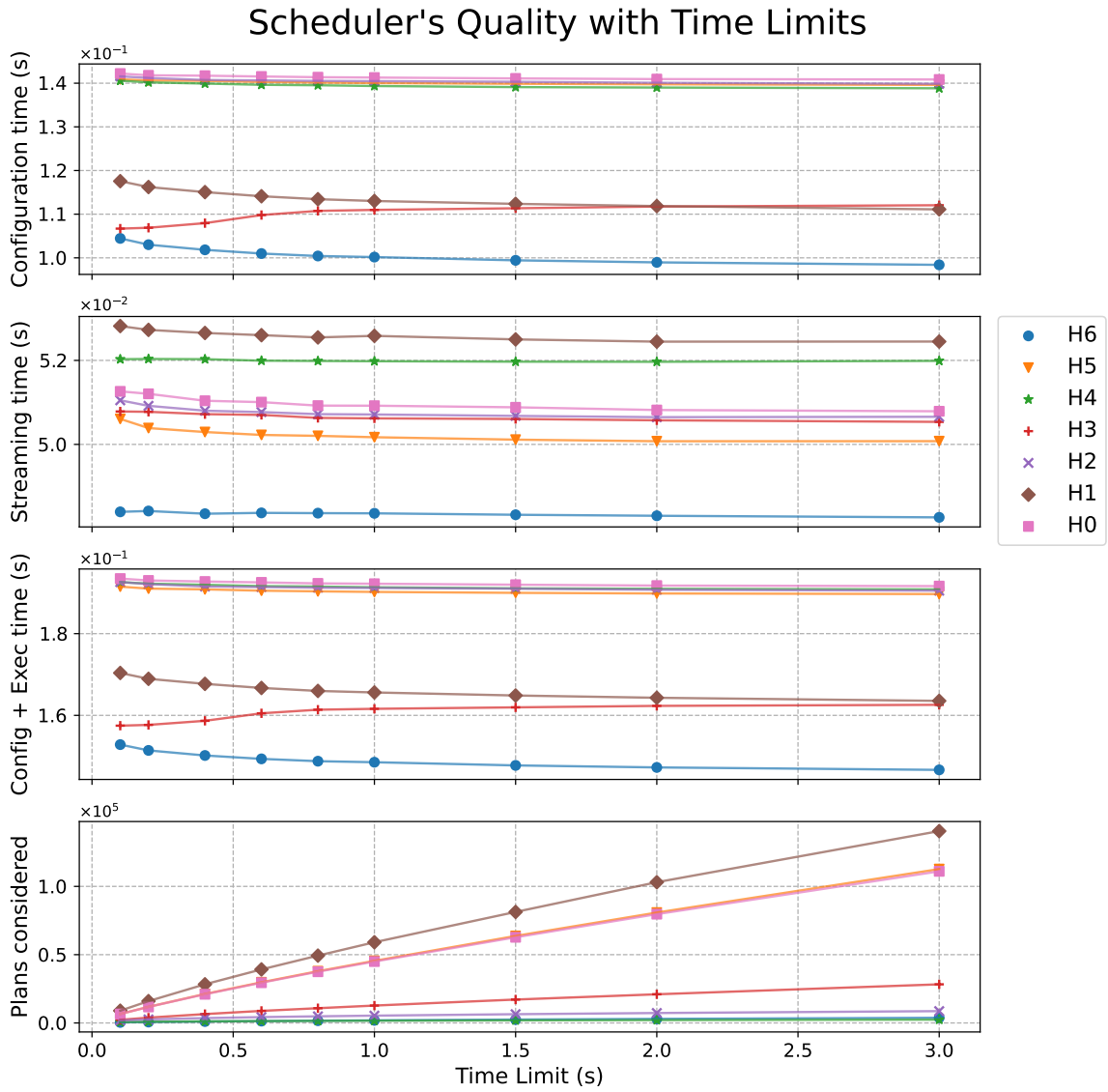

Figure 4.11: With requests still having substantial scheduling runtimes due to many execution plans being found, we can limit the time spent on scheduling. The significant runtime points to many equally good plans from which the scheduler should choose the ones it happens to find first. Therefore, spending more time scheduling these problems does not give substantially better quality plans.

Using fast heuristics to prune such vast search spaces is helpful for a general-purpose approach that can be used even on low-performance CPUs (or even a soft-core CPU) with the assumption that the data streaming times strongly dominate with large dataset sizes. In the following evaluation, we confirm that the system is operational (Section 4.2). Then in Section 4.3, we evaluate scheduling to examine the relative sizes

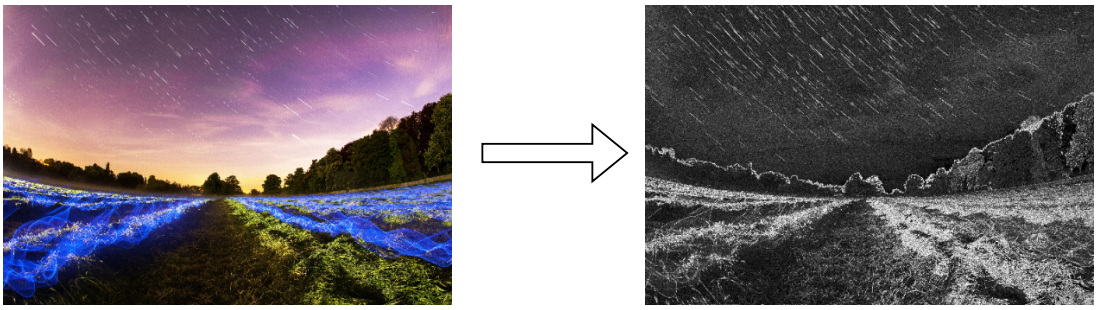

Figure 4.12: Our system can process any image using the prerequisite black and white converter and Sobel filter operations as long as the image's width is a multiple of 64 (could always be padded accordingly).

of scheduling, configuration and execution times on the FPGA board with queries from the TPC-H benchmark.

4.2 Image Processing Example

For our evaluation, we first check that the system works in practice and successfully packs data into packets in a format dictated by our modules. Therefore, we use the two image processing modules: the black and white converter module and the Sobel module. These modules are for accelerating a two-step (daisy-chained) dataflow edge-detection task with images of varying sizes, as demonstrated in Figure 4.12. A basic implementation of *OrkhestraFPGAStream* tested the following implementation details:

Running the middleware: First, the CPU on the Zynq XCZU9EG chip is a 64-bit 4-core ARM Cortex-A53 that can run at 1200MHz. This system successfully boots up with Ubuntu 20.04.4 LTS for running our middleware and providing all libraries that are common on desktop systems.

Successful memory blocks allocation and use: In order to share memory for reading/writing input/output data between the CPU and FPGA, we allocated a large memory area on our 2666 MHz 4GB DDR4 RAM through Linux's Contiguous Memory Allocator (CMA). Inside this area, new *u-dma-buf* devices allow the creation of kernel-level memory blocks with a physical memory pointer (for only the DMA module to access). Meanwhile, as mentioned before, these blocks can also be accessed as user space memory blocks by the middleware, given that it has retrieved the virtual pointers from the corresponding devices. However,

due to reserved memory in the middle of the memory area for operating the MPSoC, we can only allocate a CMA memory area with a size of close to 2GB. Inside this area, we built a filesystem with pointers to different data blocks held in memory for streaming. For example, to only process one image, we can use up to 2GB of memory - 1GB each for storing the input and output.

Initialising modules with data formatting requirements: The execution plan for a two-step dataflow process is straightforward after creating two directly connected nodes where the black and white converter node reads the image input, and the Sobel operation node writes the final output. The Sobel operation node has to contain the image size parameters for the driver to write these values into the module during initialisation. As the black and white module expects the image pixel data to be encoded as 3-byte RGB values, they enforce the images to be at least 64 pixels wide to pack multiple rows of pixels onto the 512-bit wide datapath without alignment issues (e.g., one row of pixels for a 64-pixel wide image will get streamed in three clock cycles). The Sobel filter module operates on data in which the brightness of each black and white pixel is represented as a byte value. As a result, the execution plan needs to specify the number of bytes required for each pixel in the image processing streams. This flexibility allows the system to work with both image formats by telling the DMA how many clock cycles it takes to stream a row of pixels. Consequently, the DMA gets initialised to place data linearly onto the datapath while avoiding any data shuffling for the data that arrives from the main memory.

Using PR modules: The Sobel module fits into three locations, and the black and white converter module fits into six different locations while they have no resource-elastic alternatives or variants, and collectively they fit the PRR 15 times without any overlaps. We confirmed that the modules work in both placement scenarios: 1) they are as far from each other as possible (to test setup time violations), and 2) they are directly connected (to test hold-time violations). In larger systems, placing buffers between two modules that are too far from each other might be necessary to fix timing errors, but this is not required in our system, and in practical scenarios, there will likely be other modules placed between them.

Successful acceleration: Overall the flow was operating from reading 4k RGB input frames to writing 4k black and white outputs into the main memory with 87 FPS. However, we spend many resources on the DMA module to be able to shuffle

data which is wasted in this case. Such a flexible system needs to provide more value than the overheads of this increased flexibility to see any performance benefits. Many related works come to this same conclusion and stop here and start using ASICs or GPUs for acceleration instead, but we want to show that with more dynamic flows, such a system can effectively exploit the benefits of resource elasticity.

As the optimising of all of these modules and static setups is nuanced enough with different tools and levels of constraints and code quality, drawing any conclusions on the number of additional resources required for partially reconfigurable modules falls out of scope for this thesis and, instead, we examine the middleware. Nevertheless, the resource requirements of all modules and comparable static pipelines are listed in Appendix A for reference. The static pipelines are constructed by synthesising the PR modules together. They are constrained to use resources only within the PR region and rely on the surrounding static shell for communication with the PS Arm Processor and main memory. A more detailed comparison between the static pipelines and PR modules will be presented in the next section.

In the next section, we will demonstrate the effectiveness of this approach on more dynamic problems with a larger module library while providing resource-performance trade-offs so that the scheduler can dynamically create optimised pipelines. Data analytics is appropriately more dynamic, but additionally, we process the requests using the same system with the same data structures as was used for the image processing example to confirm the versatility of our approach (both types of requests could even be processed in parallel) which is further examined next in Chapter 5.

4.3 TPC-H Evaluation

To study how different processes contribute to the overall runtime in dynamic stream processing systems, we target queries from the TPC-H benchmark, which, as discussed in Section 2.1.2, is grown to be the industry standard for benchmarking systems processing data analytics. According to Boncz et al., [33], the TPC-H benchmark has many challenges, which can be grouped into aggregation performance, join performance, data access locality, expression calculation, correlated subqueries and parallel execution. To understand the execution side of the resulting performance while running TPC-H queries, we first discuss data types, operations, tables and acquiring query plans.

First, the benchmark queries are built up using filtering, sorting, joining, aggregation, and arithmetic operations. However, as we lack modules capable of processing substreams partitioned by a value within the original stream necessary for the GROUP BY operation, we omit queries with these operations as related work covers the usage of this operation in great detail already (e.g., Ibex [324]). Furthermore, we lack a regular expression parser as it does not offer any new interesting streaming characteristics that the scheduler does not yet support and acts mainly like a filter, and as such, this runtime parsing is also omitted in related works targeting TPC-H (e.g., Q100 [325]). Outside of targetting the TPC-H workloads, numerous regular expression parsing accelerator implementations exist, like in deep packet inspection algorithms [334] that, for example, are used in network intrusion detection systems [285]. Nevertheless, we can still use a filtering module to find pattern matches that target the beginning of the string by either finding exact matches or by finding the suitable range of ASCII values if we need to match strings whose length is not a multiple of 4 (size of a word in a chunk). Lastly, the division operation is often done with a single literal value, and as such, we leave the division operations for the CPU as a postprocessing step.

Therefore, to challenge our assumptions about the efficacy of using PR systems for dynamic dataflow workloads like data analytics, we target queries 6, 14, and 19 (created with an available TPC-H query generator [243]) shown in the following listings:

```
SELECT
  SUM(l_extendedprice * l_discount) AS revenue
FROM
  lineitem
WHERE
  l_shipdate >= DATE '1994-01-01'
  AND l_shipdate < DATE '1994-01-01' + interval '1' year
  AND l_discount BETWEEN.06 - 0.01
  AND.06 + 0.01
  AND l_quantity < 24;
```

Listing 4.1: TPC-H Q6

```
SELECT
  100.00 * SUM(
    CASE WHEN p_type LIKE 'PROMO%' THEN l_extendedprice * (1 -
    l_discount) ELSE 0 END
  ) / SUM(
    l_extendedprice * (1 - l_discount)
  ) AS promo_revenue
```

```
FROM
  lineitem,
  part
WHERE
  l_partkey = p_partkey
  AND l_shipdate >= DATE '1995-09-01'
  AND l_shipdate < DATE '1995-09-01' + interval '1' month;
```

Listing 4.2: TPC-H Q14

```
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 (
```

Table 4.5: An example record from the *lineitem* table showing its data types and column values. (1/3)

Column name	L_ORDERKEY	L_PARTKEY	L_SUPPKEY	L_LINENUMBER	L_QUANTITY
Column type	INTEGER	INTEGER	INTEGER	INTEGER	DECIMAL(15,2)
Example value	1	1552	93	1	17

```
34    p_partkey = l_partkey
35    AND p_brand = 'Brand#34'
36    AND p_container IN (
37      'LG CASE', 'LG BOX', 'LG PACK', 'LG PKG'
38    )
39    AND l_quantity >= 20
40    AND l_quantity <= 20 + 10
41    AND p_size BETWEEN 1 AND 15
42    AND l_shipmode IN ('AIR', 'AIR REG')
43    AND l_shipinstruct = 'DELIVER IN PERSON'
44  );
```

Listing 4.3: TPC-H Q19

All eight tables used in the TPC-H benchmark are in the *3rd normal form* (which is a database schema type where information duplication in tables is reduced by having column values directly related to the primary value or key in that table). Our selected queries only process two tables - *lineitem* and *part* table, which grow linearly with the scale factor (SF) and have uniformly distributed data. The *lineitem* table is the biggest one in the data set (∼70% of the whole dataset size), while the *part* table is roughly 2%. The tables consist of all the data types present in the benchmark: INTEGER, DECIMAL(15,2), CHAR, DATE, and VARCHAR (as is shown in Tables 4.5 and 4.6).

We padded the values with whitespace for VARCHAR column types until the maximum length was filled. In future systems where such padding adds too much additional data to each record, additional data encoding schemes have to get used. Furthermore, all columns with string values were padded to have a length equal to a multiple of 4 bytes (where one character is encoded with 1 byte) to avoid alignment issues. This padding could be reduced by combining different column values to have a fitting length collectively, but again, this padding only had a negligible effect on the data size.

How many records are in each table is dictated by the scale-factor (SF) of the generated benchmark test table set as shown in Table 4.7. The scale factor number shows how many GB of data does the whole database contain.

Table 4.5: An example record from the *lineitem* table showing its data types and column values. (2/3 *Continued*)

Column name	L_EXTENDEDPRICE	L_DISCOUNT	L_TAX	L_RETURNFLAG	L_LINESTATUS
Column type	DECIMAL(15,2)	DECIMAL(15,2)	DECIMAL(15,2)	CHAR(1)	CHAR(1)
Example value	24710.35	0.04	0.02	N	O

Table 4.5: An example record from the *lineitem* table showing its data types and column values. (3/3 *Continued*)

Column name	L_SHIPDATE	L_COMMITDATE	L_RECEIPTDATE	L_SHIPINSTRUCT	L_SHIPMODE	L_COMMENT
Column type	DATE	DATE	DATE	CHAR(25)	CHAR(10)	VARCHAR(44)
Example value	1996-03-13	1996-02-12	1996-03-22	DELIVER IN PERSON	TRUCK	egular courts above the

Table 4.6: An example record from the *part* table showing its data types and column values. (1/2)

Column name	P_PARTKEY	P_NAME	P_MFGR	P_BRAND
Column type	INTEGER	VARCHAR(55)	CHAR(25)	CHAR(10)
Example value	1	goldenrod lavender spring chocolate lace	Manufacturer#1	Brand#13

Table 4.6: An example record from the *part* table showing its data types and column values. (2/2 *Continued*)

Column name	P_TYPE	P_SIZE	P_CONTAINER	P_RETAILPRICE	P_COMMENT
Column type	VARCHAR(25)	INTEGER	CHAR(10)	DECIMAL(15,2)	VARCHAR(23)
Example value	PROMO BURNISHED COPPER	7	JUMBO PKG	901.00	ly. slyly ironi

Table 4.7: For the TPC-H benchmark, the *lineitem* and *part* table sizes grow linearly with the scale factor SF.

Table name	SF 0.1	SF 1	SF 10
lineitem	600572	6001215	59986052
part	20000	200000	2000000

4.3.1 Parsing SQL

As most gains in performance are achieved with optimised query planning as discussed in the background Section 2.1.2, the first step to running SQL queries is getting an IR graph where the declarative SQL statement is transformed to a concrete set of operational steps by an established DBMS system beforehand. For example, for TPC-H query 19 (Q19), three ways immediately stand out to complete the query when using the same operator set as shown in Figure 4.13. Therefore currently, our middleware

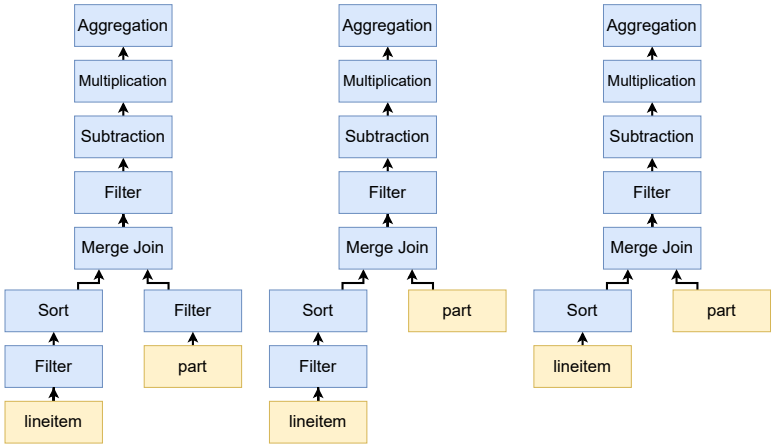

Figure 4.13: Potential query plans for the TPC-H query 19 using the merge join operator.

works with the one given while leaving any query optimising steps for preprocessing. The middleware may only replace hash operations (like replacing hash joins with merge joins due to module availability) or change the order of the input streams used in join operations as an option to resolve data placement conflicts on the datapath.

We look at two established open-source DBMS examples in this thesis: 1) PostgreSQL as one of the most popular general-purpose DBMS, and 2) Apache Trino (used to be called PrestoSQL) as a distributed big data query engine specifically designed for data analytics and querying data from memory nodes that another DBMS like PostgreSQL may locally manage.

First, PostgreSQL provides the capability to extract the query plan by utilising "query hooks," which can be implemented through custom plugins that intercept and modify both the query planning and execution processes. For example, these hooks were successfully utilised by Sharygin et al., [279] to improve PostgreSQL's performance with dynamic online optimisations despite the overheads of this monitoring (similar to what we want to achieve in HW). In detail, upon compiling and installing a relevant extension, it can be loaded into the PostgreSQL server using the "CREATE EXTENSION" command. As a result, we enable the automatic triggering of custom functions during query planning, enabling us to extract the query plan.

Second, a similar mechanism is available in Apache Trino, allowing for the interception and modification of the query execution process. However, effectively exploiting these features requires a comprehensive understanding of the underlying codebase of the DBMS. For instance, in the case of PostgreSQL, the utilisation of these hooks may necessitate the disabling of JIT compilation, along dealing with any following

issues.

As an alternative to such interfacing with each new DBMS, a more universal approach is utilising ODBC driver connectors. However, in a prototype setup where the DBMS system is installed on the same device, using ODBC and the associated networking overheads can be avoided. Hence, finally, we used a common way to obtain the query plan from these DBMS systems (with or without ODBC), which is to use the "EXPLAIN" command in combination with the desired query, as demonstrated in the listings:

```
[
  {
    "Plan": {
      "Node Type": "Aggregate",
      "Output": [
        "sum((lineitem.l_extendedprice
              * ('1'::numeric - lineitem.l_discount)))"
      ],
      "Plans": [
        {
          "Node Type": "Merge Join",
          "Merge Cond": "(lineitem.l_partkey = part.p_partkey)",
          "Join Filter": "...",
          "Plans": [
            {
              "Node Type": "Sort",
              "Sort Key": [
                "lineitem.l_partkey"
              ],
              "Plans": [
                {
                  "Node Type": "Seq Scan",
                  "Relation Name": "lineitem",
                  "Filter": "((lineitem.l_shipmode = ANY ('{AIR,\"
   AIR REG\"}'::bpchar[])) AND (lineitem.l_shipinstruct = 'DELIVER
   IN PERSON'::bpchar) AND (((lineitem.l_quantity >= '1'::numeric)
   AND (lineitem.l_quantity <= '11'::numeric)) OR ((lineitem.
   l_quantity >= '10'::numeric) AND (lineitem.l_quantity <= '20'::
   numeric)) OR ((lineitem.l_quantity >= '20'::numeric) AND (
   lineitem.l_quantity <= '30'::numeric))))"
                }
              ]
            },
```

```
            {
              "Node Type": "Sort",
              "Sort Key": [
                "part.p_partkey"
              ],
              "Plans": [
                {
                  "Node Type": "Seq Scan",
                  "Relation Name": "part",
                  "Filter": "..."
                }
              ]
            }
          ]
        }
      ]
    }
  }
]
```

Listing 4.4: PostgreSQL EXPLAIN output

```
Output[revenue]
  revenue := sum
  HashAggregate(FINAL)
    sum := sum(sum_5)
    LocalExchange[SINGLE] ()
      RemoteExchange[GATHER]
        HashAggregate(PARTIAL)
          sum_5 := sum(expr)
          Project[]
            expr := (extendedprice) * ((DOUBLE 1.0) - (discount))
            InnerJoin[(""partkey"" = ""partkey_0"") AND ...]
                     [$hashvalue, $hashvalue_6]
              ScanFilterProject[table = tpch:lineitem:sf1.0,
                                filterPredicate = ...]
                $hashvalue := ...
              LocalExchange[HASH][$hashvalue_6] (""partkey_0"")
                RemoteExchange[REPLICATE]
                  ScanProject[table = tpch:part:sf1.0,
                                      pushdownFilters = ...]
                    $hashvalue_8 := ...
```

Listing 4.5: Trino EXPLAIN output

However, these plans need more information about where each column of data should be placed on the datapath, and additional processing is required to extract constant values used for initialising the modules, including converting the filter requirements into DNF clauses. Therefore to know where each data should be placed while accepting query plans from multiple potential DBMS systems, we created an API where each function stores different relations between nodes and columns. Then the flow involves giving an EXPLAIN command output to a DBMS-specific parser that automatically calls the appropriate general-purpose API function, as shown in the following hardcoded example Listings written in C++:

```
void Example::CreateQ19TPCH(SQLQueryCreator& sql_creator) {
  auto lineitem_table =
      CreateLineitemTable(&sql_creator, 6001215, "lineitem1.csv");
  auto first_filter = sql_creator.RegisterFilter(lineitem_table);
  ConfigureQ19FirstFilter(sql_creator, first_filter);
  auto part_table =
      CreatePartTable(&sql_creator, 200000, "part1.csv");
  auto join = sql_creator.RegisterJoin(first_filter, "L_PARTKEY",
                                       part_table, "P_PARTKEY");
  auto second_filter = sql_creator.RegisterFilter(join);
  ConfigureQ19SecondFilter(sql_creator, second_filter);
  auto addition =
      sql_creator.RegisterAddition(second_filter, "L_DISCOUNT",
                                   true, 1);
  auto multiplication = sql_creator.RegisterMultiplication(
      addition, "L_EXTENDEDPRICE", "L_DISCOUNT", "TEMP_MUL");
  sql_creator.RegisterAggregation(multiplication, "TEMP_MUL");
}
```

Listing 4.6: Building Q19 execution plan

```
auto Example::CreatePartTable(SQLQueryCreator* sql_creator,
                              int row_count, std::string filename)
                              -> std::string {
  std::vector<TableColumn> columns;
  columns.emplace_back(ColumnDataType::kInteger, 1, "P_PARTKEY");
  columns.emplace_back(ColumnDataType::kVarchar, 55, "P_NAME");
  columns.emplace_back(ColumnDataType::kVarchar, 25, "P_MFGR");
  columns.emplace_back(ColumnDataType::kVarchar, 10, "P_BRAND");
  columns.emplace_back(ColumnDataType::kVarchar, 25, "P_TYPE");
  columns.emplace_back(ColumnDataType::kInteger, 1, "P_SIZE");
  columns.emplace_back(ColumnDataType::kVarchar, 10, "P_CONTAINER");
```

```
  columns.emplace_back(ColumnDataType::kDecimal, 1,"P_RETAILPRICE");
  columns.emplace_back(ColumnDataType::kVarchar, 23, "P_COMMENT");
  return sql_creator->RegisterTable(filename, columns, row_count);
}
```

Listing 4.7: Building *part* table definition

```
void Example::ConfigureQ19FirstFilter(SQLQueryCreator& sql_creator,
                                      std::string& first_filter) {
  auto quantity_big = sql_creator.AddDoubleComparison(
      first_filter, "L_QUANTITY",
      CompareFunctions::kLessThanOrEqual, 30.0);
  auto quantity_small = sql_creator.AddDoubleComparison(
      first_filter, "L_QUANTITY",
      CompareFunctions::kGreaterThanOrEqual, 1.0);
  auto ship_air_reg = sql_creator.AddStringComparison(
      first_filter, "L_SHIPMODE",
      CompareFunctions::kEqual, "AIR REG");
  auto ship_air = sql_creator.AddStringComparison(
      first_filter, "L_SHIPMODE",
      CompareFunctions::kEqual, "AIR");
  auto deliver_in_person = sql_creator.AddStringComparison(
      first_filter, "L_SHIPINSTRUCT",
      CompareFunctions::kEqual, "DELIVER IN PERSON");
  auto shipmode = sql_creator.AddOr(first_filter,
                                    {ship_air, ship_air_reg});
  sql_creator.AddAnd(first_filter, {shipmode, deliver_in_person,
                                    quantity_small, quantity_big});
}
```

Listing 4.8: Building initial filter requirements

As a result, we see that the system is provided with all the information regarding the data types of columns and row counts of the tables and table files contained in the table. Subsequently, after all the operation nodes and their dependencies have been supplied, the middleware can efficiently allocate the columns to the specified offsets as demanded by the modules (e.g., the column by which the stream is sorted must be first). Any conflicts can be resolved in time through the use of additional chunks. The Shunting Yard algorithm is utilized to process arithmetic and filtering boolean expressions, producing the expressions in Reverse Polish notation (as explained by Krtolica et al., [160]). This approach solves precedence problems and parses any nested operations, enabling to provide the middleware the necessary operating parameters.

Table 4.8: Plans from PostgreSQL/Trino use CPU-centered optimisations. Nevertheless, when comparing against plans that we created manually, we see comparable performance when executing with TPC-H query 19 at a scale factor of 1 on FPGAs.

PostgreSQL plan execution time	Pruned plan execution time	Additional projections plan execution time
0.277 sec	0.280 sec	0.260 sec

Table 4.8 shows the runtimes of three different query plans when executing TPC-H query 19. The plan generated by PostgreSQL (Trino gives the same plans) is close to being the fastest, faster than the plan with a pruned query plan but slower than the plan with additional projection operations. Although the *part* table filtering node could potentially be removed (as shown in the comparison when executing the pruned plan), the scheduler places it in the first scheduling spin, reducing data sizes early and improving overall performance. Adding additional projection operations could also improve performance (showing there is further optimisation space), but this requires coordination with the scheduler, which is beyond the scope of this thesis. Therefore, we use the query plans given by PostgreSQL for the rest of the evaluation.

4.3.2 Comparison with Static

In order to measure the difference between time spent on configuration, initialisation, scheduling, execution, and any time spent on updating internal data structures between execution runs outside of initialisation and scheduling, we did our measurements after ten hot runs for end-to-end runtimes that exclude copying tables between the SD card and the memory blocks in DDR. As discussed in Section 3.1, memory-aware scheduling and building systems around it is a nuanced enough field that it deserves more attention in its own right and is left out of the scope for this evaluation and, as such, an in-memory system architecture model is assumed here (given the amount of memory available nowadays). Furthermore, for any potential future deployment scenarios that demand high performance, the integration with a DBMS will have to be tighter than querying the EXPLAIN output as is explained in the previous Subsection 4.3.1, and as such, optimising the performance of this preprocessing step was left out of the scope of this evaluation, and the associated runtime is not considered.

Accelerating Q19

As we previously already started looking at Q19, one of the benchmark's most complicated and dynamic queries, we first examine the runtimes associated with executing this query against datasets generated with different SFs in Figure 4.14. What makes Q19 dynamic is that it has a join that requires sorting where both streams are preceded and followed by a filter, and the resulting stream is then used for final arithmetic operations - making the problem sizes unpredictable.

In order to see how the data-dependent filters affect the flow, we need to evaluate runtimes with small datasets that fit into our memory and larger datasets that do not fit into our memory. We confirmed that all modules stream data at the I/O throughput speeds. Therefore, we emulated larger processing workloads by running the data through the pipeline multiple times to get the measured numbers reported in Table 4.9. As a result, we can preliminarily confirm that with large enough datasets, streaming times (marked as execution time in Figure 4.14) start dominating the overall runtimes.

Using static pipelines

Now, to compare the effectiveness of our dynamic approach against a static one with a more dynamic workflow than our image processing example, we also created two static configurations that fit into our PR region to be swapped according to requirements (again, the resource requirements of these pipelines are in the Appendix A.2). For fairness, both approaches use the same PR region, which means that our dynamic reconfigurable system and the static reference variant have the same number of resources available. While there exists a theoretical third approach where the static shell used for communication with the PS and main memory is synthesized together with the static pipeline, we've chosen not to compare against it in this thesis. Our focus is on contrasting the dynamic pipeline approach with a static PR "island" approach, sidestepping the optimization nuances of the surrounding static shell, as it falls outside our primary objective of evaluating the middleware abstraction layer. The two configurations are designed to require as few reconfigurations as possible to minimize the number of reconfiguration steps for the queries we are using for evaluation (Q6, Q14, and Q19):

- Filter; +/-; Multiplier; Global Aggregation; Linear Sort
- Merge Sort; Merge Join; Filter; Global Aggregation

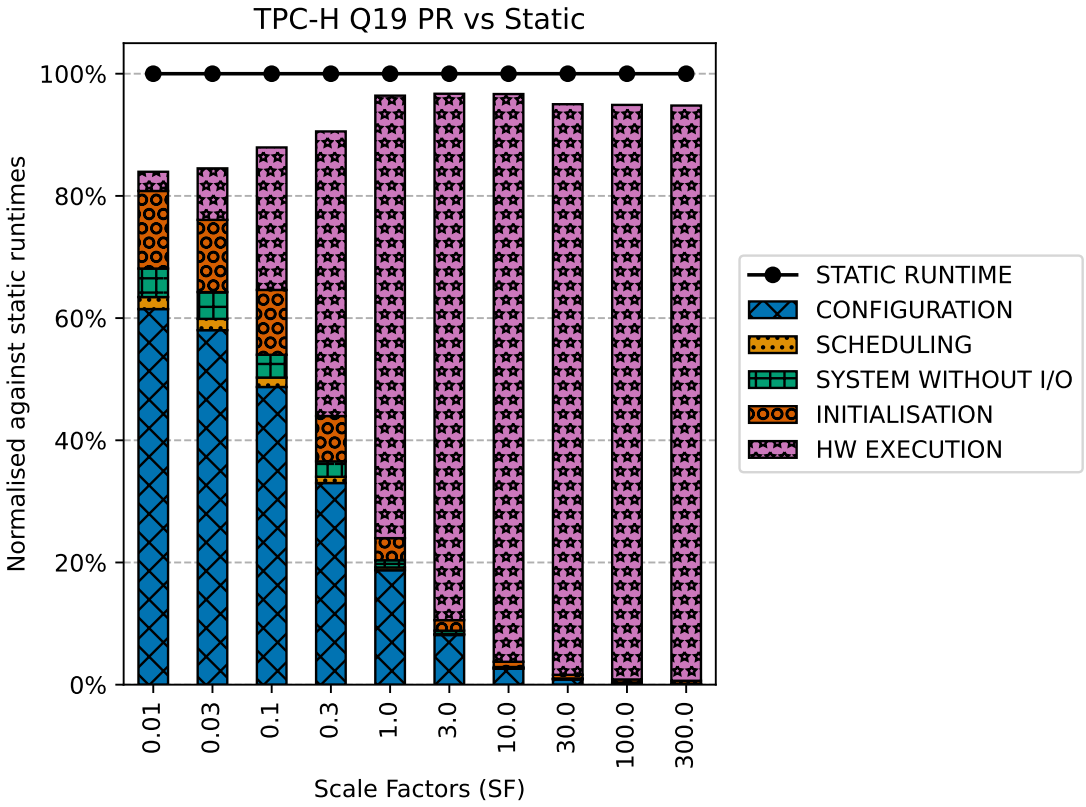

Figure 4.14: The runtime of TPC-H Q19 scales sublinearly as the configuration and scheduling overheads get amortized with large problems. Due to limited DDR size, SF=1 input data is recycled to verify runtimes of large SF workloads. The static approach underperforms the dynamic approach, even with smaller datasets it was designed for. Mainly because the entire static pipeline must be loaded and configured, unlike the dynamic approach which can selectively configure resources as needed.

Instead of scheduling and handling various modules individually, the two static configurations are switched as a whole, requiring less scheduling while having a negligible system overhead. Because of the lack of fragmentation issues that would otherwise appear with synthesizing modules separately into bounding boxes, the static configurations can fit one extra module into the PR region, which is impossible with our current PR module library. Nevertheless, due to needing to reconfigure a larger area with full static reconfiguration cycles, the configuration cost was larger than the configuration cost of the more fine-grain PR approach, as we can see from Table 4.10[8].

[8]The reconfiguration overhead has increased when comparing these numbers with the results in [200]. This difference is due to writing blank bitstreams before each module bitstream for increased reliability and removing intermittent bugs in our system. As a result, each bitstream is twice the size (due to combining module bitstreams with the blanking instructions), affecting larger bitstreams more - as a result, making the static workloads slower. This blanking can be potentially optimised with bitstream

When comparing the static and dynamic approaches, we can confirm further assumptions, such as, for small problem sizes, all overheads constitute a substantial part of the whole runtime duration. The near-constant configuration overheads entail that we can expect better performance benefits when scaling up the system as the configuration costs become proportionally neglectable.[9] As a result, we can confirm from the runtime values from Table 4.11 a speedup of 1.05× over our static pipelines with the SF 300-sized data set. Since query 19 contains many filtering operations, there may not be much room for the resource elastic sorter to improve performance (as shown in the next experiment).

From Figure 4.14 we can see that the speedup can fluctuate when scaling up as there are scale factors when the static approach would be again more preferable to the dynamic approach. The accelerator pipeline created with PR modules and the static system have different utilities for merge sorting, and this causes jumps, where the runtime duration gets slower due to needing another major run (appearing at different SFs for the PR and static variants). In other words, there are situations where substantially more reconfiguration steps are required when scaling up due to not having the ideal combinations of modules fit the PR region at that specific problem size which can point to a lack of module variants or alternatives in the module library. However, to gain even more speedup consistently, in our next experiment, we alter our workload to have a more significant time spent on an operation that our FPGA excels at accelerating.

TPC-H Q19 without filtering

With standard TPC-H data, the filter and the projection in Q19 remove 98% of the data, drastically reducing the accelerated problem size while showing that we spend a large proportion of processing time moving immediately discarded data to the FPGA. If we use data that passes all of the filters instead, we have to sort 50 times more data, which is the cause for additional speedup over the static pipelines', as shown in Figure 4.15. Instead of 1.05× speedup, we now achieve a 1.61× speedup for SF 300, as shown in Table 4.14. This difference would be even more prominent with an even more improved compute-to-memory ratio in a column-store database where the projection operations are not required (these remove more than a third of the data as

compression, but this is out of the scope here.

[9] Using partial reconfiguration to set up pipelines dynamically can be compared with a buffet restaurant - the increased flexibility of what customers can consume is at first more expensive; however, as customers consume more, the amount they invested initially brings more value in return.

Table 4.9: TPC-H query 19 runtimes of our dynamic reconfiguration approach in seconds.

Scale Factor	Config	Scheduling	Init	System	Execution	Sum
0.01	0.044	0.0014	0.0033	0.0090	0.0022	0.060
0.03	0.044	0.0014	0.0033	0.0090	0.0064	0.064
0.1	0.044	0.0014	0.0033	0.0095	0.0209	0.079
0.3	0.044	0.0014	0.0034	0.0099	0.0625	0.121
1.0	0.054	0.0013	0.0034	0.0104	0.2079	0.277
3.0	0.059	0.0014	0.0038	0.0125	0.6281	0.705
10.0	0.059	0.0016	0.0048	0.0194	2.1032	2.188
30.0	0.059	0.0021	0.0072	0.0405	6.3207	6.430
100.0	0.059	0.0040	0.0164	0.1118	21.0832	21.275
300.0	0.060	0.0093	0.0435	0.3179	63.2434	63.674

Table 4.10: TPC-H query 19 runtimes of our static apprach in seconds. The static solution swaps between two static configurations in a region.

Scale Factor	Config	Scheduling	Init	System	Execution	Sum
0.01	0.051	0.0011	0.0030	0.013	0.0022	0.071
0.03	0.052	0.0011	0.0030	0.014	0.0064	0.076
0.1	0.051	0.0011	0.0030	0.013	0.0210	0.090
0.3	0.052	0.0012	0.0032	0.015	0.0627	0.134
1.0	0.051	0.0011	0.0034	0.021	0.2105	0.287
3.0	0.052	0.0013	0.0042	0.039	0.6328	0.729
10.0	0.052	0.0017	0.0069	0.093	2.1097	2.263
30.0	0.052	0.0029	0.0148	0.323	6.3754	6.768
100.0	0.052	0.0070	0.0442	0.948	21.3679	22.419
300.0	0.052	0.0180	0.1276	2.780	64.1963	67.174

Table 4.11: Overall runtime speedup improves while using dynamically configured pipelines with larger SF running TPC-H Q19

SF	0.01	0.03	0.1	0.3	1	3	10	30	100	300
Speedup	1.19	1.18	1.14	1.1	1.04	1.03	1.03	1.05	1.05	1.05

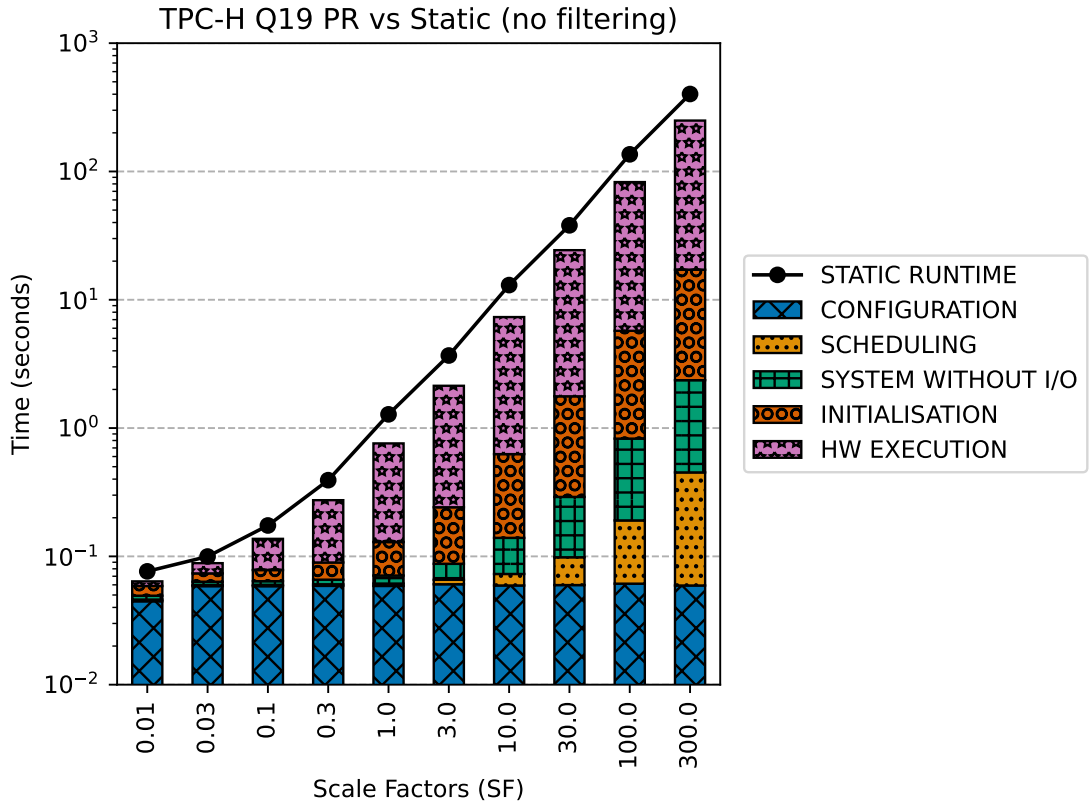

Figure 4.15: With data that passes the *lineitem* filters, even with a logarithmic scale, we see that the data streaming time dominates the total runtime even more. Without filtering the 300GB dataset size test case, the sorting requires four major runs with the dynamic approach, whereas the static approach requires five major runs.

most columns in the TPC-H tables are not required).

TPC-H Q19 with cached tables

In order to also see how our PR approach compares against our static approach in a column-store DBMS, we removed unused columns from the input tables and stored them as "cached" tables. There are additional encoding and data formatting operations in a columnar DBMS that are missing in this experiment, and therefore we can only get an estimation from the results presented in Figure 4.16. Nevertheless, we see that our assumption of increased speedup stands true in this experiment (speedup of 1.12× instead of 1.05×), where we again use standard filtered data but omit unused columns from the I/O operations as seen from Tables 4.15, 4.16 and 4.17.

Table 4.12: TPC-H query 19 runtimes of our dynamic reconfiguration approach in seconds without the initial *lineitem* filtering.

Scale Factor	Config	Scheduling	Init	System	Execution	Sum
0.01	0.045	0.0014	0.0033	0.0094	0.0051	0.064
0.03	0.059	0.0013	0.0035	0.0103	0.0149	0.089
0.1	0.059	0.0015	0.0040	0.0142	0.0588	0.137
0.3	0.059	0.0017	0.0052	0.0238	0.1845	0.274
1.0	0.059	0.0026	0.0095	0.0584	0.6302	0.759
3.0	0.061	0.0050	0.0219	0.1539	1.8949	2.136
10.0	0.059	0.0136	0.0664	0.4873	6.7052	7.332
30.0	0.060	0.0387	0.1922	1.4780	22.5860	24.355
100.0	0.061	0.1293	0.6370	4.9014	76.9255	82.654
300.0	0.059	0.3919	1.9098	14.7805	232.1328	249.274

Table 4.13: TPC-H query 19 runtimes of our static apprach in seconds without the initial *lineitem* filtering.

Scale Factor	Config	Scheduling	Init	System	Execution	Sum
0.01	0.052	0.0011	0.0032	0.014	0.0057	0.076
0.03	0.052	0.0013	0.0036	0.025	0.0186	0.100
0.1	0.052	0.0013	0.0049	0.053	0.0633	0.174
0.3	0.052	0.0019	0.0086	0.139	0.1909	0.393
1.0	0.052	0.0038	0.0221	0.469	0.7321	1.279
3.0	0.052	0.0091	0.0612	1.259	2.2970	3.678
10.0	0.053	0.0279	0.1986	4.967	7.7692	13.015
30.0	0.053	0.0857	0.5995	12.769	24.5309	38.037
100.0	0.053	0.2869	1.9967	44.401	89.2648	136.002
300.0	0.053	0.8517	5.9868	120.459	274.2831	401.633

Table 4.14: Overall runtime speedup improves more while using dynamically configured pipelines with each SF running TPC-H Q19 without filtering over a static solution.

SF	0.01	0.03	0.1	0.3	1	3	10	30	100	300
Speedup	1.2	1.13	1.27	1.43	1.68	1.72	1.78	1.56	1.65	1.61

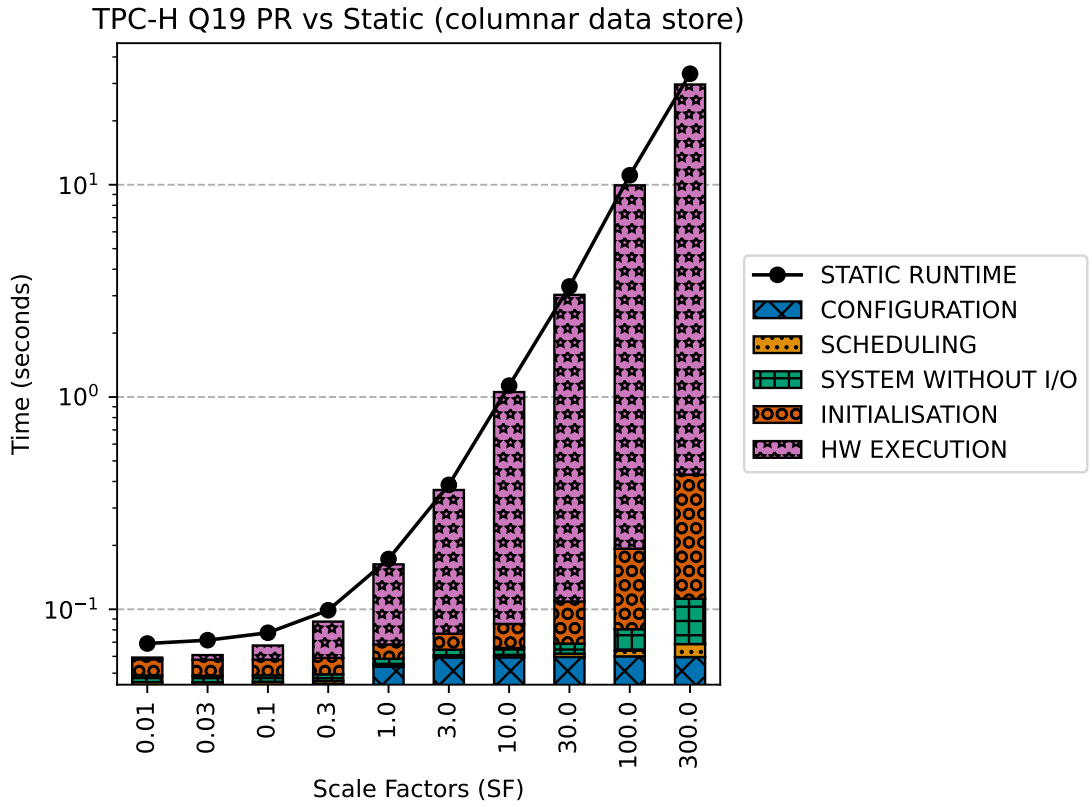

Figure 4.16: TPC-H Q19 runtimes with cached data containing only required columns presented in a logarithmic scale.

Table 4.15: TPC-H query 19 runtimes of our dynamic reconfiguration approach in seconds with only necessary columns.

Scale Factor	Config	Scheduling	Init	System	Execution	Sum
0.01	0.044	0.0014	0.0034	0.0094	0.0011	0.059
0.03	0.044	0.0014	0.0033	0.0092	0.0030	0.061
0.1	0.044	0.0014	0.0033	0.0090	0.0096	0.068
0.3	0.045	0.0014	0.0033	0.0097	0.0285	0.088
1.0	0.054	0.0013	0.0034	0.0098	0.0946	0.163
3.0	0.059	0.0014	0.0038	0.0122	0.2880	0.365
10.0	0.059	0.0016	0.0047	0.0196	0.9701	1.055
30.0	0.060	0.0021	0.0071	0.0399	2.9200	3.029
100.0	0.060	0.0039	0.0166	0.1124	9.7473	9.940
300.0	0.060	0.0090	0.0435	0.3185	29.2424	29.673

Table 4.16: TPC-H query 19 runtimes of our static apprach in seconds with only necessary columns.

Scale Factor	Config	Scheduling	Init	System	Execution	Sum
0.01	0.051	0.0011	0.0030	0.012	0.0011	0.069
0.03	0.052	0.0011	0.0030	0.013	0.0030	0.072
0.1	0.051	0.0012	0.0030	0.012	0.0096	0.078
0.3	0.052	0.0011	0.0032	0.014	0.0286	0.099
1.0	0.052	0.0011	0.0034	0.019	0.0969	0.173
3.0	0.052	0.0013	0.0042	0.036	0.2922	0.386
10.0	0.052	0.0017	0.0070	0.096	0.9749	1.132
30.0	0.052	0.0028	0.0150	0.278	2.9696	3.317
100.0	0.052	0.0066	0.0446	0.961	10.0157	11.080
300.0	0.052	0.0194	0.1288	2.997	30.1432	33.341

Table 4.17: The overall runtime speedup also improves while using dynamically configured pipelines with each SF running TPC-H Q19 with only necessary columns over a static solution.

SF	0.01	0.03	0.1	0.3	1	3	10	30	100	300
Speedup	1.16	1.18	1.15	1.13	1.06	1.06	1.07	1.1	1.11	1.12

Simultaneous queries

Now, when looking at other queries (Q6, Q14), we see a similar distribution of runtimes between the different system processes as we saw with Q19 (Figure 4.17). Just like Q19, the other two queries start with a heavy filtering operation, and as such, with standard TPC-H benchmark data, the number of rows that reach the end of the pipeline is insignificant. Q6 is least suited to our approach as the query is simple and does not benefit from our approach as the multiplication and aggregation functions are not resource-elastic and lack join and sorting operations. Q14 does need a filter before a sort and a join which makes it very similar to Q19. Just the filter requirements are modest for Q14.

However, we can still execute these three queries faster than the static approach (without altering the data) if we make the acceleration request more complex such that the scheduler has more choices to get additional advantages from using PR over static

Table 4.18: TPC-H benchmark runtimes in seconds while running Q6, Q14, and Q19. Our system's runtimes are compared to a static solution's combined runtimes. In addition, the separate runtimes have been summed together for additional comparison.

SF	Dynamic Combined	Static Combined	Speedup Combined	Dynamic Separate	Static Separate	Speedup Separate
1	0.78	0.81	1.05X	0.75	0.78	1.04X
10	6.46	6.62	1.03X	6.27	6.41	1.02X
100	61.69	63.32	1.03X	61.43	63.34	1.03X

solutions. Using the inherent parallelism of the FPGAs that comes from dataflow processing, we can process all three queries simultaneously where the dynamic approach is faster as the static system in all cases, as shown in Table 4.18. As a comparison, executing all queries individually using our dynamic approach provides a slightly lower speedup compared to executing them concurrently (due to higher configuration overhead).

4.3.3 Comparison with State-of-the-Art

As this is not a complete product ready to compete in the very competitive DBMS query analytics engine market due to the lack of functionality to even process all of the queries in the TPC-H (or TPC-DS) benchmark, amongst other standard DBMS features (security, transactional operations), it is not easy to draw fair comparisons with other DBMS engines. Furthermore, as the functionality it does have is not fully optimised or scaled to use the whole board either, it is difficult to compare with other similar FPGA accelerators as they also employ only a fraction of all possible optimisation opportunities (e.g., implementations on larger boards with faster memory interfaces and faster and wider datapaths). Nevertheless, we present a few example comparisons with the current working state of our system against performance numbers provided from related work for reference to estimate the effectiveness of our proposed system with no intention to discredit any related work.

Comparison with related work on FPGAs

Most related work uses column-based storage, as from a performance perspective in purely OLAP workloads, it helps a lot by removing redundant I/O operations. Consequently, related work on FPGAs is mainly using data from column-store databases;

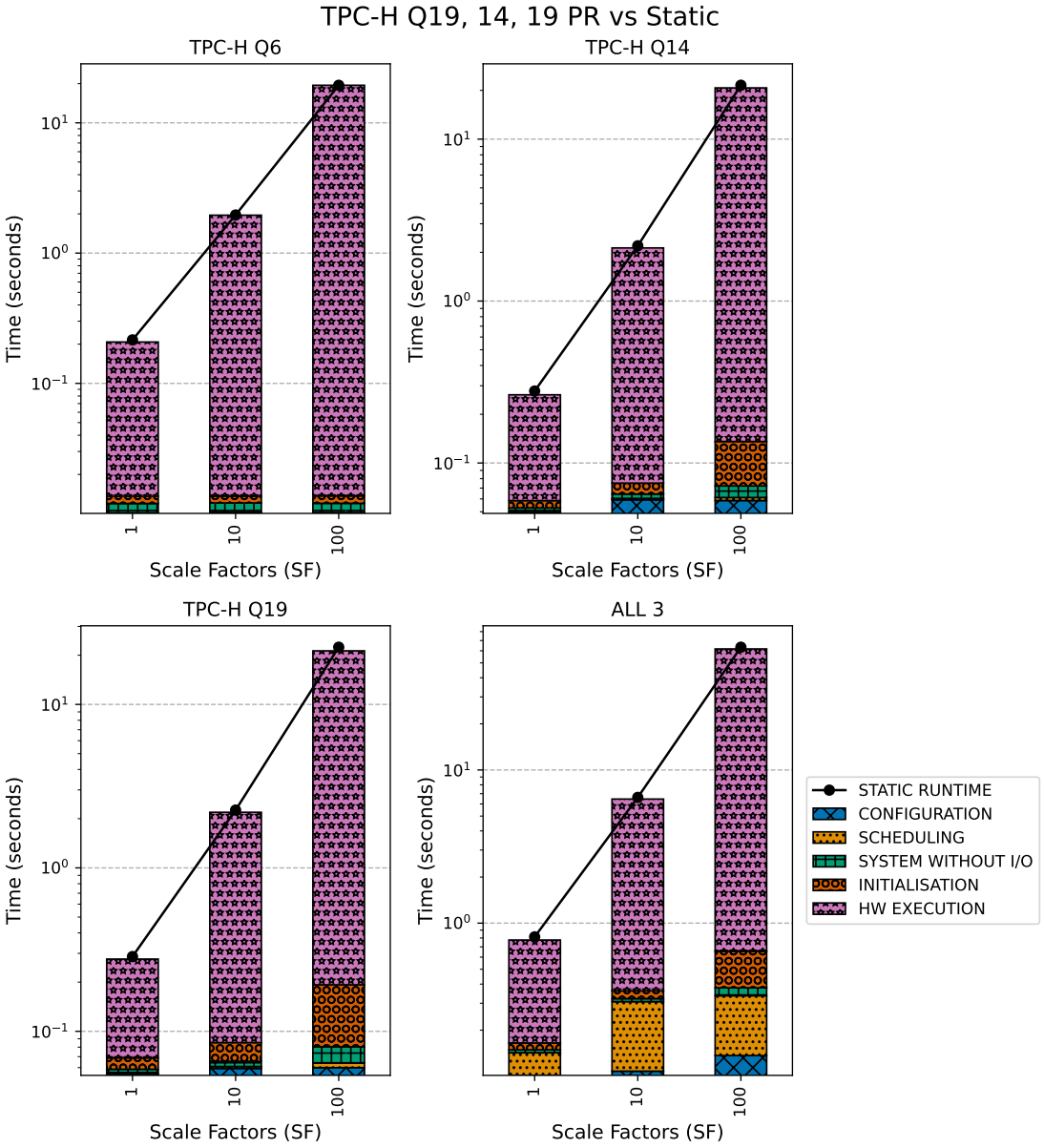

Figure 4.17: The execution time dominates all queries. However, while executing all queries simultaneously, many more modules can be reused to save on configuration costs with minimal scheduling time increase to gain a further advantage ahead of a static solution.

for example, if we look at a similar PR system proposed by Ziener et al., [348] that achieves a peak throughput of 1.7 GB/s through a single region (512bit wide datapath processing at 125MHz), while the system contains four regions in total. As our peak throughput for TPC-H queries is higher (4.7 GB/s) due to the increased clock frequency (300MHz), their system would still be faster in occasions where they can use

all four regions in parallel to achieve a throughput speed of 6.8 GB/s. However, their regions are connected through an immovable join operator, which is more inflexible as queries with multiple joins (more common in TPC-DS) would have to be streamed multiple times. Furthermore, given fast enough memory interfaces, our modules could process data at 19.2 GB/s without changing any implementation details - showing how wide datapath processing at high speeds is an advantage on FPGAs that facilitate good scalability.

When comparing against static FPGA systems, we can consider Xilinx's open-source HLS solution, which can achieve up to 4.14 GB/s when executing TPC-H queries (as reported on GitHub[329]). While these systems are easier to implement with all required functionality, the performance of these implementations can degrade significantly for some operations (for example, anti-join can drop to 203.02 MB/s) while also consuming more resources. Xilinx's results are demonstrated with a few static configurations on an Alveo U280 card, which has 1M logic cells. However, in order to support all TPC-H queries, their implementation consists of multiple expensive configurations running on two Alveo U280 cards, with data directed to the correct PE based on requirements. This approach necessitates significant overprovisioning, as much of the FPGA area is left unused due to the need to accommodate all queries in the benchmark.

Comparison with CPU

When we want to compare against industry-leading DBMS solutions, we have to look at systems running on CPUs. Here, the same story is still true: column-based DBMS have a significant advantage in OLAP workloads, and as such, any column-based DBMS will be faster than our system while processing TPC-H queries. Therefore we look at one of the world's most popular open-source DBMS, which also coincidentally accesses data in a row-based manner - PostgreSQL. However still, the hardware it runs on and the configuration it is set up with (and even what kind of optimisations the SW uses while processing different SF-sized data sets) is so variable that it makes it difficult to draw up fair comparisons between our FPGA based system and any DBMS running on a desktop system (or even a server-grade system).

We compared the performance of PostgreSQL (version 10.18, using default configuration) running TPC-H queries on our scheduling simulation system from Section 4.1.3. We used this comparison as a reference point. The comparison here is also similar to comparing ourselves against our static approach, as with smaller datasets,

Table 4.19: Heterogeneous HW shows better acceleration benefits with larger datasets as expected when comparing Q19 runtimes against PostgreSQL execution times.

SF	0.01	0.03	0.1	0.3	1	3	10	30
FPGA	0.060	0.064	0.079	0.121	0.277	0.705	2.188	6.430
PostgreSQL	0.022	0.031	0.072	0.188	0.586	1.647	5.267	15.775

the gap between FPGA and CPU performance is minimal as seen in Table 4.19. However, the gap grows exponentially bigger with larger datasets despite the CPU using more parallel threads to help with the processing as the dataset sizes grow. The PostgreSQL runtimes on the ARM CPU of the FPGA were an order of magnitude worse than the desktop runtimes. Therefore, the dynamic FPGA system was able to provide transparent acceleration for queries with supported operations, while falling back on PostgreSQL execution for all other queries.

4.4 Chapter Conclusion

In conclusion, in this chapter, we examined the efficacy of the proposed system and assured that it is viable to use in practice.

More explicitly, this chapter proposed a system to create, manage, and run optimised data flow-oriented acceleration pipelines on FPGAs for complex problems like database query processing directly under the control of a runtime system such that arbitrary SQL queries can be automatically executed on an FPGA. We designed a dynamic stream processing platform where modules are picked automatically from a pre-synthesised library by a scheduler while performing optimisations to improve FPGA utilisation and performance (by reducing the number of runs through the FPGA and by reducing the configuration time). Illustratively, the scheduler could be considered a JIT query compiler as it translates query nodes into stream processing pipelines at runtime.

We compared the performance of our dynamic system to a static system with the same resource budget and observed consistent performance improvements under *favourable* conditions. However, such favourable conditions are limited to a narrow area in the problem space when sorting by complexity. It is important to note that the static alternatives were optimised for specific use cases ahead of time, while the dynamic system executed ad-hoc queries without using prior knowledge. On one end, our

approach suits dynamic problems that require HW reconfiguration frequently enough such that even despite the overhead costs, the alternative, more specialised static solutions (e.g., ASICs) often become too ineffective or expensive in comparison. On the other end, these same problems must still be static enough such that the required HW reconfiguration steps are limited enough to avoid slowing down the overall performance with frequent reconfiguration steps while providing enough acceleration over general-purpose devices (e.g., CPUs or GPUs). All of the overhead costs can be optimised to an extent such that these processes can take a negligible amount of time compared to the overall end-to-end processing runtimes, and consequently, this *favourable* conditions area becomes wider. For instance, we list some possible further optimisation options here (other than HW or SW quality improvements) for each aspect of the associated overheads:

- **Configuration** - Implementing ICAP use, allocating more resources to PR regions or allocating some modules to the static partition of the system while tolerating overprovisioning, or changing scheduling priorities to find scheduling plans with reduced configuration costs.
- **Initialisation** - Use more specialised modules that require fewer register writes.
- **Scheduling** - More specialised heuristics that, for example, can use historical data for enabling stochastic and meta-heuristic approaches that can also be accelerated by the FPGA if required.
- **Parsing** - Tighter integration with any front-end parsing applications with more limited flexibility.

Therefore we have successfully shown the validity of this idea from the performance point of view, as the dynamic dataflow approach can overcome any comparative static approach's performance for specific problems like data analytics. Furthermore, due to the flexibility of this approach, the system can still use the static approach if it turns out to be beneficial, as multiple modules can be combined to act as a single pipeline unit if the added bitstream cost is worth it in the targeted application workloads.

Another consideration is the approach of statically synthesising the entire FPGA without distinguishing between shell and role areas. While this method might yield better performance by eliminating fragmentation-induced issues and maximising resource availability for the accelerators, it has its drawbacks. If there are not enough resources available for all of the desired operations, the reconfiguration overheads can

escalate as the whole board needs be reconfigured (including the infrastructure logic responsible for the communication between the accelerators and the PS). At that point it can be more practical to utilise multiple boards, as evidenced by the evaluation of the Vitis library [329].

Nevertheless, performance is not the only part of the system that should be examined - **maintainability** is a core concern for anyone adapting new technologies or methodologies. With the added flexibility of our system, we also introduce an overhead in complexity which we will address in the next chapter.

Chapter 5

Generalisation

Lastly, this thesis highlights an often omitted non-functional requirement: maintainability. After examining 1) the current state of virtualised FPGA acceleration (Chapter 2), 2) the core idea behind a runtime-managed FPGA dataflow accelerator (Chapter 3), and 3) the implementation and evaluation details of an example system capable of dynamically building accelerator pipelines for executing data analytics and image processing workloads (Chapter 4), we can concur the idea of dynamic accelerator pipelines has prospects. Nevertheless, continuous effort is needed to make this approach more accessible and effective, which can be done with every new module, scheduling strategy, or supported device added by third parties. Over time it becomes a shared ecosystem through which different aspects of the system can be optimised independently and then combined with ease. Therefore we still need to address generalisation, scalability and security.

In this chapter, we talk about how the middleware orchestrating the dynamic stream processing is generalised through SW engineering practices in Section 5.1. Then we look at how the module library can be optimally expanded in Section 5.2. Finally, we look at how potential third-party modules can be used securely with a virus scanner in Section 5.3.

5.1 System Generalisation

In the previous chapter, we showed a sufficient performance boost when using FPGAs with a dynamic approach that allows constructing dataflow pipelines on the fly. However, we do not want to focus on only providing accelerators for applications that have

workloads that suit the dataflow MoC as we see that the selection range of fitting applications is made narrower by the ease-of-use of more general-purpose devices where the industry is willing to sacrifice performance for faster time-to-market. Given the complexity of the system we are working with, which requires configuring low-level logic gates at the HW level, while simultaneously managing high-level SW requests, we must consider the potential for the system's maintenance to rapidly escalate out of control, resulting in numerous errors across multiple dimensions. Therefore, to facilitate developers' work and enhance their *Quality of Life* (QoL), it is necessary to introduce greater levels of abstraction.

5.1.1 Correlation with Existing Abstraction Systems

First, in this chapter, we look at operating systems (OSs) for insight (albeit tangentially). Historically OSs provide the necessary abstraction to help tie everything (different SW libraries and HW controllers) together in a maintainable way. With the added complexity of HW reconfiguration on FPGAs, there is all the more reason to keep following this idea of achieving maintainability through various abstraction layers, as they can be managed chiefly automatically but also expanded on if necessary. Especially given that one of the most significant hurdles for FPGAs to become ubiquitous is the intricacy of creating efficient systems with current tools.

Similarly to our system, OSs provide the ability to switch out the underlying HW (assuming the new modules work with our existing drivers) or SW (for accelerating new applications) or middleware (e.g., for different scheduling strategies) without breaking the functionality of the other parts. However, it is also possible to perceive this system as not an OS - not more so than the FOS that is often mentioned throughout this thesis or any of the other OS-like systems mentioned in the numerous virtualisation surveys that have surfaced recently ([245, 302, 282, 120, 254]), but more as a JIT compiler. Our system similarly has a usual JIT aspect which changes the bytecode equivalent IR (the HW pipeline of PR modules in our case) depending on the optimisation needed based on any conditions arising during runtime. Nevertheless, when designing such a system, we also see that just like a compiler that uses a lot of different

instructions defined in the available ISA[1] (which can be even extended with more optimised bytecode operations [291]), our system can similarly do a better job with more modules (i.e., instructions from the compiler point of view).

Long story short, following the Linux model is an excellent way to maintain such flexibility to enable adding additional modules. Linux, with its kernel, provides many abstraction layers with various access levels for security, and the mechanisms to cross these abstraction layer boundaries can be provided through optional modules that use the filesystem as a standard interface (e.g., storing configuration variables as files) to make the system as lightweight as required. Consequently, it is omnipresent and provides a fundamental OS skeleton which is then modified through different Linux distribution providers to work from low-cost wearable devices to high-performance servers. For example, Reghenzani et al., [262] survey methodologies and applications that use Linux for its functionality to change its code in real-time for real-time applications on various devices.

5.1.2 Driver Generalisation

Now, this chapter focuses on what is necessary to replicate creating such a base framework from which effective specialised dataflow accelerator generators can be created. For any framework, the main goal is to improve the process of adding new features (HW modules, in our case). As each HW module may be initialised for different use cases, they need corresponding SW logic to run, and therefore we created a modular middleware consisting of drivers that can be added or removed as necessary to support any changes in HW. Each driver ensures that necessary constraints are obeyed while abstracting the low-level implementation details and serving high-level functionality (i.e., setting up high-level filter conditions with low-level initialisation instructions). In practice, these drivers consist of many memory-mapped register writes (where the addresses may be unique to each module) of specific initialisation values (which may be

[1] Instruction set architecture (ISAs) lets SW use HW through instructions. There are many flavours of highly optimised modern ISAs like the following examples: 1) x86 ([65]), 2) ARM ([78]), 3) RISC-V ([316]). Nevertheless, standalone, these ISAs can have undesirable performance as the instructions often invoke simple atomic operations, resulting in repeated memory operations in complex tasks involving loops. Therefore, these ISAs are often extended in high-performance applications where multiple instructions are combined into new instructions (e.g., instead of using multiplication and addition operations, a multiply-accumulate operation achieves the same result faster) that can also call specialised accelerators and reconfigurable HW resources [83]. However, extending ISAs is a vast research field that examines how to integrate these new instructions effectively without excessive bloat or how to design new schedulers that can use these extensions - challenges that we have to examine also for our system.

unique to each module again) that are likely incompatible with other modules. Therefore to avoid limiting any HW-level optimisation opportunities, the set of drivers used for the whole module library can be highly heterogeneous.

As with any highly heterogeneous system - this becomes more difficult to maintain while scaling up. Hence next, we highlight these three crucial driver design principles to manage this rising complexity:

1. Have a common shared interface with the rest of the system.

2. Find common characteristics between different module driver functionalities.

3. Use these characteristics to create optional mechanisms to tell the rest of the system of additional constraints (including constraints about how data is transferred over standard interfaces).

First, as a nonnegotiable requirement, drivers must have a standard interface. In our system, all initialisation values extracted from parsed input acceleration requests get mapped into memory blocks that are then passed between the system and the drivers. These blocks are represented as nested 2D lists - lists of lists (alternatively called a 2D array or a 2D vector in various programming languages). In this way, the driver can request as much data as necessary from the middleware while the data can still be organised into various sublists - just like Linux uses the filesystem as the standard interface. For example, the first list of values in this data structure can inform the driver or the middleware about any specific encodings used in a particular instance, enabling the driver code to check and correct errors.

Second, writing new drivers for any new HW module requires repeated effort. In the following subsection, we examine the common characteristics in our module library where the supporting driver code could be reused with new HW modules.

Last, enumerating all these characteristics allows us to create customised constraint-checking code that can be mapped to each of these indexed characteristics to let the system automatically adapt to any new modules that may use existing characteristics as they require existing condition checks. Moreover, it also creates a modular system where new constraints and HW modules can be easily added to this mapping. Similarly, a Linux OS image works on any HW layout given a corresponding device tree where each node that defines different aspects of the current HW layout must have a value for the *compatible* property to match different existing drivers to their previously mapped HW counterparts.

5.1.3 Driver Characteristics

In order to maintain a manageable complexity of such a system, we can identify different module characteristics that drive how the drivers should be designed:

Register writing: Having a single universal driver with a large number of unused code and a vast degree of customizability is not that practical as there is not a large enough overlap between the different register writes involved with running the modules in our library. Moreover, if we constrain all modules to have a common set of registers (some modules may not use all of these values) and the type of values that go in there (e.g., start, input/output stream IDs, data packet sizes, chunk/channel IDs) there are still a lot of mutually exclusive type of register writes left as some modules need significantly more initialisation data. For instance, our large filter modules need ≈ 4000 register writes to initialise all of the literals consisting of the multiple conjunctions for each DNF clause. The amount of register writes required for initialisation affects setup time, and thus different mechanisms (e.g., warning the scheduler about long initialisation times) might be used depending on how many values have to be written. These different mechanisms will require corresponding driver support that must be custom tailored for each case.

Register reading: However, the runtime system may also read memory-mapped initialisation registers (given the HW support), increasing the interface's capabilities by establishing two-way communications. The read values help with snooping, debugging, monitoring, getting results faster, or triggering early termination with corresponding interruption support. Nevertheless, as with register writing, this can only be generalised to an extent, and with different use cases, custom logic is needed.

Data formatting requirements: As mentioned in earlier chapters, the modules can have requirements on how data is positioned on the datapath wires. For instance, a module could perform a stable sort that only evaluates the first column word values for optimisation and simplification. The same applies to the join module, where the matching key value must be placed in the first column. Moreover, long rows may have to expand in multiple consecutive clock cycles, which may change formatting requirements (e.g., for longer strings). In our system, the data placement can be adjusted with another module capable of shuffling data (e.g., join) or through the crossbar in our DMA module.

Data formatting capabilities: In detail, our join module implementation can shuffle data around in time but not in space (within the same clock cycle), unlike our full crossbars, which can simultaneously permute space and time. This constraint or any other use case details still have to get encapsulated by this characterisation. Especially if more dataflow utility modules are added (e.g., purely to add limited reordering support), there has to be a mechanism to tell this information (i.e., about the extent of any module's data reformatting capabilities) to the scheduler.

Data content requirements: In addition to spatial data formatting requirements, some operations have data content requirements to produce correct results. For instance, the JOIN operator uses a merge join module, so the data has to be presorted. Other cases (like sorting) require operating with hundreds of streams - requiring channel IDs to be provided as well.

Blocking operations: Illustratively, sorting changes the data flow with blocking characteristics where all input records must be seen before outputting any result rows. This characteristic could be used by a scheduler in a different system that is capable of doing PR not only between runs but during a run when waiting before a blocking operation.

Partitioned operations: There exist operations that need to be executed by multiple module types for efficient FPGA acceleration. Such an example is again sorting, which in our HW library is implemented using linear and merge sort stages [151]. Additional scheduling considerations include recognising when a single linear sorter can fully sort a small problem, where the scheduler must omit merge sorting stages.

Tandem requirements: The scheduler also has to consider module-to-module communication. Due to stream formatting constraints, some modules must be placed next to a specific white-listed set of modules (black-listing could also be possible). In our case, the merge sorter has to get its input from the DMA module. The DMA can create multi-channel streams, which the merge sorter uses to merge the virtual streams within the partially sorted stream. Additionally, the Sobel operator module must get input from the Black and White module that transforms the data into having the correct encoding.

System requirements: Lastly, a module can add constraints to the platform running

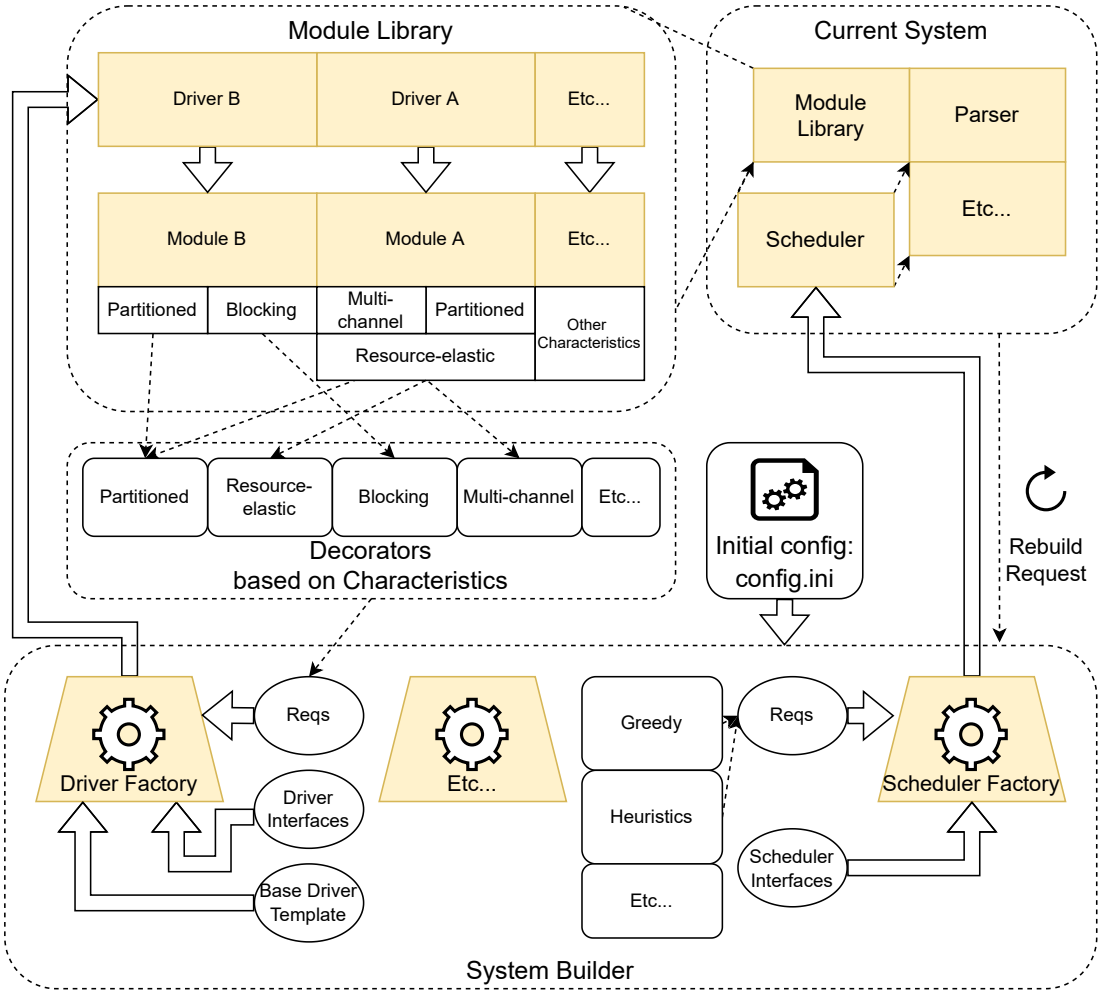

Figure 5.1: Given well-defined interfaces it is possible to dynamically build and later alter not only the HW pipelines but also the various SW system parts.

it. Varying producer and consumer rates and volumes can change memory requirements. Any join module could also increase the stream size. The stream size is likely to increase with full outer joins, but for other types of joins, the system must still be ready to hold the worst-case scenario, which is the combined memory size of all input streams.

5.1.4 Decorator Based Generic Drivers

These common characteristics of how modules must be scheduled, initialised, and used (in addition to managing the resource-elasticity of these modules), require specialised code that can be reused for various system parts. To enable reusing this code, we

use SW design practices like polymorphism[2]. The ideal way is to define expected interfaces between all parts of the middleware and then use *decorators* and *factories* to create parts of the system based on high-level requirements (e.g., informing about additional custom logic and decorators required to support the functionality of a new module).

A fitting use case of this type of system design is demonstrated by how it is possible to build up drivers by listing all the constraint types their supported modules need and then letting the builder build fitting drivers as shown in Figure 5.1. Consequently, each module consists of different decorators which define its behaviours and corner case conditions, and the rest of the system can check for these decorators and use additional constraint checking as necessary. In addition, the diagram indicates that both the module drivers and the entire system are constructed using factory objects in our system builder. These factory objects are responsible for creating instances of the various components that make up the system, all while adhering to the interface constraints we have defined.

The main benefit is that it makes the system more expandable. Instead of compiling the whole system each time a new feature (e.g., a new module) is introduced, the desired feature can be supported by combining existing features with new configuration options (possibly even modified during runtime, given a sufficient amount of existing features). By telling the initial builder to create elements that still obey the same constraint but with different implementations (e.g., for different strategies), you can easily switch up the whole underlying code of the system. Such frameworks have been developed for self-adaptive applications on other devices already, such as the *MUSIC* (Methodology for Ubiquitous Self-adaptive and Interoperable Computing) framework, which offers a systematic methodology for the creation of self-adaptive applications. The framework includes the specification of adaptation logic, the utilization of contextual information, and the deployment and assessment of the application, as thoroughly

[2]Object-oriented programming uses polymorphism, where highly correlating code segments are packaged into classes. Through the inheritance of these classes, we can share existing code with calls to general-purpose functions (parent class) first as a standard interface which is then, as the name suggests, morphed between various implementation options (child classes) to meet the desired requirements. As is the case when creating dynamic HW pipelines, mapping different options through indirection to make such a dynamic SW system work also has a performance penalty. However, modern compilers can offer performance benefits in more complicated code bases, as Demeyer showed while studying C++ applications [67]. In order to maximise the benefits of polymorphism, SW design patterns like decorator-based and factory-based designs have been adopted (described in publications that conclude modularity eases maintainability [317]). Consequently, we can build more complex systems with tools familiar with these patterns, from which the FPGA community should also aim to benefit.

outlined by Hallsteinsen et al. in their publication [102]. These systems also employ a middleware that monitors the context of the executing application and adapts it as required, utilizing components from a component repository.

Such runtime SW reconfiguration was not necessary for our test cases presented in the previous chapter, but we still laid the foundations for these features by defining the necessary interfaces to support expanding our prototype later. A potential future use case would be swapping out different scheduler implementations (e.g., using a meta-heuristic scheduler once enough historical data has been accumulated and swapping back given drastic workload changes).

5.2 Module Library Expansion

One part of the system's maintainability is the supporting SW features providing various QoL improvements that make adding new functionality easier. Moreover, as our system performs better with more complex tasks, we need this maintainability to become more generalised, eventually through third-party contributions. However, we still need to address how to make adding new HW modules easier on the HW side.

When incorporating modules into a library, it is essential to align the provided functionality with anticipated system requests. The highest priority is to provide the missing modules that enable the complete execution of expected requests on the FPGA. Additionally, if feasible, the library should offer module alternatives with varying utility or performance, considering the resource cost relative to the base library ensures a practical selection. Providing as many utility-based alternatives as possible is beneficial. If the library becomes too extensive, less frequently used modules can be pruned later. For modules offering performance variations (e.g., operating at different clock speeds), they should be integrated only if the scheduler can accommodate such variations (as the evaluated scheduler does not take that into consideration because the whole module library operates at the same clock speed).

Another consideration is determining which module variants to include in the library once the desired functionality and performance modules are identified. After building HW modules that meet the targeted dataflow interface and optimising the resource usage, there are still various options for figuring out which Pareto-front resource footprint to use. We use a resource popularity concept to find optimal placement regions with potential resource footprint matches and optimise the module library for the given PR regions in this section.

Another question is how many module alternatives are needed for this approach, and can we have too many modules? With more modules, the scheduling algorithm can do a better job. However, we have two limiting factors: bitstreams take up limited space in memory, and the scheduling will take more time with more extensive module libraries.

Therefore in this section, we analyse this module placement problem and compare the following two approaches with a case study at the end of this section:

1. **Histogram-based approach** - Using modules with resource footprints that can be placed in the largest number of locations compared to other valid footprint options.

2. **Heatmap-based approach** - Using modules that can be placed in the least popular locations.

5.2.1 PR Region Histogram

There are previously explored methods for choosing PR regions (e.g., Becker et al., [22] scan through the fabric with various masks marking suitable areas given matching resource requirements); however, we also want to consider placing multiple modules into a PR region simultaneously next to each other. We can estimate the level of heterogeneity of a configurable region if we look at the resource string. In a configurable region with low heterogeneity, there will be a relatively small number of different types of resources, and additionally, there will be multiple commonly appearing resource patterns (that consist of even more common subpatterns). On the flip side, with high heterogeneity, the number of different resource types is high, and with few infrequent resource types, it is less likely to find resource column patterns appearing more than once. Again, on the ZU9EG, there are three types of resources (M for CLB, D for DSP, and B for BRAM) that all include routing resources, and Figure 5.2 shows a PR region that is well-suited for facilitating dynamic stream processing pipelines due to numerous patterns that appear often. Frequently appearing patterns make it likely that any design placed in one part of the PR region will also fit another part of the region. Therefore, while scanning for PR regions, as described by Becker et al., maximizing the number of patterns is another important objective.

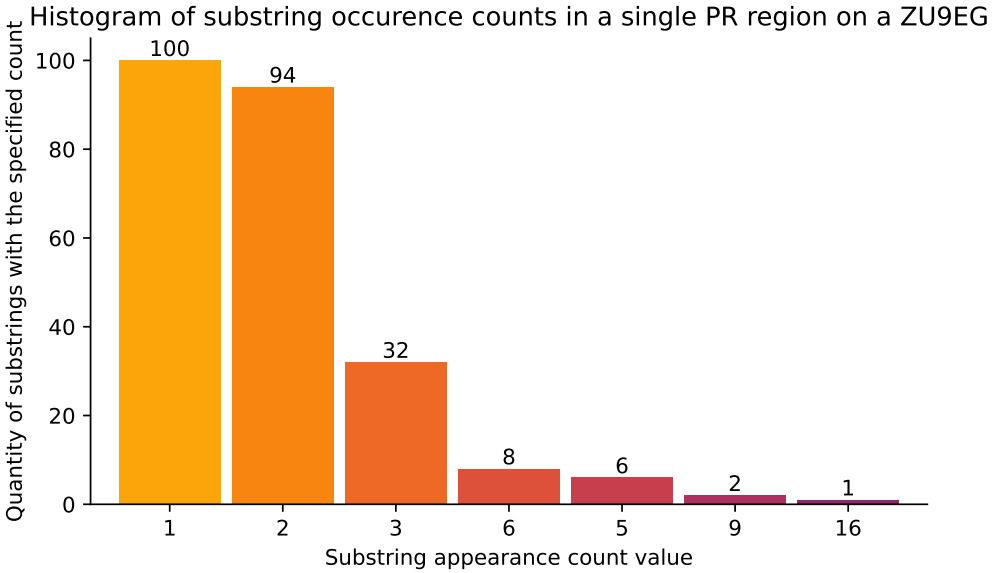

Figure 5.2: How often do all possible substrings appear in a PR region in a ZU9EG? The most common substring in "MMDMDBMMDBMMDMDBMMDBMMDMDB-MMDBM" is "M", appearing 16 times.

5.2.2 HW Module Library Heatmap

Regardless of the heterogeneity of the configurable region, the effectiveness of a scheduler finding valid configurations diminishes with highly contested resource columns by the modules in the HW library. If we have a highly heterogeneous PR region, it is still possible to build an effective system by adding more modules into the library that use the less popular resource column clusterings. Hence, adding even suboptimal module implementations of a particular task that use less popular resource column patterns can improve the system's overall performance.

Nevertheless, the first step to building up a module library is to make all of the modules as small as possible to reduce internal module fragmentation (as was done with the module library presented in the previous chapter). However, after that, we have two different approaches, as mentioned earlier, when expanding the library:

1. Create modules that use subpatterns that appear multiple times in the configurable region after analysing the resources with a subpattern histogram (Figure 5.2).

2. Use the popularity of each resource column to build up a heatmap - then add modules that touch the cold areas.

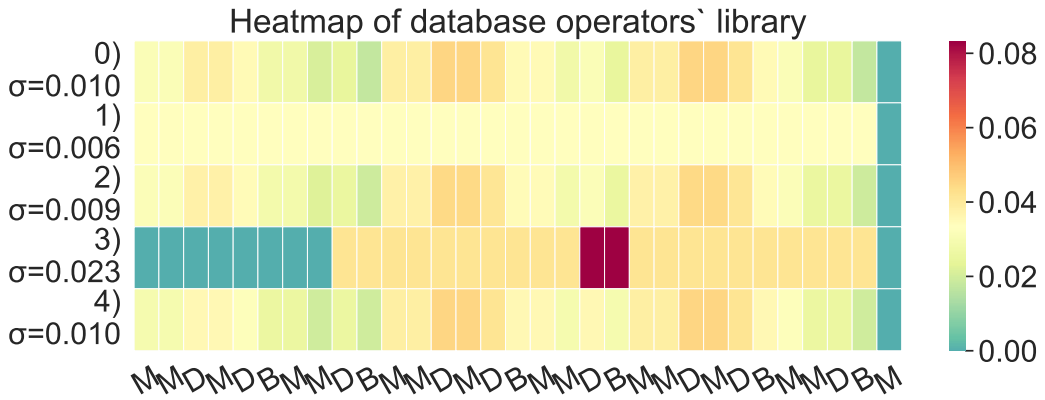

Figure 5.3: **(0)** Original module library from [195] with only one merge sorter variant; **(1)** An additional sorter module that can fit into 3 locations; **(2)** Combination of (0) and (1); **(3)** A bigger merge sorter which can fit only into 2 locations; **(4)** Combination of (0) and (3).

In Figure 5.3, we have a decision when adding a module (either in row 1 or row 3) to the library in row 0. We can see that adding additional modules that cover unpopular resources lowers the standard deviation σ of the normalised popularity values, resulting in a more even distribution. As a side note, the popularity of different resource columns can also be used in scheduling to find valid configurations sooner, as shown in [92]. However, minimising the number of modules is also necessary to reduce scheduling times and the configuration memory footprint of the module library. Therefore in the next subsection, we evaluate whether it is better to use modules that can be placed in multiple places (first strategy) or modules that use less popular resources (second strategy).

5.2.3 Case Study: Database Module Library Expansion

We compare the histogram and heatmap approaches from the previous section in a practical scenario with a module library for database operation acceleration on a ZCU102 board. Our module requirements and the PR region's resource string are based on the values presented by Manev in his thesis [195]. In this comparison, we extended the library with additional alternatives for each previous module in the library. These modules are for sorting, filtering, and joining with multiple variants. First, we found all possible resource footprints with each module's required number of resources. Next, we narrowed down the considered modules to only include the smallest ones possible from the set of modules not included in the library already. Then we chose the best

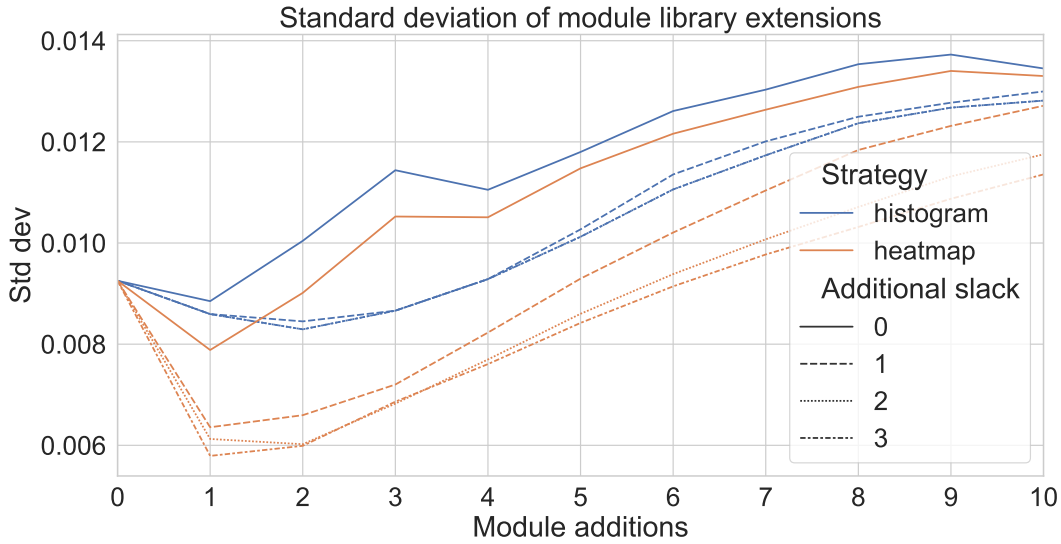

Figure 5.4: In order to cover the configurable region more evenly, choosing larger modules with the heatmap approach leads to best results given that only 1 to 2 modules of each type are added.

module from these alternatives with one of the following strategies. With the histogram strategy, we chose the module with the most placement options. With the heatmap strategy, we evaluated each module variant after extending the library and picked the module with the lowest standard deviation of resource popularity values. Lastly, we experimented with allowing up to three additional resource columns for larger modules (as opposed to using modules with the smallest resource footprint when building up the initial library) to be considered with either strategy. As a result, in Figure 5.4, we can see that as the initial library was already well constructed, adding additional modules was only effective while adding one or two modules. With more modules, the distribution of resource usage got more uneven as the optimal spots were taken.

In Figure 5.5, we can see that while considering larger modules, the one or two additional resource columns do not substantially affect the overall hardware size. Furthermore, we can see that the heatmap approach does pick larger modules more often than the histogram approach. Therefore, not many modules are required to improve the number of placement options of the modules in the HW library if the initial configurable region is with low heterogeneity. However, adding larger and seemingly more nonoptimal modules can improve the likelihood of finding a valid configuration. Using the heatmap approach and considering up to 3 resource columns larger modules, we decrease the standard deviation of the normalised popularity values by 34.6% while only having a 10.3% larger footprint than adding a single module of each type with the

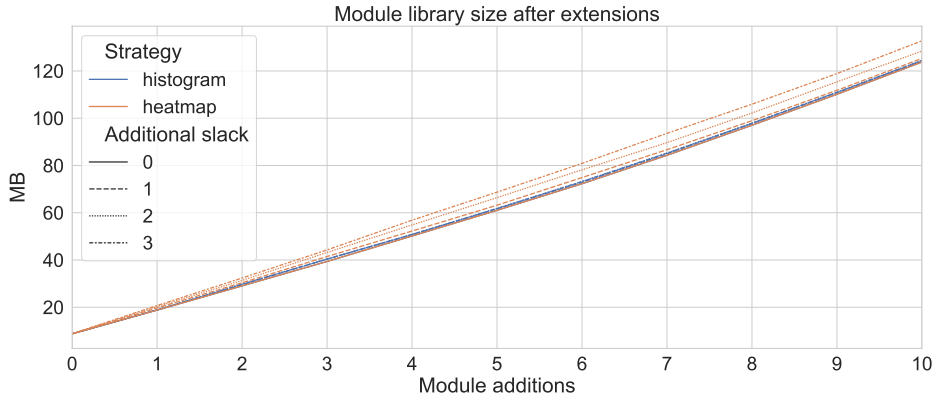

Figure 5.5: The library size is largest when the library gets extended with larger modules using the heatmap approach. The original library is 8.8 MB large, containing 15 modules.

histogram approach while considering only the smallest modules. Nevertheless, the modules cannot be made too large because when the modules' external fragmentation decreases, the internal module fragmentation will start dominating.

After determining the best set of these additional modules, currently, these will likely have to be built manually (involving a degree of trial-and-error) using the vendor tools, but eventually, this process can be fully automated. However, setting up the correct bounding boxes and synthesising modules with the correct amount of slack is challenging to do reliably with the current set of vendor tools available. As such, creating such a tool flow is outside the scope of this thesis due to needing expertise about current tools, which may change in the future.

5.3 Security

Now, after creating the appropriate SW support and presenting approaches for adding additional HW modules while keeping the module library lightweight, we look at the last hurdle that stops dynamic stream processing from becoming more common on FPGAs - creating the supporting ecosystem to improve the system's usability and performance. An effective way to create most ecosystems is to provide a cloud service[3].

[3]There is research interest in evaluating the healthiness of ecosystems as it provides information to newcomers choosing between different options while suggesting how to improve to all existing participants: end-users, developers, and investors. For example, Jansen [126] evaluates ecosystems by measuring the productivity, robustness and niche creation of the base projects themselves and the network around them. As such, cloud service providers make good keystone projects that foster activity due to the low barrier and more importantly, it enables others to create new projects that use this cloud product

Cloud services are especially beneficial when the primary system idea performs better with larger datasets and ad-hoc queries and where the supporting infrastructure can be challenging to build up alone (like our system). FPGA acceleration has been recently used in the cloud often as an FPGA-as-a-Service (FaaS) product with examples from researchers (aside from commercial examples) creating cloud systems using FPGAs standalone [19] or even along other devices [69]. In this case, whenever any product uses third-party contributions and is accessible online - security concerns must be prioritised to protect against malicious actors [251, 322]. Standard security checks on the SW level are needed, like ensuring that the drivers do not support modules accessing and saving data from prohibited streams or memory areas. However, we also need further security checks for the HW level since we accept unknown bitstream contents. As the FPGA device manufacturers do not directly support multi-tenancy with applications built separately from each other, the device tools inevitably miss out on checking some security aspects, for which we demonstrate a virus scanner-like application that can detect malicious design patterns in HW that the tools miss. This vendor-agnostic approach enables the option for adding further security checks independently.

5.3.1 Importance of HW Security

Traditional HW security involves checking for physical access to devices (e.g., manipulating the power supply is an attack vector), however with the recent cloud service providers offering to use FPGAs, there has been significant research done in the hardware security area concerning remote access to the device [3, 129]. Due to their low-level programmability at the hardware level, FPGAs have vulnerabilities unknown to only software-programmable devices, such as CPUs or GPUs. FPGAs allow mounting physical side-channel attacks remotely on hardware that initially required physical access to attack those other devices. For instance, traditionally, the confidentiality of a device can be broken by monitoring power, voltage, temperature, or other system parameters to gain any information that one usually should not have access to, and this can eventually be used to find cryptographic keys with enough data to find strong correlations [259]. Alternatively, system integrity can also be broken by introducing soft errors by manipulating the operation of the power supply [351].

Remote monitoring of temperature and supply voltage can be performed with ring

- possibly even standalone or as a competitor. Consequently, Lucassen et al., [186] specifically evaluate different cloud service providers and argue that active communities make them likely to succeed in the long term.

oscillators (ROs) connected to Time-to-Digital Converters (TDCs) [273] and the power supply could be manipulated with short-circuits [23, 99] or with a sudden creation of excessive dynamic power consumption. Let us look at the simplest form of interruption - service availability. For this, an attacker would draw enough dynamic power to cause a voltage drop, which would crash the FPGA. This attack has been demonstrated in research where only power cycling would bring the system back into service [90]. It should be kept in mind that these attacks have been made using bitstreams generated directly by the FPGA vendor tools without further manipulation.

Another way of disrupting a service running on an FPGA is to make it operate slower and eventually violate timing slack. For example, it is possible to degrade FPGAs with elevated voltage and temperature conditions [288], which can be produced remotely with methods mentioned earlier.

However, creating FaaS prototypes in the cloud and testing their security is out of the scope of this thesis as this is an extensive research field with various well-written works that we have already cited in this section. Therefore in this section, we only highlight two important aspects of security from a high-level point of view for a better understanding of the required system design ideas:

1. **Bitstream-level security** - Detect malicious hardware designs while scanning the bitstreams with a virus scan.

2. **System level security** - Look for malicious driver behaviour and use precautionary execution policies.

5.3.2 Bitstream Based Verification

The traditional ways to defend against any HW attacks involve using ROs, equivalently to creating the attacks themselves [347]. These approaches are based on detecting an attack while monitoring the system, just like an attacker. This monitoring approach means an attack must already occur for detection, so this mitigation strategy only provides partial protection. In addition to monitoring, ROs can also be used to produce more noise [158]. Producing noise around the logic is a known hiding technique [96]. An alternative to hiding is masking, where the functionality of the logic is the same, but the implementation is less intuitive, making reaching any correct correlations again more unlikely [178]. However, all of these security measures require additional HW resources. Therefore, we look at bitstream verification as an alternative that does not use any HW resources. Nevertheless, adding more layers to any system's defence for

better reliability is reasonable, and all of these measures could be used alongside each other if the resource cost is acceptable.

Our system provides acceleration with pre-synthesised implementations and, as mentioned at the beginning of this thesis, such bitstreams come with many benefits, but they also come with security risks, for example, trojan horse attacks [335]. By scanning these bitstreams before even loading them onto the board, we can not only detect or hide from these attacks but also prevent them from happening in the first place. Our approach looks for malicious patterns in the netlist that can be flagged with different levels of severity, where afterwards, the system can decide whether to allow running the design. More details about this virus scanner and the level of accuracy it detects the attacks with are given in our papers ([199, 165]). Nevertheless, this subsection will give a high-level view how these attacks work and how a virus-scanner could be designed to detect these attacks next.

Detecting Attacks while Scanning Graphs

The netlist of a HW design (as illustrated in Figure 5.6) can be visualised as a graph while marking the various resources on the FPGA, like programmable interconnect points (PIPs), as nodes and the wires connecting these various resources as the edges. Constructing such a graph can be done with the following steps:

1. Get the architecture graph of the area we want to scan. The architecture graph consists of nodes and edges that are not programmable.

2. Get the dynamically configured nodes and edges (programmable resources). The dynamically configured wiring and LUT configurations can be retrieved after translating the bitstream values [337, 138].

3. Combine the two graphs and start traversing the resulting graph looking for malicious behaviour.

As these netlists can become very large (in increasingly more heterogenous systems), the runtime of any graph traversal algorithms required to detect these patterns becomes an issue. However, as we are dealing with PR modules only in areas constrained by a bounding box, we can sidestep these runtime problems. Furthermore, these scans can be done offline in our case as we are using pre-synthesised designs as a part of a module library that is likely constructed and optimised before accepting any acceleration requests.

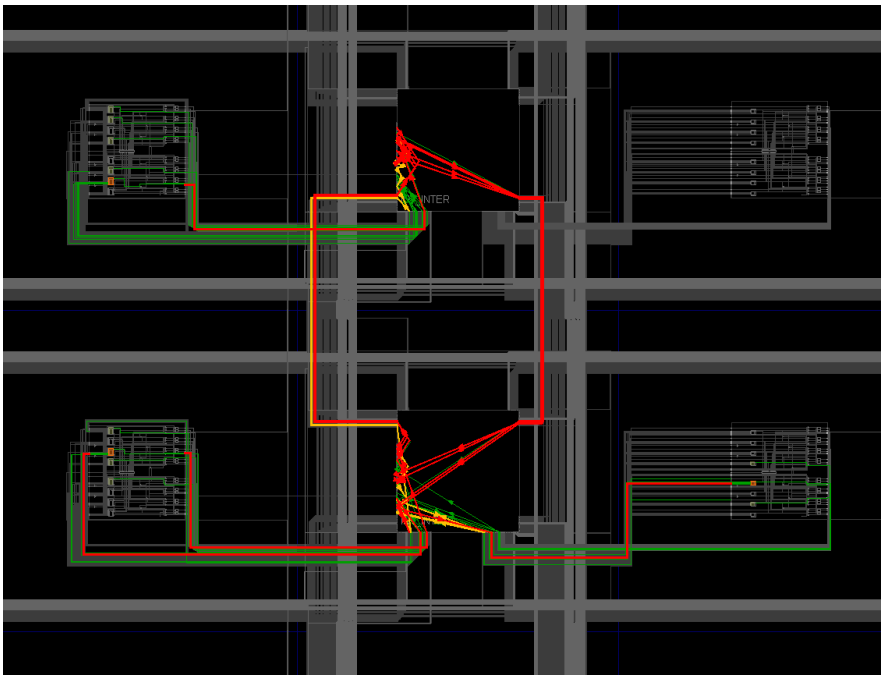

Figure 5.6: A ring oscillator consisting of three inverters (orange LUTs) can be decomposed into both static wires, inherent to the device's architecture, and dynamically configured wires. The design represented by the green highlight also incorporates yellow and red dynamic wires, which depict alternative dynamic configurations to complete the ring oscillator. The virus scanner must be capable of detecting all possible configurations.

Nevertheless, bitstream scanning can be easily improved while it also solves trust issues. The IP provider does not have to trust the platform and vice versa if they both trust the virus scanner, as is shown by Zeitouni et al., [340]. Meanwhile, building the scanner to consist of separate steps supports maintaining each one separately and can also help solve any remaining trust issues if separate trusted parties maintain these: 1) graph building and 2) virus signature scanning. Therefore as more malicious virus signatures are found, more different scans can be added to the scan process just like different boards can be supported by expanding the graph-building capabilities. Next, we will go through two attack examples that can cause high switching frequency and how the virus scanner can detect these.

Ring Oscillators

One of the easiest ways to generate switching activity in the circuitry, which runs faster than the clock frequency, is through ring oscillators. A ring oscillator is a combinatorial

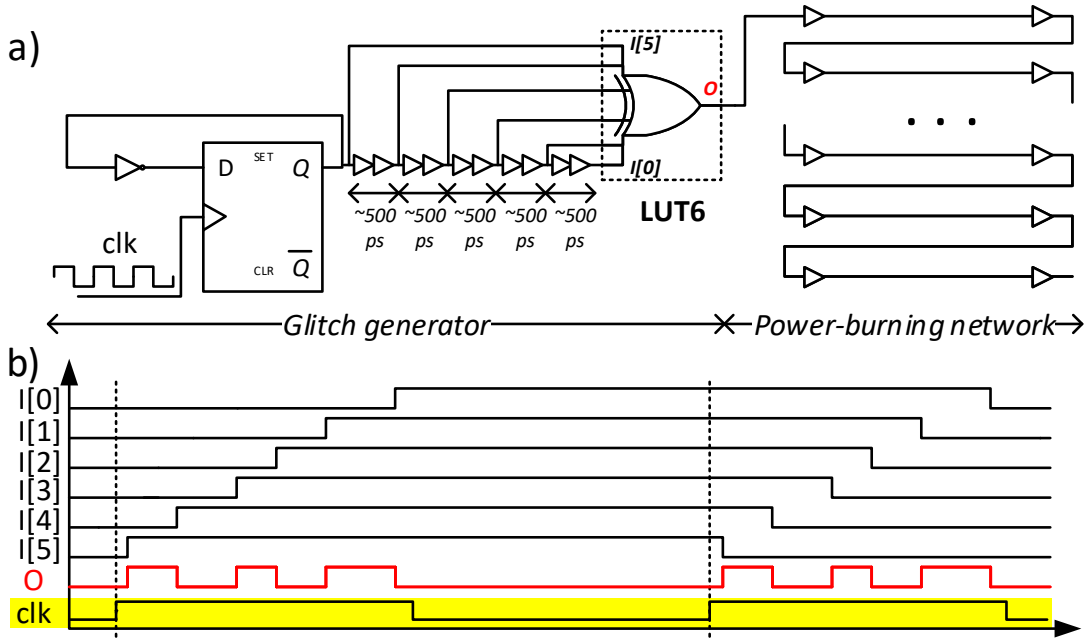

Figure 5.7: Illustrations of the presented glitch amplification attack with (a) the attack circuit consisting of glitch generators and power-burning networks, and (b) the waveform of the signals involved in the attack circuit.

loop whose value oscillates. That is usually achieved by using an odd number of NOT gates. Consequently, ROs can reach frequencies close to 6GHz, much faster than any real-world FPGA design could ever toggle [198]. As they run much faster than the clock, ROs are a source to precisely measure system states as well as sources to draw excessive waste power. Consequently, Amazon prohibits using designs with combinatorial loops in cloud servers that provide FPGAs [11]. Nevertheless, there have been papers showing ring oscillator designs that bypass vendor tools' Design Rule Checking (DRC) with no ring oscillators flagged [89].

These oscillators show that corresponding attacks can be surprisingly straightforward, and anyone can use the cloud services to carry through an attack (*script kiddie* class of attack). For this reason, mitigation strategies for FPGA security attacks are of paramount importance. Hence, one of the virus scanner scans detects all loops in the design that can show up between synchronised signal sources and drains. As we do not detect if these loops oscillate, there is a degree of false positives, but all legitimate loops can be redesigned or flagged appropriately.

Glitch Amplification

Once all ROs have been reliably detected, there are other ways of switching circuits faster than the clock - one of which is glitching. In digital circuits, glitches are fluctuating signal states that appear after input signals change and before output signals fully settle. These glitches can be created intentionally when multiple input signals arrive with significant time disparity in a combinatorial circuit. Thus it is possible to create high-speed glitching signals that cause substantial power spikes on long output antennas with a large fanout (demonstrated in Figure 5.7).

Glitches can be amplified through an XOR gate, which has the property that any change at the input causes a change in the output. Therefore by adjusting routing delays, it is possible to physically implement an oscillator where a source of a toggle flip-flop is routed to an XOR with different delays to create glitches that, in turn, are fed back to the toggle flip-flop. However, there are ways to hide such malicious designs, such as using less glitchy functions at LUTs while generating glitches.

We do not use power drawing or timing characteristics in our graph, so instead of accurate power draw modelling (the vendors' tools do not report this accurately either), we get an upper-bound activity estimate that can flag false positives. Therefore after successfully identifying designs that force an Ultra96 board to shut down with either malicious ROs or glitch amplification designs, we had to update our graph generation and glitch scanning to include LUT functionality parsing for more accurate detection. Our activity estimation is based on how many signal state switches occur (in the worst case scenario) in the design, assuming all inputs switch states. Designs with ROs would be reported to have an infinite activity value (as we do not have timing characteristics, looping signals can keep switching an uncountable number of times before the next input switch) and as such, having a ROs detecting scan beforehand is compulsory.

Once we know the design does not have any ROs, we can use the following steps to detect more hard-to-spot glitch amplification attacks:

1. For all used LUTs in the design, find how many inputs this LUT uses and then find the maximum number of times the output can change given changes on the input side - this will be our denominator.

2. For all used LUTs in the design, find the exact LUT function (i.e., XOR, NXOR, AND, OR) and count the number of times the output changes with all possible combinations on the input side - this will be our numerator.

3. Now, for all used LUTs in the design, we can find the probability that a change

on the input side affects the output of a LUT and count the upper bound signal state switch count of the entire design, assuming all of the inputs change on any given clock cycle. Based on the board, a certain threshold can mark activity values higher than the threshold as malicious.

This process shows how such a system could be maintained security-wise and upgraded as new attacks are found, just like the system could be expanded functionality-wise with new HW modules.

5.3.3 Data Isolation

Now, simply using safe HW modules does not make the system safe. The modules can still break the system if they break the data flow or the interface rules, which can regularly occur accidentally. Alternatively, these behaviours could be hidden more maliciously, like a HW module that intentionally loses data packets in long-running jobs is difficult to spot. Since modules can have internal memory and access to all of the streams that pass through them, they can even store data from other streams and leak sensitive data.

Allowing the general public to use such black-box modules would be imprudent, especially in a multi-tenant scenario. Therefore module designs and their corresponding custom driver code has to get verified before their use becomes available to the general public. Otherwise, they could be used if a user has exclusive access to the whole system, and at the end of each session, the SW and HW systems are cleared.

The scanner mentioned in the previous subsection helps keep HW undamaged here and is more beneficial for alternative systems like FOS, where different modules do not form dataflow pipelines. Nevertheless, FOS and the current system could be used simultaneously, where some slots are used for static designs while others are partitioned into more fine-grain slots for dynamic dataflow acceleration.

5.4 Chapter Conclusion

In this chapter, we complement an ecosystem for dataflow systems on FPGAs to increase productivity with code reuse and reduced tool runtimes (compilation times). In short, we looked at the following aspects: 1) generalisation, 2) scalability, and 3) security.

First, for generalisation, we discussed the overlapping features between different modules in our module library and how shared driver features could support adding additional modules in the future. Building systems with various parts that have shared code is more accessible using SW principles derived from object-oriented programming. Being able to characterise module features and building up new support infrastructure based on existing features enables faster system building and even hot-swapping different parts during runtime if necessary (similarly to the HW pipelines).

To support scaling this ecosystem, we defined the problem of placing different modules of different tasks such that it is possible to find the best configuration for any dataflow problem. By assigning each PR region and module a corresponding resource string and then counting with string matching the popularity of each column, we effectively demonstrated a way to assess the whole library's weak points. This approach would eventually allow a farming approach for creating various module libraries for each desired FPGA device or PR region.

Lastly, we discussed security concerns and how these could be resolved in an iterative and scalable manner with a virus scanner and strict execution policies in addition to existing traditional security measures. Different levels of extensibility and freedom come with various costs that can be tuned for each situation.

These findings have highlighted all design concerns associated with building a runtime system that fully utilises FPGAs' rarely used runtime PR capabilities. Using physically implemented modules is still beneficial in design time for application domains and problem sizes where PR is not required. With the generalisable constraints, the placement problem can be used in any system as a domain-specific compiler where the modules in the supported HW library act similarly to machine code instructions. Such flexibility will be crucial to building up a supporting ecosystem and infrastructure.

Chapter 6

Conclusion

In this last chapter, we conclude the thesis by presenting how this approach cultivates creating an ecosystem around reusable code for FPGAs. Therefore first, we give an overview of contributions in Section 6.1. Then the thesis concludes with a future work discussion in Section 6.2.

6.1 Outcome

FPGAs provide high throughput and energy efficiency, particularly for stream processing problems. We proposed a system to create, manage, and run optimised data flow-oriented acceleration pipelines on FPGAs for problems with complex dependency topologies like data analytics directly under the control of a runtime system such that arbitrary high-level requests (e.g., SQL queries) can be automatically executed on an FPGA (which is only bound by the available modules and memory in the system). As a result, we designed a dynamic stream processing platform where modules are picked automatically from a pre-synthesised library by a scheduler while performing optimisations to improve FPGA utilisation and performance (by reducing the number of runs through the FPGA). The system can be considered a JIT compiler as it translates acceleration requests into stream processing pipelines, significantly lowering the barrier of entry to using specialised accelerators.

6.1.1 Overview

The core aspects required to build a full working resource elastic dynamic stream processing includes:

Responsive Reconfigurability: First and foremost, this system parses SQL to accelerate it on an FPGA and this capability was publicly demonstrated [201]. What sets it apart from related work are the following benefits: 1) this does not require any compilation/synthesis steps that are required for HLS systems, 2) it does not require a large amount of overprovisioning that comes with static solutions, and 3) given suitable HW modules, it can perform all calculations on an FPGA without using generalised soft-core logic. All of this is enabled by dynamic partial reconfiguration.

Performance: Using PR leads to our second outcome of this study: we showed that the overhead costs of DPR can be overcome by accelerating applications that comprise mutual exclusive or dynamic parts of execution and sufficient large datasets. This performance increase is possible with resource elasticity. Enabling dynamic load balancing and intelligent resource allocation to operations that may have varying workloads due to the ability to split a problem into multiple runs using PR can improve the system's flexibility and adaptability to meet different runtime requirements.

Dataflow Orchestration: This project used a given resource elastic module library that uses a streaming interface required for creating dataflow pipelines on-the-fly. In order to swap modules, it uses a resource allocation protocol where each module's resource requirements are given with resource footprints corresponding to fine-grain thin and tall PR region partitions. However, in this thesis, in order to make such a system (with wide data paths and various constraints set by the modules and the operations) run in practice, we built a corresponding SW stack that can format the data to meet various module constraints while also operating the whole flow while processing and tracking these conditions automatically. These on-the-fly dataflow pipelines help avoid the von Neumann bottleneck.

Flexible Interfacing: Such dynamic flexibility has to be scheduled, and as the scheduling has to be able to react to runtime conditions, it has to be automatic. Therefore we have to break down incoming requests and extract all dependencies between different operations and the modules that can execute these operations and track the parameters of these tasks. Given a flexible module library, we use a functional capacity metric to check how many resources are required for each operation and build a graph-like IR that works with our stream orchestration SW

stack. Different front-end applications can create requests using this IR as had been demonstrated for DBMS systems - demonstrating the flexibility necessary to work in multiple stream processing workloads.

Scheduling: The requirement to schedule these modules both in time and space as opposed to just time, which is the case for traditional CPU schedulers, resulted in the formulation of the presented scheduling approach. Given the NP-hard complexity of this task, we see that this requires either excessive scheduling time or processing power (which is not ideal in a runtime system with a limited number of resources), a lot of historical data (not always available with ad-hoc systems), or heuristics and accepting suboptimal scheduling. Nevertheless, we were able to demonstrate the performance benefits of this dynamic approach when executing with large datasets and increasing the computational load per I/O operation. As a result, the execution times were significantly reduced.

Generalisability: As more complex tasks show better performance improvements, we need to make the infrastructure as scalable and maintainable as possible. For this reason, we showed how creating various interfaces to decouple all system parts from each other and enabling different parts to be developed in isolation helps scalability. Meanwhile, the dynamic approach still offers the option to compile everything together statically for high-performance-oriented use cases. Both in HW and SW consider the security aspects highlighted in the last chapter of this thesis.

Abstraction: The here developed runtime systems manages FPGA resources entirely transparent by providing users with a high-level view on the system such that optimised stream processing applications are composed, partial reconfiguration is invoked, and the whole operation is controlled by only using SQL.

Open-Source: All this work is open source and is already being improved upon and examined for future work. The following will highlight the open-source tools this thesis project helped create.

6.1.2 Open-Source Tools

Creating more open-source tools for the community to build upon, energises the community to create new novel ideas while learning how to advocate more reuse of existing work. Hence here is a list of repositories for tools related to this project:

OrkhestraFPGAStream [197]: Creating dataflow pipelines on the ZCU102 is available to experiment with in an interactive shell we showcased in the demo paper [201]. It is possible to change some parameters of the system and then execute SQL queries parsed either with PostgreSQL or TrinoSQL, given that these are set up on the FPGA.

FPGADefender [196]: In order to check for any malicious activity inside bitstreams from unknown sources, this virus scanner tool can scan for various patterns and report the degree of suspiciousness. This tool is currently being made faster and more customisable by Joe Powell [249].

Byteman [190]: Lastly, it is crucial to highlight an open-source tool that can support such workflow where modules can be relocated after synthesis. Byteman (a successor to BitMan [238]) encapsulates the information about bitstreams in Xilinx's technical reference manuals about the instructions to program FPGAs and allows changing these to cut, merge and move bitstreams around on a large amount of boards [192].

6.2 Future Work

As this thesis heavily focuses on the maintainability of this system, currently, the next aspect that this system should improve on the most is still paradoxically maintainability. For larger scale adoption, more debugging HW and SW features are required that are not a core topic of this thesis. Furthermore, more automation is needed to create these HW and SW features, as the main problem of this idea is that it requires a scale that this project still needs to reach. Then this maintainability needs to be enforced by some practical requirements like automatically using runtime hot swappable SW parts. For example, this can get tested with a meta-heuristic scheduler. Once there is enough historical data, a better scheduler (meta-heuristic) should be used, enforcing a higher degree of decoupling and hence maintainability of the system.

Alternative examples also require runtime statistics gathering and making changes accordingly. For instance, the scheduler could be improved to use a more complicated model for calculating the streaming cost, as with multiple simultaneous streams, there could be situations where the execution speed of the modules becomes a bottleneck. Then various concurrent streaming speeds should be monitored, and a more appropriate DMA module should be chosen that streams multiple streams at different priorities

rather than giving all streams an equal amount of packets.

Aside from SW features, more HW features and accelerator modules would improve this system's quality of benchmarking results. We used DBMS acceleration as a stress test for FPGA stream processing with varying operator and utility characteristics. All hardware and software parts in our system are interchangeable independently for maximum reusability and productivity benefits. As the next step, more modules are needed, particularly a GROUP-BY module, to run the whole TCP-H benchmark. Additionally, more scheduling strategies and data serialization techniques will have to be tested for integration with existing systems.

Lastly, migrating to different system layouts requires additional infrastructure work. However, this effort could eventually lead up to being able to report competitive performance numbers in practice rather than in theory (e.g., whole TPC-H and TPC-DS numbers with access to fast and large memory). With different cost and utility models, it is also essential to determine how much discovery is required to find the best ratios without spending too much time on scheduling. Moreover, if the cost and utility values are worse than the CPU alternative, the CPU should process the data instead, leading to tighter couplings with a CPU DBMS, for example.

6.3 Closing Thoughts

Creating ecosystems with such scale is a chicken-and-an-egg problem. When querying chatbots powered by an AI model, like GPT-3 for instance, that is pre-trained with a dataset consisting of a diverse set of websites, books and articles from before 2021, we get an estimation of the general perception of any topic, albeit this is unreliable, yet still fascinating [59]. We queried OpenGPT, which is built on top of the Generative Pre-trained Transformer (GPT-3.5) language model. This model has been fine-tuned (hence the .5 version difference) for conversation using both supervised and reinforcement learning methods, and it tries to predict appropriate sequences of text. Therefore we asked if DPR can make FPGAs more usable with FPGA-OS-like frameworks and what the performance consequences would be. Here is the answer after asking the bot to make the answer more concise:

> *Dynamic Partial Reconfiguration (DPR) allows for real-time reconfiguration of a portion of an FPGA, providing added flexibility to FPGA-OS.*

This technique is achieved by dividing the FPGA into multiple reconfigurable regions that can be independently configured, enabling parallel execution of multiple tasks on the same FPGA. However, DPR can introduce latency and impact performance. Nevertheless, the added flexibility and ease of use of FPGA-OS can outweigh these performance penalties, which can also be mitigated by optimized communication interfaces and memory controllers. DPR is a powerful tool to increase the adoption of FPGAs as general-purpose accelerators, but it comes with trade-offs in performance that depend on the use-case.

FPGAs have been in use for over 30 years, but there is still no commonly agreed-upon answer for what an FPGA-OS should look like. The system presented here shows a major direction how resource elasticity can be used to transparently boost performance among multiple queries (or applications) while considering runtime conditions.

With reusable HW making new accelerators can be done during runtime. As more modules, schedulers, DMAs, and streaming interface options become available for modular projects similar to our prototype, it becomes easier to reuse other people's work. Such sharing is essential to building and growing an ecosystem to make FPGAs more of a commodity, which also increases the effectiveness of these systems.

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*, pages 265–283. USENIX Association, 2016.

[2] François Abel, Jagath Weerasinghe, Christoph Hagleitner, Beat Weiss, and Stephan Paredes. An FPGA Platform for Hyperscalers. In *Hot Interconnects*, pages 29–32. IEEE Computer Society, 2017.

[3] Muhammed Kawser Ahmed, Joel Mandebi, Sujan Kumar Saha, and Christophe Bobda. Multi-Tenant Cloud FPGA: A Survey on Security. *CoRR*, abs/2209.11158, 2022.

[4] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. Weaving Relations for Cache Performance. In *VLDB*, pages 169–180. Morgan Kaufmann, 2001.

[5] Jasmin Ajanovic. PCI express*(PCIe*) 3.0 Accelerator Features. *Intel Corporation*, 10:2–2, 2008.

[6] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing . *Proc. VLDB Endow.*, 8(12):1792–1803, 2015.

[7] Gustavo Alonso, Zsolt István, Kaan Kara, Muhsen Owaida, and David Sidler.

doppioDB 1.0: Machine Learning inside a Relational Engine. *IEEE Data Eng. Bull.*, 42(2):19–31, 2019.

[8] Gustavo Alonso, Timothy Roscoe, David A. Cock, Mohsen Ewaida, Kaan Kara, Dario Korolija, David Sidler, and Zeke Wang. Tackling Hardware/Software Co-design from a Database Perspective. In *CIDR*. www.cidrdb.org, 2020.

[9] Sara Alonso, Jesús Lázaro, Jaime Jiménez, Leire Muguira, and Alejandro Largacha. Analysing the interference of Xen hypervisor in the network speed. In *DCIS*, pages 1–6. IEEE, 2020.

[10] Eman Alqudah and Amin Jarrah. Parallel implementation of genetic algorithm on FPGA using Vivado high level synthesis . *Int. J. Bio Inspired Comput.*, 15(2):90–99, 2020.

[11] Amazon Inc. AWS EC2 FPGA HDK+SDK Errata, 2023. `https://github.com/aws/aws-fpga/blob/master/ERRATA.md`.

[12] AMD. AMD Acquires Xilinx, 2022. `https://www.amd.com/en/corporate/xilinx-acquisition`.

[13] Gene M. Amdahl. Computer Architecture and Amdahl's Law. *Computer*, 46(12):38–46, 2013.

[14] Robin Anil, Gökhan Çapan, Isabel Drost-Fromm, Ted Dunning, Ellen Friedman, Trevor Grant, Shannon Quinn, Paritosh Ranjan, Sebastian Schelter, and Özgür Yilmazel. Apache Mahout: Machine Learning on Distributed Dataflow Systems. *J. Mach. Learn. Res.*, 21:127:1–127:6, 2020.

[15] Apache Software Foundation. Apache Projects Directory, 2023. `https://projects.apache.org/projects.html`.

[16] Janniele A. S. Araujo, Haroldo G. Santos, Bernard Gendron, Sanjay Dominik Jena, Samuel S. Brito, and Danilo S. Souza. Strong bounds for resource constrained project scheduling: Preprocessing and cutting planes . *Comput. Oper. Res.*, 113, 2020.

[17] Florian Arrestier, Karol Desnos, Maxime Pelcat, Julien Heulot, Eduardo Juárez, and Daniel Ménard. Delays and states in dataflow models of computation. In *SAMOS*, pages 47–54. ACM, 2018.

[18] Praveenkumar Babu and Eswaran Parthasarathy. Reconfigurable FPGA architectures: A survey and applications. *Journal of The Institution of Engineers (India): Series B*, 102(1):143–156, 2021.

[19] Marco Bacis, Rolando Brondolin, and Marco D. Santambrogio. BlastFunction: an FPGA-as-a-Service system for Accelerated Serverless Computing . In *DATE*, pages 852–857. IEEE, 2020.

[20] Melyssa Barata, Jorge Bernardino, and Pedro Furtado. An Overview of Decision Support Benchmarks: TPC-DS, TPC-H and SSB. In *WorldCIST (1)*, volume 353 of *Advances in Intelligent Systems and Computing*, pages 619–628. Springer, 2015.

[21] Andreas Becher, Lekshmi B. G., David Broneske, Tobias Drewes, Bala Gurumurthy, Klaus Meyer-Wegener, Thilo Pionteck, Gunter Saake, Jürgen Teich, and Stefan Wildermann. Integration of FPGAs in Database Management Systems: Challenges and Opportunities . *Datenbank-Spektrum*, 18(3):145–156, 2018.

[22] Tobias Becker, Markus Koester, and Wayne Luk. Automated placement of reconfigurable regions for relocatable modules. In *ISCAS*, pages 3341–3344. IEEE, 2010.

[23] Christian Beckhoff, Dirk Koch, and Jim Tørresen. Short-Circuits on FPGAs Caused by Partial Runtime Reconfiguration. In *2010 International Conference on Field Programmable Logic and Applications*, pages 596,601. IEEE, 2010-08.

[24] Christian Beckhoff, Dirk Koch, and Jim Tørresen. Go Ahead: A Partial Reconfiguration Framework. In *FCCM*, pages 37–44. IEEE Computer Society, 2012.

[25] Christian Beckhoff, Dirk Koch, and Jim Tørresen. Automatic Floorplanning and Interface Synthesis of Island Style Reconfigurable Systems with GoAhead . In *ARCS*, volume 7767 of *Lecture Notes in Computer Science*, pages 303–316. Springer, 2013.

[26] Christian Beckhoff, Dirk Koch, and Jim Tørresen. Portable module relocation and bitstream compression for Xilinx FPGAs. In *FPL*, pages 1–8. IEEE, 2014.

[27] Eugen Betke and Julian M. Kunkel. Real-Time I/O-Monitoring of HPC Applications with SIOX, Elasticsearch, Grafana and FUSE . In *ISC Workshops*,

volume 10524 of *Lecture Notes in Computer Science*, pages 174–186. Springer, 2017.

[28] Alessandro Biondi, Alessio Balsini, Marco Pagani, Enrico Rossi, Mauro Marinoni, and Giorgio C. Buttazzo. A Framework for Supporting Real-Time Applications on Dynamic Reconfigurable FPGAs . In *RTSS*, pages 1–12. IEEE Computer Society, 2016.

[29] Jacek Blazewicz, Jan Karel Lenstra, and A. H. G. Rinnooy Kan. Scheduling subject to resource constraints: classification and complexity . *Discret. Appl. Math.*, 5(1):11–24, 1983.

[30] Michaela Blott, Thomas B. Preußer, Nicholas J. Fraser, Giulio Gambardella, Kenneth O'Brien, Yaman Umuroglu, Miriam Leeser, and Kees A. Vissers. FINN-*R*: An End-to-End Deep-Learning Framework for Fast Exploration of Quantized Neural Networks . *ACM Trans. Reconfigurable Technol. Syst.*, 11(3):16:1–16:23, 2018.

[31] Christophe Bobda, Joel Mandebi Mbongue, Paul Chow, Mohammad Ewais, Naif Tarafdar, Juan Camilo Vega, Ken Eguro, Dirk Koch, Suranga Handagala, Miriam Leeser, Martin C. Herbordt, Hafsah Shahzad, Peter Hofste, Burkhard Ringlein, Jakub Szefer, Ahmed Sanaullah, and Russell Tessier. The Future of FPGA Acceleration in Datacenters and the Cloud. *ACM Trans. Reconfigurable Technol. Syst.*, 15(3):34:1–34:42, 2022.

[32] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. Breaking the memory wall in MonetDB. *Commun. ACM*, 51(12):77–85, 2008.

[33] Peter A. Boncz, Thomas Neumann, and Orri Erling. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark . In *TPCTC*, volume 8391 of *Lecture Notes in Computer Science*, pages 61–76. Springer, 2013.

[34] VG Bondur. Modern approaches to processing large hyperspectral and multispectral aerospace data flows . *Izvestiya, Atmospheric and Oceanic Physics*, 50(9):840–852, 2014.

[35] Charl P. Botha and Frits H. Post. Hybrid Scheduling in the DeVIDE Dataflow Visualisation Environment. In *SimVis*, pages 309–322. SCS Publishing House e.V., 2008.

[36] Adnan Bouakaz, Pascal Fradet, and Alain Girault. A Survey of Parametric Dataflow Models of Computation. *ACM Trans. Design Autom. Electr. Syst.*, 22(2):38:1–38:25, 2017.

[37] Frédéric Boulanger, Cécile Hardebolle, Christophe Jacquet, and Dominique Marcadet. Semantic Adaptation for Models of Computation. In *ACSD*, pages 153–162. IEEE Computer Society, 2011.

[38] Sebastian Breß, Henning Funke, and Jens Teubner. Robust Query Processing in Co-Processor-accelerated Databases. In *SIGMOD Conference*, pages 1891–1906. ACM, 2016.

[39] Sebastian Breß, Max Heimel, Norbert Siegmund, Ladjel Bellatreche, and Gunter Saake. GPU-Accelerated Database Systems: Survey and Open Challenges. *TLDKS*, 15:1–35, 2014.

[40] Joseph T Buck. Static scheduling and code generation from dynamic dataflow graphs with integer-valued control streams . In *Proceedings of 1994 28th Asilomar Conference on Signals, Systems and Computers*, volume 1, pages 508–513. IEEE, 1994.

[41] Alex R. Bucknall, Shanker Shreejith, and Suhaib A. Fahmy. Build Automation and Runtime Abstraction for Partial Reconfiguration on Xilinx Zynq UltraScale+ . In *FPT*, pages 215–220. IEEE, 2020.

[42] Canonical. Install Ubuntu on Xilinx, 2021. `https://ubuntu.com/download/xilinx`.

[43] Jacques Carette, Roshan P. James, and Amr Sabry. Chapter Two - Embracing the laws of physics: Three reversible models of computation . *Adv. Comput.*, 126:15–63, 2022.

[44] Riccardo Cattaneo, Christian Pilato, Matteo Mastinu, Oliver Kadlcek, Oliver Pell, and Marco Domenico Santambrogio. Runtime adaptation on dataflow HPC platforms. In *AHS*, pages 84–91. IEEE, 2013.

[45] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. A

Cloud-Scale Acceleration Architecture. In *MICRO*, pages 7:1–7:13. IEEE Computer Society, 2016.

[46] Najdet Charaf, Christoph Tietz, and Diana Goehringer. MaNaBIT: A Versatile Tool for Manipulating and Analyzing FPGA Bitstreams. In *FCCM*, page 1. IEEE, 2022.

[47] Fei Chen, Yi Shan, Yu Zhang, Yu Wang, Hubertus Franke, Xiaotao Chang, and Kun Wang. Enabling FPGAs in the cloud. In *Conf. Computing Frontiers*, pages 3:1–3:10. ACM, 2014.

[48] Shimin Chen, Anastasia Ailamaki, Manos Athanassoulis, Phillip B. Gibbons, Ryan Johnson, Ippokratis Pandis, and Radu Stoica. TPC-E vs. TPC-C: characterizing the new TPC-E benchmark via an I/O comparison study . *SIGMOD Rec.*, 39(3):5–10, 2010.

[49] Hawon Chu, Seounghyun Kim, Joo-Young Lee, and Young-Kyoon Suh. Empirical evaluation across multiple GPU-accelerated DBMSes. In *DaMoN*, pages 16:1–16:3. ACM, 2020.

[50] W. Paul Cockshott and Greg Michaelson. Are There New Models of Computation? Reply to Wegner and Eberbach. *Comput. J.*, 50(2):232–247, 2007.

[51] Philip Colangelo, Enno Lübbers, Randy Huang, Martin Margala, and Kevin Nealis. Application of convolutional neural networks on Intel® Xeon® processor with integrated FPGA. In *HPEC*, pages 1–7. IEEE, 2017.

[52] M. Colgan. Does GPU hardware help Database workloads?, 2018. `https://blogs.oracle.com/database/does-gpu-hardware-help-database-workloads/`.

[53] Jason Cong, Zhenman Fang, Michael Lo, Hanrui Wang, Jingxian Xu, and Shaochong Zhang. Understanding Performance Differences of FPGAs and GPUs. In *FCCM*, pages 93–96. IEEE Computer Society, 2018.

[54] Jason Cong, Jason Lau, Gai Liu, Stephen Neuendorffer, Peichen Pan, Kees Vissers, and Zhiru Zhang. FPGA HLS Today: Successes, Challenges, and Opportunities. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 15(4):1–42, 2022.

[55] Alejandro Corbellini, Cristian Mateos, Alejandro Zunino, Daniela Godoy, and Silvia N. Schiaffino. Persisting big-data: The NoSQL landscape. *Inf. Syst.*, 63:1–23, 2017.

[56] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.*, 31(3):8, 2013.

[57] Jordi Cortadella, Marc Galceran Oms, Michael Kishinevsky, and Sachin S. Sapatnekar. RTL Synthesis: From Logic Synthesis to Automatic Pipelining. *Proc. IEEE*, 103(11):2061–2075, 2015.

[58] João Paulo Sá da Silva, Valery Sklyarov, and Iouliia Skliarova. Comparison of On-chip Communications in Zynq-7000 All Programmable Systems-on-Chip . *IEEE Embed. Syst. Lett.*, 7(1):31–34, 2015.

[59] Robert Dale. GPT-3: What's it good for? *Nat. Lang. Eng.*, 27(1):113–118, 2021.

[60] Claudia D'Ambrosio and Andrea Lodi. Mixed integer nonlinear programming tools: an updated practical overview. *Ann. Oper. Res.*, 204(1):301–320, 2013.

[61] Jonas Dann, Daniel Ritter, and Holger Fröning. Non-Relational Databases on FPGAs: Survey, Design Decisions, Challenges. *CoRR*, abs/2007.07595, 2020.

[62] Klaus Danne and Marco Platzner. A Heuristic Approach to Schedule Periodic Real-Time Tasks on Reconfigurable Hardware . In *FPL*, pages 568–573. IEEE, 2005.

[63] Ali Davoudian, Liu Chen, and Mengchi Liu. A Survey on NoSQL Stores. *ACM Comput. Surv.*, 51(2):40:1–40:43, 2018.

[64] Marcos Dias de Assunção, Alexandre Da Silva Veith, and Rajkumar Buyya. Distributed data stream processing and edge computing: A survey on resource elasticity and future directions . *J. Netw. Comput. Appl.*, 103:1–17, 2018.

[65] Ulan Degenbaev. *Formal specification of the x86 instruction set architecture*. PhD thesis, Saarland University, 2012.

[66] André DeHon, Yury Markovsky, Eylon Caspi, Michael Chu, Randy Huang, Stylianos Perissakis, Laura Pozzi, Joseph Yeh, and John Wawrzynek. Stream Computations Organized for Reconfigurable Execution. *Microprocess. Microsystems*, 30(6):334–354, 2006.

[67] Serge Demeyer. Refactor Conditionals into Polymorphism: What's the Performance Cost of Introducing Virtual Calls? . In *ICSM*, pages 627–630. IEEE Computer Society, 2005.

[68] Christopher Dennl, Daniel Ziener, and Jürgen Teich. On-the-fly Composition of FPGA-Based SQL Query Accelerators Using a Partially Reconfigurable Module Library . In *FCCM*, pages 45–52. IEEE Computer Society, 2012.

[69] Dionysios Diamantopoulos, Raphael Polig, Burkhard Ringlein, Mitra Purandare, Beat Weiss, Christoph Hagleitner, Mark A. Lantz, and François Abel. Acceleration-as-a-μService: A Cloud-native Monte-Carlo Option Pricing Engine on CPUs, GPUs and Disaggregated FPGAs. In *CLOUD*, pages 726–729. IEEE, 2021.

[70] Amilcar do Carmo Lucas, Sven Heithecker, and Rolf Ernst. FlexWAFE - A High-end Real-Time Stream Processing Library for FPGAs. In *DAC*, pages 916–921. IEEE, 2007.

[71] Gordana Dodig-Crnkovic. Significance of Models of Computation, from Turing Model to Natural Computation . *Minds Mach.*, 21(2):301–322, 2011.

[72] Markus Dreseler, Martin Boissier, Tilmann Rabl, and Matthias Uflacker. Quantifying TPC-H Choke Points and Their Optimizations. *Proc. VLDB Endow.*, 13(8):1206–1220, 2020.

[73] François Duhem, Fabrice Muller, Robin Bonamy, and Sébastien Bilavarn. FoRTReSS: a flow for design space exploration of partially reconfigurable systems . *Des. Autom. Embed. Syst.*, 19(3):301–326, 2015.

[74] François Duhem, Fabrice Muller, and Philippe Lorenzini. FaRM: Fast Reconfiguration Manager for Reducing Reconfiguration Time Overhead on FPGA .

In *ARC*, volume 6578 of *Lecture Notes in Computer Science*, pages 253–260. Springer, 2011.

[75] Stephen A. Edwards, Luciano Lavagno, Edward A. Lee, and Alberto L. Sangiovanni-Vincentelli. Design of embedded systems: formal models, validation, and synthesis. *Proc. IEEE*, 85(3):366–390, 1997.

[76] Joerg Evermann, Jana-Rebecca Rehse, and Peter Fettke. Process Discovery from Event Stream Data in the Cloud - A Scalable, Distributed Implementation of the Flexible Heuristics Miner on the Amazon Kinesis Cloud Infrastructure . In *CloudCom*, pages 645–652. IEEE Computer Society, 2016.

[77] Jian Fang, Yvo T. B. Mulder, Jan Hidders, Jinho Lee, and H. Peter Hofstee. In-memory Database Acceleration on FPGAs: a Survey. *VLDB J.*, 29(1):33–59, 2020.

[78] Anthony C. J. Fox and Magnus O. Myreen. A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture . In *ITP*, volume 6172 of *Lecture Notes in Computer Science*, pages 243–258. Springer, 2010.

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

[80] Free Software Foundation. GNU Binary Utilities, 2023. `https://sourceware.org/binutils/docs-2.40/binutils/index.html`.

[81] Juan Fumero, Michail Papadimitriou, Foivos S. Zakkak, Maria Xekalaki, James Clarkson, and Christos Kotselidis. Dynamic Application Reconfiguration on Heterogeneous Hardware. In *VEE*, pages 165–178. ACM, 2019.

[82] Brian Gaide, Dinesh Gaitonde, Chirag Ravishankar, and Trevor Bauer. Xilinx Adaptive Compute Acceleration Platform: Versal$^{\text{TM}}$ Architecture. In *FPGA*, pages 84–93. ACM, 2019.

[83] Carlo Galuzzi and Koen Bertels. The Instruction-Set Extension Problem: A Survey. *ACM Trans. Reconfigurable Technol. Syst.*, 4(2):18:1–18:28, 2011.

[84] Lin Gan, Haohuan Fu, Oskar Mencer, Wayne Luk, and Guangwen Yang. Chapter Four - Data Flow Computing in Geoscience Applications. *Adv. Comput.*, 104:125–158, 2017.

[85] Shubham Gandhare and B Karthikeyan. Survey on FPGA architecture and recent applications. In *2019 International Conference on Vision Towards Emerging Trends in Communication and Networking (ViTECoN)*, pages 1–4. IEEE, 2019.

[86] Yuanpei Gao, Haijiang Ye, Jian Wang, and Jinmei Lai. FPGA bitstream compression and decompression based on LZ77 algorithm and BMC technique . In *ASICON*, pages 1–4. IEEE, 2015.

[87] Marc Geilen and Twan Basten. Requirements on the Execution of Kahn Process Networks. In *ESOP*, volume 2618 of *Lecture Notes in Computer Science*, pages 319–334. Springer, 2003.

[88] Ahmad Ghazal, Tilmann Rabl, Minqing Hu, Francois Raab, Meikel Poess, Alain Crolotte, and Hans-Arno Jacobsen. BigBench: towards an industry standard benchmark for big data analytics. In *SIGMOD Conference*, pages 1197–1208. ACM, 2013.

[89] I. Giechaskiel, K. Rasmussen, and J. Szefer. Measuring Long Wire Leakage with Ring Oscillators in Cloud FPGAs. In *Proceedings of the International Conference on Field-Programmable Logic and Applications*, FPL, September 2019.

[90] Dennis R. E. Gnad, Fabian Oboril, and Mehdi Baradaran Tahoori. Voltage Drop-based Fault Attacks on FPGAs Using Valid Bitstreams. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–7. IEEE, 2017.

[91] Herman H. Goldstine and Adele Goldstine. The Electronic Numerical Integrator and Computer (ENIAC). *IEEE Ann. Hist. Comput.*, 18(1):10–16, 1996.

[92] Nicolae Bogdan Grigore and Dirk Koch. Placing Partially Reconfigurable Stream Processing Applications on FPGAs. In *FPL*, pages 1–4. IEEE, 2015.

[93] Christoph Gröger. Industrial analytics - An overview. *it Inf. Technol.*, 64(1-2):55–65, 2022.

[94] Varun Grover and James T. C. Teng. An examination of DBMS adoption and success in American organizations. *Inf. Manag.*, 23(5):239–248, 1992.

[95] Fei Guan, Long Peng, Luc Perneel, and Martin Timmerman. Open source FreeRTOS as a case study in real-time operating system evolution . *J. Syst. Softw.*, 118:19–35, 2016.

[96] Tim Güneysu and Amir Moradi. Generic Side-Channel Countermeasures for Reconfigurable Devices. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2011*, pages 33–48, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[97] Liucheng Guo, David B. Thomas, Ce Guo, and Wayne Luk. Automated framework for FPGA-based parallel genetic algorithms. In *FPL*, pages 1–7. IEEE, 2014.

[98] Peter J. Haas, Ihab F. Ilyas, Guy M. Lohman, and Volker Markl. Discovering and Exploiting Statistical Properties for Query Optimization in Relational Databases: A Survey . *Stat. Anal. Data Min.*, 1(4):223–250, 2009.

[99] Ilija Hadzic, Sanjay Udani, and Jonathan M. Smith. FPGA Viruses. In *International Workshop on Field Programmable Logic and Applications*, pages 291–300. Springer, 1999.

[100] Tobias Hahn, Andreas Becher, Stefan Wildermann, and Jürgen Teich. Raw Filtering of JSON Data on FPGAs. In *DATE*, pages 250–255. IEEE, 2022.

[101] Amir HajiRassouliha, Andrew J. Taberner, Martyn P. Nash, and Poul M. F. Nielsen. Suitability of recent hardware accelerators (DSPs, FPGAs, and GPUs) for computer vision and image processing algorithms . *Signal Process. Image Commun.*, 68:101–119, 2018.

[102] Svein O. Hallsteinsen, Kurt Geihs, Nearchos Paspallis, Frank Eliassen, Geir Horn, Jorge Lorenzo, Alessandro Mamelli, and George Angelos Papadopoulos. A development framework and methodology for self-adapting applications in ubiquitous computing environments . *J. Syst. Softw.*, 85(12):2840–2859, 2012.

[103] Simen Gimle Hansen, Dirk Koch, and Jim Tørresen. High Speed Partial Run-Time Reconfiguration Using Enhanced ICAP Hard Macro . In *IPDPS Workshops*, pages 174–180. IEEE, 2011.

[104] Theo Härder. DBMS Architecture - Still an Open Problem. In *BTW*, volume P-65 of *LNI*, pages 2–28. GI, 2005.

[105] Guy Harrison. *Database Survey*, pages 217–228. Apress, Berkeley, CA, 2015.

[106] Sönke Hartmann and Dirk Briskorn. A survey of variants and extensions of the resource-constrained project scheduling problem . *Eur. J. Oper. Res.*, 207(1):1–14, 2010.

[107] Sönke Hartmann and Dirk Briskorn. An updated survey of variants and extensions of the resource-constrained project scheduling problem . *Eur. J. Oper. Res.*, 297(1):1–14, 2022.

[108] Sönke Hartmann and Rainer Kolisch. Experimental evaluation of state-of-the-art heuristics for the resource-constrained project scheduling problem . *Eur. J. Oper. Res.*, 127(2):394–407, 2000.

[109] Milad Hashemi, Kevin Swersky, Jamie A. Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. Learning Memory Access Patterns. In *ICML*, volume 80 of *Proceedings of Machine Learning Research*, pages 1924–1933. PMLR, 2018.

[110] Timothy Hayes, Oscar Palomar, Osman S. Unsal, Adrián Cristal, and Mateo Valero. Vector Extensions for Decision Support DBMS Acceleration. In *MICRO*, pages 166–176. IEEE Computer Society, 2012.

[111] Zhenhao He, David Sidler, Zsolt István, and Gustavo Alonso. A Flexible K-Means Operator for Hybrid Databases. In *FPL*, pages 368–371. IEEE Computer Society, 2018.

[112] Thomas Heinis, Cesare Pautasso, and Gustavo Alonso. Design and Evaluation of an Autonomic Workflow Engine. In *ICAC*, pages 27–38. IEEE Computer Society, 2005.

[113] Linus Heinzl, Ben Hurdelhey, Martin Boissier, Michael Perscheid, and Hasso Plattner. Evaluating Lightweight Integer Compression Algorithms in Column-Oriented In-Memory DBMS . In *ADMS@VLDB*, pages 26–36, 2021.

[114] Joost Hoozemans, Johan Peltenburg, Fabian Nonnemacher, Akos Hadnagy, Zaid Al-Ars, and H Peter Hofstee. FPGA Acceleration for Big Data Analytics: Challenges and Opportunities. *IEEE Circuits and Systems Magazine*, 21(2):30–47, 2021.

[115] Augusto Y. Horita, Denis Silva Loubach, and Ricardo Bonna. Analysis and Identification of Possible Automation Approaches for Embedded Systems Design Flows . *Inf.*, 11(2):120, 2020.

[116] Shicheng Hu, Song Wang, Yonggui Kao, Takao Ito, and Xuedong Sun. A Branch and Bound Algorithm for Project Scheduling Problem with Spatial Resource Constraints . *Mathematical Problems in Engineering*, 2015, 2015.

[117] Michael Hübner, Diana Göhringer, Juanjo Noguera, and Jürgen Becker. Fast dynamic and partial reconfiguration data path with low hardware overhead on Xilinx FPGAs . In *IPDPS Workshops*, pages 1–8. IEEE, 2010.

[118] Eddie Hung. Mind the (synthesis) gap: Examining where academic FPGA tools lag behind industry . In *FPL*, pages 1–4. IEEE, 2015.

[119] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. MonetDB: Two Decades of Research in Column-oriented Database Architectures . *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.

[120] Qaiser Ijaz, El-Bay Bourennane, Ali Kashif Bashir, and Hira Asghar. Revisiting the High-Performance Reconfigurable Computing for Future Datacenters . *Future Internet*, 12(4):64, 2020.

[121] Intel. Intel Acquisition of Altera, 2015. `https://newsroom.intel.com/press-kits/intel-acquisition-of-altera/#gs.fj4u8x`.

[122] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Mikio Takeuchi, Takeshi Ogasawara, Toshio Suganuma, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. Design, implementation, and evaluation of optimizations in a JavaTM Just-In-Time compiler . *Concurr. Pract. Exp.*, 12(6):457–475, 2000.

[123] Zsolt István. The Glass Half Full: Using Programmable Hardware Accelerators in Analytics . *IEEE Data Eng. Bull.*, 42(1):49–60, 2019.

[124] Abhishek Kumar Jain, Douglas L. Maskell, and Suhaib A. Fahmy. Resource-Aware Just-in-Time OpenCL Compiler for Coarse-Grained FPGA Overlays . *CoRR*, abs/1705.02730, 2017.

[125] Peter Jamieson, Kenneth B. Kent, Farnaz Gharibian, and Lesley Shannon. Odin II - An Open-Source Verilog HDL Synthesis Tool for CAD Research. In *FCCM*, pages 149–156. IEEE Computer Society, 2010.

[126] Slinger Jansen. Measuring the health of open source software ecosystems: Beyond the scope of project health . *Inf. Softw. Technol.*, 56(11):1508–1519, 2014.

[127] Axel Jantsch. Models of Embedded Computation. In *Embedded Systems Handbook*. CRC Press, 2005.

[128] Jay Carlson. So you want to build an embedded Linux system?, 2020. `https://jaycarlson.net/embedded-linux/`.

[129] Chenglu Jin, Vasudev Gohil, Ramesh Karri, and Jeyavijayan Rajendran. Security of Cloud FPGAs: A Survey. *CoRR*, abs/2005.04867, 2020.

[130] Norman P. Jouppi, Cliff Young, Nishant Patil, and David A. Patterson. Motivation for and Evaluation of the First Tensor Processing Unit. *IEEE Micro*, 38(3):10–19, 2018.

[131] V. Michael Bove Jr. and John A. Watlington. Cheops: a reconfigurable dataflow system for video processing. *IEEE Trans. Circuits Syst. Video Technol.*, 5(2):140–149, 1995.

[132] Inas Jawad Kadhim, Prashan Premaratne, Peter James Vial, and Brendan Halloran. Comprehensive survey of image steganography: Techniques, Evaluations, and trends in future research . *Neurocomputing*, 335:299–326, 2019.

[133] Gilles Kahn. The Semantics of a Simple Language for Parallel Programming. In *IFIP Congress*, pages 471–475. North-Holland, 1974.

[134] Heiko Kalte and Mario Porrmann. REPLICA2Pro: task relocation by bitstream manipulation in virtex-II/Pro FPGAs . In *Conf. Computing Frontiers*, pages 403–412. ACM, 2006.

[135] Ahmed Kamaleldin, Ahmed M. Soliman, Ahmed Nagy, Youssef Gamal, Ahmed Shalash, Yehea Ismail, and Hassan Mostafa. Design guidelines for the high-speed dynamic partial reconfiguration based software defined radio implementations on Xilinx Zynq FPGA . In *ISCAS*, pages 1–4. IEEE, 2017.

[136] Nachiket Kapre and Samuel Bayliss. Survey of domain-specific languages for FPGA computing. In *FPL*, pages 1–12. IEEE, 2016.

[137] Richard M. Karp and Raymond E. Miller. Parallel Program Schemata. *J. Comput. Syst. Sci.*, 3(2):147–195, 1969.

[138] Sahand Kashani, Mahyar Emami, and James R. Larus. Bitfiltrator: A General Approach for Reverse-Engineering Xilinx Bitstream Formats . In *FPL*. IEEE, 2022.

[139] Ichiro Kawazome. u-dma-buf(User space mappable DMA Buffer), 2022. `https://github.com/ikwzm/udmabuf/`.

[140] Nadir Khan, Jorge Castro-Godínez, Shixiang Xue, Jörg Henkel, and Jürgen Becker. Automatic Floorplanning and Standalone Generation of Bitstream-Level IP Cores . *IEEE Trans. Very Large Scale Integr. Syst.*, 29(1):38–50, 2021.

[141] Bart Kienhuis, Ed F. Deprettere, Kees A. Vissers, and Pieter van der Wolf. An Approach for Quantitative Analysis of Application-Specific Dataflow Architectures . In *ASAP*, pages 338–349. IEEE Computer Society, 1997.

[142] Kangnyeon Kim, Ryan Johnson, and Ippokratis Pandis. BionicDB: Fast and Power-Efficient OLTP on FPGA. In *EDBT*, pages 301–312. OpenProceedings.org, 2019.

[143] Denis Kleyko, Edward Paxon Frady, and Friedrich T. Sommer. Cellular Automata Can Reduce Memory Requirements of Collective-State Computing. *IEEE Trans. Neural Networks Learn. Syst.*, 33(6):2701–2713, 2022.

[144] Dirk Koch. *Partial reconfiguration on FPGAs: architectures, tools and applications*, volume 153. Springer Science & Business Media, 2012.

[145] Dirk Koch and Christian Beckhoff. Hierarchical reconfiguration of FPGAs. In *FPL*, pages 1–8. IEEE, 2014.

[146] Dirk Koch, Christian Beckhoff, and Guy G. F. Lemieux. An efficient FPGA overlay for portable custom instruction set extensions. In *FPL*, pages 1–8. IEEE, 2013.

[147] Dirk Koch, Christian Beckhoff, and Jürgen Teich. Bitstream Decompression for High Speed FPGA Configuration from Slow Memories . In *FPT*, pages 161–168. IEEE, 2007.

[148] Dirk Koch, Christian Beckhoff, and Jürgen Teich. ReCoBus-Builder - A novel tool and technique to build statically and dynamically reconfigurable systems for FPGAS. In *FPL*, pages 119–124. IEEE, 2008.

[149] Dirk Koch, Christian Beckhoff, and Jim Tørresen. Zero logic overhead integration of partially reconfigurable modules. In *SBCCI*, pages 103–108. ACM, 2010.

[150] Dirk Koch, Nguyen Dao, Bea Healy, Jing Yu, and Andrew Attwood. FABulous: An Embedded FPGA Framework. In *FPGA*, pages 45–56. ACM, 2021.

[151] Dirk Koch and Jim Tørresen. FPGASort: A High Performance Sorting Architecture Exploiting Run-time Reconfiguration on FPGAs for Large Problem Sorting . In *FPGA*, pages 45–54. ACM, 2011.

[152] Dirk Koch and Jim Tørresen. A Routing Architecture for Mapping Dataflow Graphs at Run-Time. In *FPL*, pages 286–290. IEEE Computer Society, 2011.

[153] Dirk Koch, Jim Tørresen, Christian Beckhoff, Daniel Ziener, Christopher Dennl, Volker Breuer, Jürgen Teich, Michael Feilen, and Walter Stechele. Partial Reconfiguration on FPGAs in Practice - Tools and Applications. In *ARCS Workshops*, volume P-200 of *LNI*, pages 297–319. GI, 2012.

[154] Markus Koester, Wayne Luk, Jens Hagemeyer, Mario Porrmann, and Ulrich Rückert. Design Optimizations for Tiled Partially Reconfigurable Systems. *IEEE Trans. Very Large Scale Integr. Syst.*, 19(6):1048–1061, 2011.

[155] Konstantina Koliogeorgi, Nils Voss, Sotiria Fytraki, Sotirios Xydis, Georgi Gaydadjiev, and Dimitrios Soudris. Dataflow Acceleration of Smith-Waterman with Traceback for High Throughput Next Generation Sequencing . In *FPL*, pages 74–80. IEEE, 2019.

[156] Rainer Kolisch and Sönke Hartmann. Experimental investigation of heuristics for resource-constrained project scheduling: An update . *Eur. J. Oper. Res.*, 174(1):23–37, 2006.

[157] Arne Koschel, Irina Astrova, Stephan-Tobias Alsleben, Jannis Bellok, Niklas Meyer, and Sebastian Meyer. Evaluating Time Series Database Management Systems for Insurance Company. In *2022 13th International Conference on Information, Intelligence, Systems & Applications (IISA)*, pages 1–6, 2022.

[158] Jonas Krautter, Dennis R. E. Gnad, Falk Schellenberg, Amir Moradi, and Mehdi Baradaran Tahoori. Active Fences against Voltage-based Side Channels in Multi-Tenant FPGAs. In *ICCAD*, pages 1–8. ACM, 2019.

[159] Charalampos Kritikakis and Dirk Koch. End-to-end Dynamic Stream Processing on Maxeler HLS Platforms. In *ASAP*, pages 59–66. IEEE, 2019.

[160] Predrag V. Krtolica and Predrag S. Stanimirovic. Reverse polish notation method. *Int. J. Comput. Math.*, 81(3):273–284, 2004.

[161] J. Satheesh Kumar, G. Saravana Kumar, and A. Ahilan. High performance decoding aware FPGA bit-stream compression using RG codes . *Clust. Comput.*, 22(6):15007–15013, 2019.

[162] Ian Kuon and Jonathan Rose. Measuring the Gap Between FPGAs and ASICs. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 26(2):203–215, 2007.

[163] Ian Kuon, Russell Tessier, and Jonathan Rose. FPGA Architecture: Survey and Challenges. *Found. Trends Electron. Des. Autom.*, 2(2):135–253, 2007.

[164] L. Guo et a. RapidStream: Parallel Physical Implementation of FPGA HLS Designs. In *FPGA*, pages 1–12. ACM, 2022.

[165] Tuan Minh La, Kaspar Matas, Nikola Grunchevski, Khoa Dang Pham, and Dirk Koch. FPGADefender: Malicious Self-oscillator Scanning for Xilinx UltraScale + FPGAs . *ACM Trans. Reconfigurable Technol. Syst.*, 13(3):15:1–15:31, 2020.

[166] Sakari Lahti, Panu Sjovall, Jarno Vanne, and Timo D. Hämäläinen. Are We There Yet? A Study on the State of High-Level Synthesis. *IEEE TCAD*, 38(5):898–911, 2019.

[167] Yi-Hsiang Lai, Ecenur Ustun, Shaojie Xiang, Zhenman Fang, Hongbo Rong, and Zhiru Zhang. Programming and Synthesis for Software-defined FPGA Acceleration: Status and Future Prospects . *ACM Trans. Reconfigurable Technol. Syst.*, 14(4):17:1–17:39, 2021.

[168] Andre Lalevee, Pierre-Henri Horrein, Matthieu Arzel, Michael Hübner, and Sandrine Vaton. AutoReloc: Automated Design Flow for Bitstream Relocation on Xilinx FPGAs . In *DSD*, pages 14–21. IEEE Computer Society, 2016.

[169] Hai Lan, Zhifeng Bao, and Yuwei Peng. A Survey on Advancing the DBMS Query Optimizer: Cardinality Estimation, Cost Model, and Plan Enumeration . *Data Sci. Eng.*, 6(1):86–101, 2021.

[170] Ulrich Langenbach, Stefan Wiehler, and Endric Schubert. Evaluation of a declarative Linux kernel FPGA manager for dynamic partial reconfiguration . In *2017 International Conference on FPGA Reconfiguration for General-Purpose Computing (FPGA4GPC)*, pages 13–18, 2017.

[171] Chris Lavin and Alireza Kaviani. Build Your Own Domain-specific Solutions with RapidWright: Invited Tutorial . In *FPGA*, pages 14–22. ACM, 2019.

[172] Ramon Lawrence, Erik Brandsberg, and Roland Lee. Next generation JDBC database drivers for performance, transparent caching, load balancing, and scale-out . In *SAC*, pages 915–918. ACM, 2017.

[173] Paul Le Guernic, Albert Benveniste, Patricia Bournai, and Thierry Gautier. Signal-A data flow-oriented language for signal processing. *IEEE Trans. Acoust. Speech Signal Process.*, 34(2):362–374, 1986.

[174] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proc. IEEE*, 75(9):1235–1245, 1987.

[175] Edward A. Lee, Stephen Neuendorffer, and Michael J. Wirthlin. Actor-Oriented Design of Embedded Hardware and Software Systems. *J. Circuits Syst. Comput.*, 12(3):231–260, 2003.

[176] Edward A. Lee and Alberto L. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 17(12):1217–1229, 1998.

[177] Ang Li and David Wentzlaff. PRGA: An Open-Source FPGA Research and Prototyping Framework. In *FPGA*, pages 127–137. ACM, 2021.

[178] Li Li and Hai Zhou. Structural transformation for best-possible obfuscation of sequential circuits . In *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, pages 55–60, 2013.

[179] Xiangwei Li and Douglas L. Maskell. Time-Multiplexed FPGA Overlay Architectures: A Survey. *ACM Trans. Design Autom. Electr. Syst.*, 24(5):54:1–54:19, 2019.

[180] Hang Liu and H. Howie Huang. SIMD-X: Programming and Processing of Graph Algorithms on GPUs. In *USENIX Annual Technical Conference*, pages 411–428. USENIX Association, 2019.

[181] Hao Jun Liu. Archipelago-an open source fpga with toolflow support. *Master's thesis, University of California at Berkeley*, 2014.

[182] Leibo Liu, Jianfeng Zhu, Zhaoshi Li, Yanan Lu, Yangdong Deng, Jie Han, Shouyi Yin, and Shaojun Wei. A Survey of Coarse-Grained Reconfigurable Architecture and Design: Taxonomy, Challenges, and Applications . *ACM Comput. Surv.*, 52(6):118:1–118:39, 2020.

[183] Daniela Sanchez Lopera, Lorenzo Servadei, Gamze Naz Kiprit, Souvik Hazra, Robert Wille, and Wolfgang Ecker. A Survey of Graph Neural Networks for Electronic Design Automation. In *MLCAD*, pages 1–6. IEEE, 2021.

[184] Alec Lu and Zhenman Fang. SQL2FPGA: Automatic Acceleration of SQL Query Processing on Modern CPU-FPGA Platforms. In *FCCM*, pages 184–194. IEEE, 2023.

[185] Chun-Hsien Lu, Chih-Sheng Lin, Hung-Lin Chao, Jih-Sheng Shen, and Pao-Ann Hsiung. Reconfigurable Multi-core Architecture – A Plausible Solution to the Von Neumann Performance Bottleneck . In *2013 IEEE 7th International Symposium on Embedded Multicore Socs*, pages 159–164, 2013.

[186] Garm Lucassen, Kevin van Rooij, and Slinger Jansen. Ecosystem Health of Cloud PaaS Providers. In *ICSOB*, volume 150 of *Lecture Notes in Business Information Processing*, pages 183–194. Springer, 2013.

[187] Bruce J. MacLennan. Natural computation and non-Turing models of computation. *Theor. Comput. Sci.*, 317(1-3):115–145, 2004.

[188] Hamidreza Maghsoudlou, Behrouz Afshar-Nadjafi, and Seyed Taghi Akhavan Niaki. A multi-objective invasive weeds optimization algorithm for solving multi-skill multi-mode resource constrained project scheduling problem . *Comput. Chem. Eng.*, 88:157–169, 2016.

[189] Divya Mahajan, Joon Kyung Kim, Jacob Sacks, Adel Ardalan, Arun Kumar, and Hadi Esmaeilzadeh. In-RDBMS Hardware Acceleration of Advanced Analytics. *Proc. VLDB Endow.*, 11(11):1317–1331, 2018.

[190] Kristiyan Manev. byteman, 2023. https://github.com/FPGA-Research-Manchester/byteman/.

[191] Kristiyan Manev and Dirk Koch. Large Utility Sorting on FPGAs. In *FPT*, pages 334–337. IEEE, 2018.

[192] Kristiyan Manev, Joseph Powell, Kaspar Matas, and Dirk Koch. byteman: A Bitstream Manipulation Framework. In *FPT*. IEEE, 2022.

[193] Kristiyan Manev, Anuj Vaishnav, and Dirk Koch. Unexpected Diversity: Quantitative Memory Analysis for Zynq UltraScale+ Systems . In *FPT*, pages 179–187. IEEE, 2019.

[194] Kristiyan Manev, Anuj Vaishnav, Charalampos Kritikakis, and Dirk Koch. Scalable Filtering Modules for Database Acceleration on FPGAs. In *HEART*, pages 4:1–4:6. ACM, 2019.

[195] Kristiyan Nedkov Manev. *Resource Elastic Dynamic Stream Processing on FPGAs Exemplified on Database Acceleration* . PhD thesis, The University of Manchester, Manchester, UK, 2022.

[196] Kaspar Matas. FPGA Virus scanner, 2020. https://github.com/FPGA-Research-Manchester/FPGAVirusScanner/.

[197] Kaspar Matas. OrkhestraFPGAStream, 2023. https://github.com/FPGA-Research-Manchester/OrkhestraFPGAStream/.

[198] Kaspar Matas, Tuan La, Nikola Grunchevski, Khoa Dang Pham, and Dirk Koch. Invited Tutorial: FPGA Hardware Security for Datacenters and Beyond. In *FPGA*, pages 11–20. ACM, 2020.

[199] Kaspar Matas, Tuan Minh La, Khoa Dang Pham, and Dirk Koch. Powerhammering through Glitch Amplification - Attacks and Mitigation. In *FCCM*, pages 65–69. IEEE, 2020.

[200] Kaspar Mätas, Kristiyan Manev, Joseph Powell, and Dirk Koch. Automated Generation and Orchestration of Stream Processing Pipelines on FPGAs. In *FPT*, pages 1–10. IEEE, 2022.

[201] Kaspar Mätas, Kristiyan Manev, Joseph Powell, and Dirk Koch. FPL Demo: Runtime Stream Processing with Resource-Elastic Pipelines on FPGAs. In *FPL*, page 466. IEEE, 2022.

[202] Maxeler Technologies. Maxeler App Gallery, 2020. `http://appgallery.maxeler.com`.

[203] Joel Mandebi Mbongue, Danielle Tchuinkou Kwadjo, and Christophe Bobda. Automatic Generation of Application-Specific FPGA Overlays with Rapid-Wright . In *FPT*, pages 303–306. IEEE, 2019.

[204] Xiangrui Meng, Joseph K. Bradley, Burak Yavuz, Evan R. Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, D. B. Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. MLlib: Machine Learning in Apache Spark. *J. Mach. Learn. Res.*, 17:34:1–34:7, 2016.

[205] Jan-Eike Michels, Keith W. Hare, Krishna G. Kulkarni, Calisto Zuzarte, Zhen Hua Liu, Beda Christoph Hammerschmidt, and Fred Zemke. The New and Improved SQL: 2016 Standard. *SIGMOD Rec.*, 47(2):51–60, 2018.

[206] Microsoft. Windows 11 requirements, 2022. `https://learn.microsoft.com/en-us/windows/whats-new/windows-11-requirements`.

[207] Umar Ibrahim Minhas, Roger F. Woods, Dimitrios S. Nikolopoulos, and Georgios Karakonstantis. Efficient, Dynamic Multi-Task Execution on FPGA-Based Computing Systems. *IEEE Trans. Parallel Distributed Syst.*, 33(3):710–722, 2022.

[208] Sparsh Mittal. A survey of FPGA-based accelerators for convolutional neural networks. *Neural Comput. Appl.*, 32(4):1109–1139, 2020.

[209] Takefumi Miyoshi, Hideyuki Kawashima, Yuta Terada, and Tsutomu Yoshinaga. A Coarse Grain Reconfigurable Processor Architecture for Stream Processing Engine . In *FPL*, pages 490–495. IEEE Computer Society, 2011.

[210] Azlinah Mohamed, Maryam Khanian Najafabadi, Bee Wah Yap, Ezzatul Akmal Kamaru-Zaman, and Ruhaila Maskat. The state of the art and taxonomy of big data analytics: view from new big data framework . *Artif. Intell. Rev.*, 53(2):989–1037, 2020.

[211] Alessio Montone, Marco D. Santambrogio, Donatella Sciuto, and Seda Ogrenci Memik. Placement and Floorplanning in Dynamically Reconfigurable FPGAs. *ACM Trans. Reconfigurable Technol. Syst.*, 3(4):24:1–24:34, 2010.

[212] René Müller, Jens Teubner, and Gustavo Alonso. Streams on Wires - A Query Compiler for FPGAs. *Proc. VLDB Endow.*, 2(1):229–240, 2009.

[213] René Müller, Jens Teubner, and Gustavo Alonso. Glacier: A Query-to-Hardware Compiler. In *SIGMOD Conference*, pages 1159–1162. ACM, 2010.

[214] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. A Modern Primer on Processing in Memory. *CoRR*, abs/2012.03112, 2020.

[215] N. Voss et al. Towards Real Time Radiotherapy Simulation. In *ASAP*, pages 173–180. IEEE, 2019.

[216] Lekshmi Beena Gopalakrishnan Nair. *Capability Aware Query Optimizer For An FPGA-Based Near-Data Processor*. PhD thesis, University of Erlangen-Nuremberg, Germany, 2021.

[217] Lekshmi Beena Gopalakrishnan Nair, Andreas Becher, Stefan Wildermann, Klaus Meyer-Wegener, and Jürgen Teich. Speculative Dynamic Reconfiguration and Table Prefetching Using Query Look-Ahead in the ReProVide Near-Data-Processing System . *Datenbank-Spektrum*, 21(1):55–64, 2021.

[218] Walid A. Najjar, Edward A. Lee, and Guang R. Gao. Advances in the dataflow computational model. *Parallel Comput.*, 25(13-14):1907–1929, 1999.

[219] Nallatech. OpenCAPI Enabled FPGAs The perfect bridge to a Data Centric World, 2018. `https://openpowerfoundation.org/wp-content/uploads/2018/10/Allan-Cantle.Nallatech-Presentation-2018-OPF-Summit%5FAmsterdam-presentation.pdf/`.

[220] Kevin Nam, Blair Fort, and Stephen Dean Brown. FISH: Linux system calls for FPGA accelerators. In *FPL*, pages 1–4. IEEE, 2017.

[221] Jean-François Nezan, Nicolas Siret, Matthieu Wipliez, Francesca Palumbo, and Luigi Raffo. Multi-purpose systems: A novel dataflow-based generation and mapping strategy . In *ISCAS*, pages 3073–3076. IEEE, 2012.

[222] Tuan D. A. Nguyen and Akash Kumar. PRFloor: An Automatic Floorplanner for Partially Reconfigurable FPGA Systems . In *FPGA*, pages 149–158. ACM, 2016.

[223] Andreas Oetken, Stefan Wildermann, Jürgen Teich, and Dirk Koch. A Bus-Based SoC Architecture for Flexible Module Placement on Reconfigurable FPGAs. In *FPL*, pages 234–239. IEEE Computer Society, 2010.

[224] Oracle. Oracle Machine Learning, 2019. `https://www.oracle.com/database/technologies/datawarehouse-bigdata/machine-learning.html`.

[225] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems, 4th Edition*. Springer, 2020.

[226] Pekka Pääkkönen and Daniel Pakkala. Reference Architecture and Classification of Technologies, Products and Services for Big Data Systems . *Big Data Res.*, 2(4):166–186, 2015.

[227] Marco Pagani, Alessandro Biondi, Mauro Marinoni, Lorenzo Molinari, Giuseppe Lipari, and Giorgio Carlo Buttazzo. A Linux-based support for developing real-time applications on heterogeneous platforms with dynamic FPGA reconfiguration . *Future Gener. Comput. Syst.*, 129:125–140, 2022.

[228] Francesca Palumbo, Nicola Carta, Danilo Pani, Paolo Meloni, and Luigi Raffo. The multi-dataflow composer tool: generation of on-the-fly reconfigurable platforms . *J. Real Time Image Process.*, 9(1):233–249, 2014.

[229] Kyprianos Papadimitriou, Apostolos Dollas, and Scott Hauck. Performance of partial reconfiguration in FPGA systems: A survey and a cost model . *ACM Trans. Reconfigurable Technol. Syst.*, 4(4):36:1–36:24, 2011.

[230] Philippos Papaphilippou and Wayne Luk. Accelerating Database Systems Using FPGAs: A Survey. In *FPL*, pages 125–130. IEEE Computer Society, 2018.

[231] Dongjoon Park, Yuanlong Xiao, Nevo Magnezi, and André DeHon. Case for Fast FPGA Compilation Using Partial Reconfiguration. In *FPL*, pages 235–238. IEEE Computer Society, 2018.

[232] Cesare Pautasso and Gustavo Alonso. Parallel computing patterns for Grid workflows. In *2006 Workshop on Workflows in Support of Large-Scale Science*, pages 1–10, 2006.

[233] Robert Pellerin, Nathalie Perrier, and Francois Berthaut. A survey of hybrid metaheuristics for the resource-constrained project scheduling problem . *Eur. J. Oper. Res.*, 280(2):395–416, 2020.

[234] Johan Peltenburg, Ákos Hadnagy, Matthijs Brobbel, Robert Morrow, and Zaid Al-Ars. Tens of gigabytes per second JSON-to-Arrow conversion with FPGA accelerators . In *FPT*, pages 1–9. IEEE, 2021.

[235] Johan Peltenburg, Lars T. J. van Leeuwen, Joost Hoozemans, Jian Fang, Zaid Al-Ars, and H. Peter Hofstee. Battling the CPU Bottleneck in Apache Parquet to Arrow Conversion Using FPGA . In *FPT*, pages 281–286. IEEE, 2020.

[236] Johan Peltenburg, Jeroen van Straten, Matthijs Brobbel, Zaid Al-Ars, and H. Peter Hofstee. Tydi: An Open Specification for Complex Data Structures Over Hardware Streams . *IEEE Micro*, 40(4):120–130, 2020.

[237] Johan Peltenburg, Jeroen van Straten, Lars Wijtemans, Lars van Leeuwen, Zaid Al-Ars, and H. Peter Hofstee. Fletcher: A Framework to Efficiently Integrate FPGA Accelerators with Apache Arrow . In *FPL*, pages 270–277. IEEE, 2019.

[238] Khoa Dang Pham, Edson L. Horta, and Dirk Koch. BITMAN: A tool and API for FPGA bitstream manipulations. In *DATE*, pages 894–897. IEEE, 2017.

[239] Khoa Dang Pham, Edson L. Horta, Dirk Koch, Anuj Vaishnav, and Thomas Kuhn. IPRDF: An Isolated Partial Reconfiguration Design Flow for Xilinx FPGAs. In *MCSoC*, pages 36–43. IEEE Computer Society, 2018.

[240] Khoa Dang 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 *FPT*, pages 221–226. IEEE, 2020.

[241] 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, 2018.

[242] Phi-Hung Pham, Darko Jelaca, Clément Farabet, Berin Martini, Yann LeCun, and Eugenio Culurciello. NeuFlow: Dataflow vision processing system-on-a-chip. In *MWSCAS*, pages 1044–1047. IEEE, 2012.

[243] David Phillips. tpch-dbgen, 2011. `https://github.com/electrum/tpch-dbgen/`.

[244] Frank Plasencia-Balabarca, Edward Mitacc-Meza, Mario Raffo-Jara, and Carlos Silva Cárdenas. A Flexible UVM-Based Verification Framework Reusable with Avalon, AHB, AXI and Wishbone Bus Interfaces for an AES Encryption Module . In *LATS*, pages 1–4. IEEE, 2019.

[245] Christian Plessl and Marco Platzner. Virtualization of Hardware - Introduction and Survey. In *ERSA*, pages 63–69. CSREA Press, 2004.

[246] Artur Podobas, Kentaro Sano, and Satoshi Matsuoka. A Survey on Coarse-Grained Reconfigurable Architectures From a Performance Perspective . *IEEE Access*, 8:146719–146743, 2020.

[247] Meikel Pöss, Raghunath Othayoth Nambiar, and David Walrath. Why You Should Run TPC-DS: A Workload Analysis. In *VLDB*, pages 1138–1149. ACM, 2007.

[248] Meikel Pöss and Dmitry Potapov. Data Compression in Oracle. In *VLDB*, pages 937–947. Morgan Kaufmann, 2003.

[249] Joseph Powell, Kaspar Matas, Kristiyan Manev, and Dirk Koch. FPL Demo: FPGA Bitstream Virus Scanning. In *FPL*, page 469. IEEE, 2022.

[250] I. Puja, P. Poscic, and D. Jaksic. Overview and Comparison of Several elational Database Modelling Metodologies and Notations . In *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1641–1646, 2019.

[251] Nitin Pundir, Fahim Rahman, Farimah Farahmandi, and Mark M. Tehranipoor. What is All the FaaS About? - Remote Exploitation of FPGA-as-a-Service Platforms . *IACR Cryptol. ePrint Arch.*, page 746, 2021.

[252] Murad Qasaimeh, Kristof Denolf, Jack Lo, Kees A. Vissers, Joseph Zambreno, and Phillip H. Jones. Comparing Energy Efficiency of CPU, GPU and FPGA Implementations for Vision Kernels . In *ICESS*, pages 1–8. IEEE, 2019.

[253] Xiaoke Qin, Chetan Muthry, and Prabhat Mishra. Decoding-Aware Compression of FPGA Bitstreams. *IEEE Trans. Very Large Scale Integr. Syst.*, 19(3):411–419, 2011.

[254] Masudul Hassan Quraishi, Erfan Bank Tavakoli, and Fengbo Ren. A Survey of System Architectures and Techniques for FPGA Virtualization. *TPDS*, 32(9):2216–2230, 2021.

[255] Marco Rabozzi, Gianluca Carlo Durelli, Antonio Miele, John Lillis, and Marco Domenico Santambrogio. Floorplanning Automation for Partial-Reconfigurable FPGAs via Feasible Placements Generation . *IEEE Trans. Very Large Scale Integr. Syst.*, 25(1):151–164, 2017.

[256] Humyun Fuad Rahman, Ripon K. Chakrabortty, and Michael J. Ryan. Managing Uncertainty and Disruptions in Resource Constrained Project Scheduling Problems: A Real-Time Reactive Approach . *IEEE Access*, 9:45562–45586, 2021.

[257] Sunita Ramagond, Siva Yellampalli, and C Kanagasabapathi. A review and analysis of communication logic between PL and PS in ZYNQ AP SoC . In *2017 International Conference On Smart Technologies For Smart Nation (SmartTechCon)*, pages 946–951, 2017.

[258] Vijayshankar Raman and Garret Swart. How to Wring a Table Dry: Entropy Compression of Relations and Querying of Compressed Relations . In *VLDB*, pages 858–869. ACM, 2006.

[259] Chethan Ramesh, Shivukumar B. Patil, Siva Nishok Dhanuskodi, George Provelengios, Sébastien Pillement, Daniel E. Holcomb, and Russell Tessier. FPGA Side Channel Attacks without Physical Access. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 45–52. IEEE, 2018.

[260] Ahmed E. Abdel Raouf, Nagwa L. Badr, and Mohamed Fahmy Tolba. Dynamic Distributed Database over Cloud Environment. In *AMLTA*, volume 488 of *Communications in Computer and Information Science*, pages 67–76. Springer, 2014.

[261] S Ravi and M Joseph. Open source HLS tools: A stepping stone for modern electronic CAD. In *2016 IEEE International Conference on Computational Intelligence and Computing Research (ICCIC)*, pages 1–8. IEEE, 2016.

[262] Federico Reghenzani, Giuseppe Massari, and William Fornaciari. The Real-Time Linux Kernel: A Survey on PREEMPT_RT. *ACM Comput. Surv.*, 52(1):18:1–18:36, 2019.

[263] Jens Rettkowski, Konstantin Friesen, and Diana Göhringer. RePaBit: Automated generation of relocatable partial bitstreams for Xilinx Zynq FPGAs . In *ReConFig*, pages 1–8. IEEE, 2016.

[264] Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi, and Jeremy Kepner. Survey and Benchmarking of Machine Learning Accelerators. In *HPEC*, pages 1–9. IEEE, 2019.

[265] Nuno Roma and Leonel Sousa. A tutorial overview on the properties of the discrete cosine transform for encoded image and video processing . *Signal Process.*, 91(11):2443–2464, 2011.

[266] Johannes Romoth, Mario Porrmann, and Ulrich Rückert. Survey of FPGA applications in the period 2000–2015. *Technical Report*, 2017.

[267] Mohammad Sadoghi, Rija Javed, Naif Tarafdar, Harsh Singh, Rohan Palaniappan, and Hans-Arno Jacobsen. Multi-query Stream Processing on FPGAs. In *ICDE*, pages 1229–1232. IEEE Computer Society, 2012.

[268] MohammadSadegh Sadri, Christian Weis, Norbert Wehn, and Luca Benini. Energy and performance exploration of accelerator coherency port using Xilinx ZYNQ . In *FPGAworld*, pages 5:1–5:8. ACM, 2013.

[269] Behzad Salami, Gorker Alp Malazgirt, Oriol Arcas-Abella, Arda Yurdakul, and Nehir Sönmez. AxleDB: A Novel Programmable Query Processing Platform on FPGA. *Microprocess. Microsystems*, 51:142–164, 2017.

[270] Khodakaram Salimifard and Mike Wright. Petri net-based modelling of workflow systems: An overview. *Eur. J. Oper. Res.*, 134(3):664–676, 2001.

[271] Ilaria Scarabottolo, Giovanni Ansaloni, George A. Constantinides, Laura Pozzi, and Sherief Reda. Approximate Logic Synthesis: A Survey. *Proc. IEEE*, 108(12):2195–2213, 2020.

[272] Dominik Scheinert, Houkun Zhu, Lauritz Thamsen, Morgan K. Geldenhuys, Jonathan Will, Alexander Acker, and Odej Kao. Enel: Context-Aware Dynamic Scaling of Distributed Dataflow Jobs using Graph Propagation . In *IPCCC*, pages 1–8. IEEE, 2021.

[273] Falk Schellenberg, Dennis R. E. Gnad, Amir Moradi, and Mehdi Baradaran Tahoori. An Inside Job: Remote Power analysis Attacks on FPGAs. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1111–1116. IEEE, 2018.

[274] Frank Schultmann and Otto Rentz. Environment-oriented project scheduling for the dismantling of buildings. *OR Spectr.*, 23(1):51–78, 2001.

[275] Javier Serrano. Introduction to FPGA design. *CAS - CERN Accelerator School: Digital Signal Processing*, pages 231–247, 2008.

[276] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, and Christopher Berner. Presto: SQL on Everything. In *ICDE*, pages 1802–1813. IEEE, 2019.

[277] David Shah, Eddie Hung, Clifford Wolf, Serge Bazanski, Dan Gisselquist, and Miodrag Milanovic. Yosys+nextpnr: An Open Source Framework from Verilog to Bitstream for Commercial FPGAs . In *FCCM*, pages 1–4. IEEE, 2019.

[278] Hafsah Shahzad, Ahmed Sanaullah, and Martin C. Herbordt. Survey and Future Trends for FPGA Cloud Architectures. In *HPEC*, pages 1–10. IEEE, 2021.

[279] Eugene Sharygin, Ruben Buchatskiy, Roman Zhuykov, and Arseny Sher. Runtime Specialization of PostgreSQL Query Executor. In *Ershov Informatics Conference*, volume 10742 of *Lecture Notes in Computer Science*, pages 375–386. Springer, 2017.

[280] David Sidler, Muhsen Owaida, Zsolt István, Kaan Kara, and Gustavo Alonso. doppioDB: A Hardware Accelerated Database. In *FPL*, page 1. IEEE, 2017.

[281] Raj Mohan Singh, Lalit Kumar Awasthi, and Geeta Sikka. Towards Metaheuristic Scheduling Techniques in Cloud and Fog: An Extensive Taxonomic Review . *ACM Comput. Surv.*, 55(3):50:1–50:43, 2023.

[282] Rym Skhiri, Virginie Fresse, Jean-Paul Jamont, Benoît Suffran, and Jihene Malek. From FPGA to Support Cloud to Cloud of FPGA: State of the Art. *Int. J. Reconfigurable Comput.*, 2019:8085461:1–8085461:17, 2019.

[283] Hayden Kwok-Hay So and Robert W. Brodersen. Improving Usability of FPGA-Based Reconfigurable Computers Through Operating System Support . In *FPL*, pages 1–6. IEEE, 2006.

[284] Hayden Kwok-Hay So and Cheng Liu. FPGA overlays. In *FPGAs for Software Programmers*, pages 285–305. Springer, 2016.

[285] Ioannis Sourdis, João Bispo, João M. P. Cardoso, and Stamatis Vassiliadis. Regular Expression Matching in Reconfigurable Hardware. *J. Signal Process. Syst.*, 51(1):99–121, 2008.

[286] Statista. Volume of data/information created worldwide from 2010 to 2025 (in zetabytes) , 2021. `https://www.statista.com/statistics/871513/worldwide-data-created/`.

[287] Michael Stonebraker, Ugur Çetintemel, and Stanley B. Zdonik. The 8 Requirements of Real-time Stream Processing. *SIGMOD Rec.*, 34(4):42–47, 2005.

[288] Edward A. Stott, Justin S. J. Wong, N. Pete Sedcole, and Peter Y. K. Cheung. Degradation in FPGAs: Measurement and Modelling. In *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '10, page 229–238, New York, NY, USA, 2010. Association for Computing Machinery.

[289] Yuri Stoyan and Tatiana Romanova. Mathematical Models of Placement Optimisation: Two-and Three-dimensional Problems and Applications . In *Modeling and optimization in space engineering*, pages 363–388. Springer, 2012.

[290] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and Policy Considerations for Modern Deep Learning Research. In *AAAI*, pages 13693–13696. AAAI Press, 2020.

[291] Toshio Suganuma, Takeshi Ogasawara, Mikio Takeuchi, Toshiaki Yasue, Motohiro Kawahito, Kazuaki Ishizaki, Hideaki Komatsu, and Toshio Nakatani. Overview of the IBM Java Just-in-Time Compiler. *IBM Syst. J.*, 39(1):175–193, 2000.

[292] Young-Kyoon Suh, Seounghyeon Kim, Joo-Young Lee, Hawon Chu, Jun Young An, and Kyong-Ha Lee. An Experimental Study across GPU DBMSes toward Cost-Effective Analytical Processing . *IEICE Trans. Inf. Syst.*, 104-D(5):551–555, 2021.

[293] Swarm64. Swarm 64 DA Technical Overview, 2021. https://swarm64.com/wp-content/uploads/2021/05/Swarm64-V5-Technical-Overview.pdf.

[294] Russell Tessier, Kenneth L. Pocek, and André DeHon. Reconfigurable Computing Architectures. *Proc. IEEE*, 103(3):332–354, 2015.

[295] Lauritz Thamsen, Ilya Verbitskiy, Jossekin Beilharz, Thomas Renner, Andreas Polze, and Odej Kao. Ellis: Dynamically Scaling Distributed Dataflows to Meet Runtime Targets. In *CloudCom*, pages 146–153. IEEE Computer Society, 2017.

[296] The Kernel Development Community. FPGA Manager Documentation in the Linux Kernel, 2023. https://www.kernel.org/doc/html/latest/driver-api/fpga/fpga-mgr.html.

[297] Thomas N. Theis and H.-S. Philip Wong. The End of Moore's Law: A New Beginning for Information Technology. *Comput. Sci. Eng.*, 19(2):41–50, 2017.

[298] Jason G Tong, Ian DL Anderson, and Mohammed AS Khalid. Soft-core processors for embedded systems. In *2006 International Conference on Microelectronics*, pages 170–173. IEEE, 2006.

[299] Matheus F. Torquato and Marcelo A. C. Fernandes. High-Performance Parallel Implementation of Genetic Algorithm on FPGA. *Circuits Syst. Signal Process.*, 38(9):4014–4039, 2019.

[300] Nemanja Trifunovic, Boris Perovic, Petar Trifunovic, Zoran Babovic, and Ali R. Hurson. Chapter Five - A Novel Infrastructure for Synergistic Dataflow Research, Development, Education, and Deployment: The Maxeler AppGallery Project . *Adv. Comput.*, 106:167–213, 2017.

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

[302] Anuj Vaishnav, Khoa Dang Pham, and Dirk Koch. A Survey on FPGA Virtualization. In *FPL*, pages 131–138. IEEE Computer Society, 2018.

[303] Anuj Vaishnav, Khoa Dang Pham, and Dirk Koch. Heterogeneous Resource-Elastic Scheduling for CPU+FPGA Architectures. In *HEART*, pages 1:1–1:6. ACM, 2019.

[304] Anuj Vaishnav, Khoa Dang Pham, Dirk Koch, and James Garside. Resource Elastic Virtualization for FPGAs Using OpenCL. In *FPL*, pages 111–118. IEEE Computer Society, 2018.

[305] Anuj Vaishnav, Khoa Dang Pham, Joseph Powell, and Dirk Koch. FOS: A Modular FPGA Operating System for Dynamic Workloads. *ACM TRETS.*, 13(4):20:1–20:28, 2020.

[306] Vishal A. Varma, Reha Uzsoy, Joseph F. Pekny, and Gary E. Blau. Lagrangian heuristics for scheduling new product development projects in the pharmaceutical industry . *J. Heuristics*, 13(5):403–433, 2007.

[307] Michalis Vavouras, Kyprianos Papadimitriou, and Ioannis Papaefstathiou. High-speed FPGA-based implementations of a Genetic Algorithm. In *ICSAMOS*, pages 9–16. IEEE, 2009.

[308] Malte Vesper. *Dynamic streamprocessing pipelines on FPGAs targeting database acceleration* . PhD thesis, The University of Manchester, Manchester, UK, 2019.

[309] Mário P. Véstias and Horácio C. Neto. Trends of CPU, GPU and FPGA for high-performance computing. In *FPL*, pages 1–6. IEEE, 2014.

[310] Jorge Vieira, Jorge Bernardino, and Henrique Madeira. Efficient Compression of Text Attributes of Data Warehouse Dimensions. In *DaWaK*, volume 3589 of *Lecture Notes in Computer Science*, pages 356–367. Springer, 2005.

[311] Kizheppatt Vipin and Suhaib A. Fahmy. ZyCAP: Efficient Partial Reconfiguration Management on the Xilinx Zynq. *IEEE Embed. Syst. Lett.*, 6(3):41–44, 2014.

[312] Kizheppatt Vipin and Suhaib A. Fahmy. FPGA Dynamic and Partial Reconfiguration: A Survey of Architectures, Methods, and Applications . *ACM Comput. Surv.*, 51(4):72:1–72:39, 2018.

[313] Herbert Walder and Marco Platzner. A Runtime Environment for Reconfigurable Hardware Operating Systems. In *FPL*, volume 3203 of *Lecture Notes in Computer Science*, pages 831–835. Springer, 2004.

[314] Yi-Qing Wang. An Analysis of the Viola-Jones Face Detection Algorithm. *Image Process. Line*, 4:128–148, 2014.

[315] Ze-ke Wang, Johns Paul, Hui Yan Cheah, Bingsheng He, and Wei Zhang. Relational Query Processing on OpenCL-based FPGAs. In *FPL*, pages 1–10. IEEE, 2016.

[316] Andrew Waterman. *Design of the RISC-V Instruction Set Architecture*. PhD thesis, University of California, Berkeley, USA, 2016.

[317] Fadi Wedyan and Somia Abufakher. Impact of design patterns on software quality: a systematic literature review . *IET Softw.*, 14(1):1–17, 2020.

[318] Jagath Weerasinghe, François Abel, Christoph Hagleitner, and Andreas Herkersdorf. Enabling FPGAs in Hyperscale Data Centers. In *UIC/ATC/ScalCom*, pages 1078–1086. IEEE Computer Society, 2015.

[319] Jan Weglarz, Joanna Józefowska, Marek Mika, and Grzegorz Waligóra. Project scheduling with finite or infinite number of activity processing modes - A survey . *Eur. J. Oper. Res.*, 208(3):177–205, 2011.

[320] Dawei Wei, Huansheng Ning, Feifei Shi, Yueliang Wan, Jiabo Xu, Shunkun Yang, and Li Zhu. Dataflow Management in the Internet of Things: Sensing, Control, and Security . *Tsinghua Science and Technology*, 26(6):918–930, 2021.

[321] Kyu-Young Whang, Il-Yeol Song, Taek-Yoon Kim, and Ki-Hoon Lee. The ubiquitous DBMS. *SIGMOD Rec.*, 38(4):14–22, 2009.

[322] Mark A. Will and Ryan K. L. Ko. Secure FPGA as a Service - Towards Secure Data Processing by Physicalizing the Cloud . In *TrustCom/BigDataSE/ICESS*, pages 449–455. IEEE Computer Society, 2017.

[323] Remigiusz Wisniewski. Dynamic Partial Reconfiguration of Concurrent Control Systems Specified by Petri Nets and Implemented in Xilinx FPGA Devices . *IEEE Access*, 6:32376–32391, 2018.

[324] Louis Woods, Zsolt István, and Gustavo Alonso. Ibex - An Intelligent Storage Engine with Support for Advanced SQL Off-loading . *Proc. VLDB Endow.*, 7(11):963–974, 2014.

[325] Lisa Wu, Andrea Lottarini, Timothy K. Paine, Martha A. Kim, and Kenneth A. Ross. The Q100 Database Processing Unit. *IEEE Micro*, 35(3):34–46, 2015.

[326] Maria Xekalaki, Juan Fumero, Athanasios Stratikopoulos, Katerina Doka, Christos Katsakioris, Constantinos Bitsakos, Nectarios Koziris, and Christos Kotselidis. Enabling Transparent Acceleration of Big Data Frameworks using Heterogeneous Hardware. *Proc. VLDB Endow.*, 15(13):3869–3882, 2022.

[327] Xelera. About Xelera - Developer of a big data analytics platform, 2021. `https://www.accelize.com/blog-welcome-to-xelera`.

[328] Yuanlong Xiao, Aditya Hota, Dongjoon Park, and Andre DeHon. HiPR: High-level Partial Reconfiguration for Fast Incremental FPGA Compilation . In *FPL*. IEEE, 2022.

[329] Xilinx. GQE Kernel Acceleration Demo, 2020. `xilinx.github.io/Vitis%5FLibraries/database/2020.1/gqe%5Fguide/kernel%5Fdemo/`.

[330] Xilinx. PetaLinux Tools Documentation: Reference Guide (UG1144), 2022. `https://docs.xilinx.com/r/en-US/ug1144-petalinux-tools-reference-guide`.

[331] Xilinx. UltraScale Architecture Configuration User Guide (UG570), 2022. `https://docs.xilinx.com/v/u/en-US/ug570-ultrascale-configuration`.

[332] Xilinx. Xilinx's Linux kernel, 2023. `https://github.com/Xilinx/linux-xlnx/blob/xilinx-v2018.3/drivers/fpga/zynqmp-fpga.c`.

[333] Xilinx. Zynq UltraScale+ Device Technical Reference Manual (UG1085), 2023. `https://docs.xilinx.com/r/en-US/ug1085-zynq-ultrascale-trm/Configuration-Security-Unit-CSU-Introduction`.

[334] Chengcheng Xu, Shuhui Chen, Jinshu Su, Siu-Ming Yiu, and Lucas Chi Kwong Hui. A Survey on Regular Expression Matching for Deep Packet Inspection: Applications, Algorithms, and Hardware Platforms . *IEEE Commun. Surv. Tutorials*, 18(4):2991–3029, 2016.

[335] Mingfu Xue, Chongyan Gu, Weiqiang Liu, Shichao Yu, and Máire O'Neill. Ten years of hardware Trojans: a survey from the attacker's perspective. *IET Comput. Digit. Tech.*, 14(6):231–246, 2020.

[336] Shaon Yousuf and Ann Gordon-Ross. An Automated Hardware/Software Co-Design Flow for Partially Reconfigurable FPGAs . In *ISVLSI*, pages 30–35. IEEE Computer Society, 2016.

[337] Hoyoung Yu, Hyung-Min Lee, Youngjoo Shin, and Youngmin Kim. FPGA reverse engineering in Vivado design suite based on X-ray project. In *ISOCC*, pages 239–240. IEEE, 2019.

[338] Michael Xi Yue, Dirk Koch, and Guy G. F. Lemieux. Rapid Overlay Builder for Xilinx FPGAs. In *FCCM*, pages 17–20. IEEE Computer Society, 2015.

[339] Rafael Zamacola, Alberto García-Martínez, Javier Mora, Andrés Otero, and Eduardo de la Torre. IMPRESS: Automated Tool for the Implementation of Highly Flexible Partial Reconfigurable Systems with Xilinx Vivado . In *ReConFig*, pages 1–8. IEEE, 2018.

[340] Shaza Zeitouni, Jo Vliegen, Tommaso Frassetto, Dirk Koch, Ahmad-Reza Sadeghi, and Nele Mentens. Trusted Configuration in Cloud FPGAs. In *FCCM*, pages 233–241. IEEE, 2021.

[341] Yue Zha and Jing Li. Hetero-ViTAL: A Virtualization Stack for Heterogeneous FPGA Clusters. In *ISCA*, pages 470–483. IEEE, 2021.

[342] Jinyu Zhan, Wei Jiang, Ying Li, Junting Wu, Jianping Zhu, and Jinghuan Yu. Accelerating Queries of Big Data Systems by Storage-Side CPU-FPGA Co-Design . *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 41(7):2128–2141, 2022.

[343] Feng Zhang, Weitao Wan, Chenyang Zhang, Jidong Zhai, Yunpeng Chai, Haixiang Li, and Xiaoyong Du. CompressDB: Enabling Efficient Compressed Data Direct Processing for Various Databases . In *SIGMOD Conference*, pages 1655–1669. ACM, 2022.

[344] Yansong Zhang, Yu Zhang, Jiaheng Lu, Shan Wang, Zhuan Liu, and Ruichen Han. One size does not fit all: accelerating OLAP workloads with GPUs. *Distributed Parallel Databases*, 38(4):995–1037, 2020.

[345] Guanwen Zhong, Vanchinathan Venkataramani, Yun Liang, Tulika Mitra, and Smaïl Niar. Design space exploration of multiple loops on FPGAs using high level synthesis . In *ICCD*, pages 456–463. IEEE Computer Society, 2014.

[346] Zheng Zhou, Chung-Ching Shen, William Plishker, and Shuvra S. Bhattacharyya. Dataflow-Based, Cross-Platform Design Flow for DSP Applications. In *Embedded Systems Development, From Functional Models to Implementations*, pages 41–65. Springer, 2014.

[347] Kenneth M. Zick and John P. Hayes. Low-cost Sensing with Ring Oscillator Arrays for Healthier Reconfigurable Systems . *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 5(1):1,26, 2012-03-01.

[348] 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 Trans. Reconfigurable Technol. Syst.*, 9(4):25:1–25:24, 2016.

[349] Xingqi Zou, Sheng Xu, Xiaoming Chen, Liang Yan, and Yinhe Han. Breaking the von Neumann bottleneck: architecture-level processing-in-memory technology . *Sci. China Inf. Sci.*, 64(6), 2021.

[350] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter A. Boncz. Super-Scalar RAM-CPU Cache Compression. In *ICDE*, page 59. IEEE Computer Society, 2006.

[351] Loïc Zussa, Jean-Max Dutertre, Jessy Clédière, and Assia Tria. Power supply glitch induced faults on FPGA: An in-depth analysis of the injection mechanism . In *2013 IEEE 19th International On-Line Testing Symposium (IOLTS)*, pages 110–115, 2013.

Appendix A

Partially reconfigurable modules' resource requirements

In order to find any general conclusions about resource overheads associated with compiling dataflow pipelines with PR modules when compared to synthesising the pipelines from combined RTL code, a more in-depth examination is required by comparing a substantial number of different module combinations. Furthermore, meanwhile, each pipeline and module itself should be built with different levels of constraints and design choices. Therefore, such an in-depth examination of the HW aspect of building dynamic dataflow systems is out of the scope of this thesis. First and foremost, we designed modules to achieve a functional dynamic PR system capable of running at 300Mhz with 512-bit wide datapaths. Then they were implemented in smallest possible bounding boxes on the Pareto-front as described by Manev [195]. Hence, here we note the number of available resources in that bounding box and how many of these were used in the implementation. Optimising the modules for faster or larger systems or using fewer resources is left for future work. Nevertheless, for reference, this appendix shows our modules' and pipelines' resource costs in both the image processing (Section A.1) and data analytics (Section A.2) workloads.

A.1 Image processing

For image processing modules, the difference between using the two PR modules separately or synthesising them together while using the same sized bounding box is an increase in LUT usage $\approx 8\%$ and in register usage $\approx 13\%$. However, in general, CLB

Table A.1: Black and white converter module resource requirements.

Resource type	Used count	Available	Used precentage
CLB LUTs	1789	3840	46.59
CLB Registers	1203	7680	15.66

Table A.2: Sobel module resource requirements.

Resource type	Used count	Available	Used precentage
CLB LUTs	6084	10560	57.61
CLB Registers	4751	21120	22.5
Block RAM Tile	15	48	31.25
DSPs	0	144	0

Table A.3: Sobel & black and white converter module resource requirements.

Resource type	Used count	Available	Used precentage
CLB LUTs	7263	14400	50.44
CLB Registers	5190	28800	18.02
Block RAM Tile	15	48	31.25
DSPs	0	144	0

usage is only higher $\approx 4\%$. The exact numbers for the individual modules are in Tables A.1 and A.2 and for the static pipeline in Table A.3.

A.2 Data analytics

For data analytics modules, we had to use two different static pipelines for the targeted acceleration requests, and as the required combinations of modules used a larger area than was available in the PRR, the static pipelines had to use the resources more densely. While the synthesis has to put more logic into a smaller area, the resulting bitstreams are also more optimised. However, on the contrary, as referenced in the introduction, for the tools, the optimisation task becomes significantly more challenging with larger designs.

Therefore, after constructing the first static pipeline, we can see that the difference between using the corresponding filter, adder, multiplier, aggregation and linear sort modules separately or synthesising them together is relatively tiny as only mainly the CLB usage is significantly higher ($\approx 5\%$).

In the case of the second pipeline, the difference between using the 64-way merge

Table A.4: Filter module capable of processing 8 DNF clauses with 1 comparator unit resource requirements.

Resource type	Used count	Available	Used precentage
CLB LUTs	2670	6720	39.73
CLB Registers	3182	13440	23.68
DSPs	0	96	0

Table A.5: Filter module capable of processing 16 DNF clauses with 2 comparator unit resource requirements.

Resource type	Used count	Available	Used precentage
CLB LUTs	4066	7680	52.94
CLB Registers	3436	15360	22.37
DSPs	0	96	0

Table A.6: Filter module capable of processing 32 DNF clauses with 4 comparator unit resource requirements.

Resource type	Used count	Available	Used precentage
CLB LUTs	7580	14400	52.64
CLB Registers	3923	28800	13.62
Block RAM Tile	0	48	0
DSPs	0	144	0

Table A.7: Aggregate sum module resource requirements.

Resource type	Used count	Available	Used precentage
CLB LUTs	856	4800	17.83
CLB Registers	1729	9600	18.01
DSPs	0	48	0

sorter, merge join, large filter and aggregation PR modules separately or synthesising them together increased $\approx 5\%$ in LUT usage and $\approx 2\%$ in register usage. In this case, the CLB usage is higher $\approx 8\%$.

As for image processing bitstreams, the detailed resource cost values for the data analytics modules and pipelines are available in the rest of the tables.

Table A.8: Multiplier module resource requirements.

Resource type	Used count	Available	Used precentage
CLB LUTs	3813	9600	39.72
CLB Registers	8439	19200	43.95
Block RAM Tile	0	48	0
DSPs	32	96	33.33

Table A.9: Arithmetic adder module resource requirements.

Resource type	Used count	Available	Used precentage
CLB LUTs	1598	6720	23.78
CLB Registers	2792	13440	20.77
DSPs	0	48	0

Table A.10: Linear sorter module resource requirements that can sort up to 512 records per output sequence.

Resource type	Used count	Available	Used precentage
CLB LUTs	9418	14400	65.4
CLB Registers	8979	28800	31.18
Block RAM Tile	35.5	48	73.96
DSPs	0	144	0

Table A.11: Linear sorter module resource requirements that can sort up to 1024 records per output sequence.

Resource type	Used count	Available	Used precentage
CLB LUTs	13852	20160	68.71
CLB Registers	10563	40320	26.2
Block RAM Tile	70.5	72	97.92
DSPs	0	192	0

Table A.12: Join module resource requirements.

Resource type	Used count	Available	Used precentage
CLB LUTs	5311	14400	36.88
CLB Registers	5142	28800	17.85
Block RAM Tile	32	48	66.67
DSPs	0	144	0

Table A.13: Merge sorter module resource requirements that can merge up to 32 sequences into a single sorted sequence.

Resource type	Used count	Available	Used precentage
CLB LUTs	5729	10560	54.25
CLB Registers	7061	21120	33.43
Block RAM Tile	18	24	75
DSPs	0	96	0

Table A.14: Merge sorter module resource requirements that can merge up to 64 sequences into a single sorted sequence.

Resource type	Used count	Available	Used precentage
CLB LUTs	6232	14400	43.28
CLB Registers	7427	28800	25.79
Block RAM Tile	38	48	79.17
DSPs	0	144	0

Table A.15: Merge sorter module resource requirements that can merge up to 128 sequences into a single sorted sequence.

Resource type	Used count	Available	Used precentage
CLB LUTs	7047	16320	43.18
CLB Registers	7825	32640	23.97
Block RAM Tile	70.5	72	97.92
DSPs	0	192	0

Table A.16: Data analytics static pipeline which consists of the following modules: large filter, adder, multiplier, aggregation, and small linear sorter.

Resource type	Used count	Available	Used precentage
CLB LUTs	23192	43200	53.7
CLB Registers	25986	86400	30.1
Block RAM Tile	35.5	144	24.7
DSPs	32	432	7.4

Table A.17: Data analytics static pipeline which consists of the following modules: medium sorter, join, large filter, aggregation

Resource type	Used count	Available	Used precentage
CLB LUTs	18944	43200	43.9
CLB Registers	17808	86400	20.6
Block RAM Tile	70	144	48.6
DSPs	0	432	0