INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN


**Dissertation**
an der Fakultät für Mathematik, Informatik und Statistik
der Ludwig-Maximilians-Universität München

# Task Scheduling on FPGA-based Accelerators with Limited Partial Reconfiguration

vorgelegt von

Pascal Jungblut

*20.04.2024*

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN


**Dissertation**
an der Fakultät für Mathematik, Informatik und Statistik
der Ludwig-Maximilians-Universität München

# Task Scheduling on FPGA-based Accelerators with Limited Partial Reconfiguration

vorgelegt von

Pascal Jungblut

*20.04.2024*

| | |
|---|---|
| Erstberichterstatter: | Prof. Dr. Dieter Kranzlmüller |
| Zweitberichterstatter: | Prof. Dr. Martin Schreiber |
| Tag der mündlichen Prüfung: | 24.07.2024 |

## Eidesstattliche Versicherung

Hiermit erkläre ich an Eidesstatt, dass die Dissertation von mir selbstständig, gemäß Promotionsordnung vom 12.07.11, §8, Abs. 2, Pkt. 5, ohne unerlaubte Beihilfe angefertigt ist.

München, den 06.10.2024

. . . . . .

*Pascal Jungblut*

*Dedicated to Alexandra Elbakyan*

## Abstract

Field Programmable Gate Arrays (FPGAs) are integrated circuits that can be reconfigured dynamically. Accelerators for offloading computation based on FPGAs demonstrate potential as energy-efficient and flexible alternatives to conventional accelerators in High-Performance and Cloud Computing. Scheduling tasks on FPGAs is comparable to the allocation of resources on a chip: each offloaded task occupies an area of the chip during its execution. Task scheduling on FPGAs is typically done using Partial Reconfiguration (PR), where a subset of the FPGA is configured to execute a task while the remaining circuits remain unchanged, inspired years of research for scheduling strategies based on PR. However, PR's reliance on low-level features limits its portability and necessitates expert knowledge, hindering the application of existing research for task scheduling and the overall adoption of FPGA-based accelerators. In this work, we aim to support software developers in integrating accelerators based on FPGAs by asking how to optimize task scheduling on FPGA-based accelerators without relying on PR?

To address our research question, we present three key contributions: first, we introduce an abstraction-agnostic methodology for describing FPGAs and scheduling strategies, centered around deducing scheduling constraints from a machine model representing a target FPGA. Given a task graph, optimal schedules can be generated with constraint programming and analyzed systematically afterwards. We apply our methodology exemplarily in a case study for two machine models – one supporting PR and one without PR – and compare the resulting schedules generated for task graphs of existing applications, demonstrating that avoiding PR is feasible. Second, we introduce an algorithm that uses heuristics to find schedules in polynomial time, since optimal scheduling is an NP-complete problem. The algorithm supports the scheduling of tasks on FPGAs without a dependency on PR. It is evaluated and the derived schedules are compared against optimal schedules. Third, we apply a genetic algorithm to perform Design Space Exploration (DSE) for the machine model. Its goal is to recommend minimal or automated changes to the input program, which in turn affect the quality of possible schedules. We show that effective recommendations can be generated.

Our results can help vendors provide significantly more streamlined workflows for programming FPGAs and thus make the platform more appealing for users. The insights can also be used to extend established programming models, such as OpenCL, to accommodate the unique characteristics of FPGAs. Based on our research, architectural optimizations for an approach with a vastly simplified PR implementation both in hardware and software can be considered. Lastly, automating the generation of constraints for an accelerator would support the selection of the most suitable scheduling strategy without human interaction.

## Kurzfassung

Field Programmable Gate Arrays (FPGAs) sind integrierte Schaltungen, die dynamisch rekonfiguriert werden können. Beschleuniger zur Auslagerung von Berechnungen auf Basis von FPGAs zeigen Potenzial als energieeffiziente und flexible Alternativen zu herkömmlichen Beschleunigern im Bereich Hochleistungs- und Cloud-Computing. Die Planung von Aufgaben auf FPGAs ist vergleichbar mit der Zuteilung von Ressourcen auf einem Chip: Jede ausgelagerte Aufgabe belegt während ihrer Ausführung einen Bereich des Chips. Die Aufgabenplanung auf FPGAs erfolgt in der Regel mit Partial Reconfiguration (PR), bei der ein Teil des FPGAs zur Ausführung einer Aufgabe konfiguriert wird, während die verbleibenden Schaltkreise unverändert bleiben. Dies inspirierte jahrelange Forschung zu Planungsstrategien auf Basis von PR. Die Abhängigkeit von PR von Low-Level-Funktionen schränkt jedoch seine Portabilität ein und erfordert Expertenwissen, was die Anwendung vorhandener Forschung für die Aufgabenplanung und die allgemeine Einführung von FPGA-basierten Beschleunigern erschwert. In dieser Arbeit möchten wir Softwareentwickler bei der Integration von FPGA-basierten Beschleunigern unterstützen, indem wir untersuchen, wie die Aufgabenplanung auf FPGA-basierten Beschleunigern ohne Verwendung von PR optimiert werden kann?

Um unsere Forschungsfrage zu beantworten, präsentieren wir drei zentrale Beiträge: Erstens führen wir eine abstraktionsagnostische Methodik ein, um FPGAs und Planungsstrategien für FPGAs zu beschreiben, die sich darauf konzentriert, Planungsbeschränkungen aus einem Maschinenmodell abzuleiten, das ein Ziel-FPGA repräsentiert. Ausgehend von einem Aufgabengraphen können optimale Pläne mit Constraint-Programmierung erstellt und anschließend systematisch analysiert werden. Wir wenden unsere Methodik beispielhaft in einer Fallstudie für zwei Maschinenmodelle an - eines mit PR-Unterstützung und eines ohne PR - und vergleichen die resultierenden Pläne, die für Aufgabengraphen bestehender Anwendungen erstellt wurden, und zeigen, dass die Vermeidung von PR machbar ist. Zweitens führen wir einen Algorithmus ein, der Heuristiken verwendet, um Pläne in polynomialer Zeit zu finden, da die optimale Planung ein NP-vollständiges Problem ist. Der Algorithmus unterstützt die Planung von Aufgaben auf FPGAs ohne Abhängigkeit von PR. Er wird bewertet und die abgeleiteten Pläne werden mit optimalen Plänen verglichen. Drittens wenden wir einen genetischen Algorithmus an, um die Design Space Exploration (DSE) für das Maschinenmodell durchzuführen. Ziel ist es, minimale oder automatisierte Änderungen am Eingabeprogramm vorzuschlagen, die die Qualität möglicher Pläne beeinflussen. Wir zeigen, dass effektive Empfehlungen generiert werden können.

Unsere Ergebnisse können Herstellern helfen, erheblich effizientere Arbeitsabläufe für die Programmierung von FPGAs bereitzustellen und damit die Plattform für Benutzer attraktiver zu gestalten. Die Erkenntnisse können auch dazu verwendet werden, etablierte Programmiermodelle, wie zum Beispiel OpenCL, um die einzigartigen Eigenschaften von FPGAs zu erweitern. Basierend auf unserer Forschung können architektonische Optimierungen für einen Ansatz mit einer stark vereinfachten PR-Implementierung sowohl in Hardware als auch in Software in Betracht gezogen werden. Schließlich würde die Automatisierung der Erstellung von Einschränkungen für einen Beschleuniger die Auswahl der am besten geeigneten Planungsstrategie ohne menschliche Interaktion unterstützen.

# Contents

# Chapter 1

## Introduction

Since the introduction of the Integrated Circuit (IC) in 1958, users could rely on an exponential increase in computational capabilities over time. In 1965, Gordon Moore famously postulated the doubling of the number of transistors for a given area every two years [1]. Simultaneously, the energy consumed by chip area stayed constant. This observation by Dennard et al. resulted in an increase in operations per Watt over time [2]. Around 2005, the *Dennard scaling* came to an end, which led to the focus on multicore processors and more specialized chips that could provide better performance per Watt [3].

In an attempt to counter the impending stagnation of computational performance for a given power envelope, the computing industry is seeking new effective methods. A good reference for state-of-the-art computing is the TOP500 list of supercomputers [4]. As the name suggests, it lists the 500 most powerful computers worldwide and also provides historic data since its first release in 1993. The High Performance Computing (HPC) Linpack Benchmark is used to assess the performance of the systems and rank them accordingly [5]. A general trend since Moore's law began breaking down in around 2005 is the increased utilization of *accelerators*, hardware extensions that are optimized for certain tasks performed by the system. Depending on the tasks, such accelerators can be more energy efficient and thus more economical than general purpose Central Processing Units (CPUs) [6]. The total power consumption and by extension the energy efficiency are major concerns, since the necessary energy has to be delivered to the facility and the waste heat has to be dissipated. Both – power consumption and heat dissipation – can be significant factors of the Total Cost of Ownership (TCO) [7].

In the November 2022 edition of the TOP500 list Graphics Processing Units (GPUs) are the prevalent accelerators. Over time the share of systems built with GPUs has been growing. A total of 59% of the performance results from these systems. GPUs offer superior energy efficiency compared to CPUs when performing the HPC Linpack Benchmark used to rank the systems [6]. The superior efficiency is apparent in a related list, the *Green500*, which ranks the systems in terms of energy efficiency instead of maximum performance [8]. Nine out of the ten most efficient supercomputers use GPUs; the only exception is *NM-3* at Preferred Networks that uses a custom accelerator chip [9].

Energy efficiency is not only crucial for supercomputers, but also for other consumers: data centers, desktops and mobile devices – either for the same economic reasons, to extend the mobility or to fit a thermal budget. Thus, they also increasingly rely on accelerators. Suitable workloads are then *offloaded* to the accelerator with the expectation of utilizing less time and potentially less energy compared to complex CPUs. Among others, the following technologies are used to implement accelerators:

- *GPUs* were originally developed for graphics processing but are increasingly used for general-purpose computing.

- Digital Signal Processors (DSPs) are used to efficiently manipulate digital signals, e.g. to filter specific frequencies from a digital signal.
- *Tensor Processing Units (TPUs)* were introduced in 2017 for machine-learning workloads implementing efficient matrix multiplication operations [10].
- Dedicated Application Specific Integrated Circuits (ASICs) can be developed, when economically viable.
- Field Programmable Gate Arrays (FPGAs) can be configured to represent any circuit that will not exceed the chip's resources.

GPUs, DSPs, TPUs and ASICs have a static logic: their circuitry is designed once and then etched into silicon. A more flexible technology is offered by FPGAs. *Field programmable* means that the circuit of the chip can be adjusted after it has been deployed. They consist of an *array* of (logic) *gates* that can be configured and connected flexibly. The resulting circuitry is similar to a conventional chip. Consequently, accelerators can also be built using FPGAs. Depending on the application these can be even more energy efficient than GPUs, because FPGAs can be configured specifically for the workload at hand. This can reduce the computational overhead, all while providing more flexibility [6], [11].

While the exact structure of an FPGA depends on its purpose, the general layout is similar in most devices. Figure 1.1 shows a simplified example structure of a typical FPGA. It consists of

- a 2-dimensional array of programmable logic blocks (rectangles)
- connections with programmable switches (circles), and
- an interface to Input/Output (I/O) systems (memory controllers, network interfaces, antennas, buses) arranged around the logic blocks.

Modern FPGAs, such as the Intel Stratix 10 GX or Xilinx' Virtex VU19p, contain up to 10 million logic blocks, some of which implement specialized functions [12]. For example, a certain amount can be dedicated DSP blocks that perform operations on digital signal data and may also perform efficient operations on IEEE-754 floating point numbers of arbitrary precision. Moreover, memory blocks are included in the array for fast access to intermediate data. More complex chips even feature a tight integration of CPU cores, e.g. ARM Cortex cores, or direct access to optical fiber network interfaces [13].

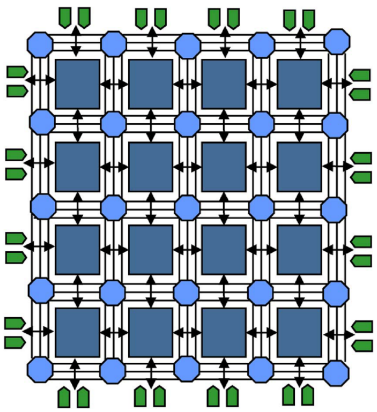


Figure 1.1: Structure of an FPGA [13]

Using FPGAs as accelerators can provide a range of advantages over CPUs and GPUs:

- For some applications FPGAs can offer acceleration and *better absolute performance* than CPUs and GPUs [14], [15].
- FPGAs can provide *better energy efficiency*, especially compared to CPUs, for example for graph processing [16], matrix operations [11] or computer vision [6].
- The programmability of the FPGA offers *potential for experimentation*: where it may cost years of development time and large financial investments to ship improved silicon to customers, FPGAs can be updated in the field, so that optimizations can easily be evaluated.
- The fine-grained *precise control* over low-level details, e.g. the bit-width of operations, potentially allows for more resource efficient solutions.
- Some FPGA-based accelerators offer *integration of high performance network interfaces*: these can be directly connected to the FPGA, and allow the implementation of optimized network protocols in hardware, "in network" processing and the design of distributed streaming accelerators [17].
- CPUs that are executed on top of the FPGA substrate allow the usage of flexible and experimental CPU implementations, so called *Soft Cores*.

Conceptually, the programming of FPGAs is more similar to the design process of an IC than to software development. This leads to some disadvantages, especially from the point of view of a software developer, who is usually tasked with the integration of accelerators:

- Compiling a program or description for an FPGA is *time-consuming*, because of their flexible construction and the resulting search space. It can take hours, especially for FPGAs with a complex structure.
- The increased control over low-level features can result in the necessity to define them rigorously, leading to *overhead in the development*.
- Interfaces and architecture details are often *proprietary* [18].
- Debugging a circuit requires *knowledge in digital circuit design*.
- FPGAs operate at *lower frequencies* compared to CPUs and GPUs.

Despite these challenges, computer science and electrical engineering research shows great interest in FPGAs and the semiconductor industry is also investing heavily in FPGA technology. Intel, the world's largest vendor of CPUs by revenue acquired the FPGA specialist Altera in 2015 for $16.7 billion (15.3 billion Euros at the time). Altera's largest competitor, Xilinx, was later acquired by CPU designer Advanced Micro Devices (AMD), a direct competitor of Intel, at a $35 billion (30.7 billion Euros) valuation [19]–[22]. Despite the attention and the investments of the industry, the installed base of FPGAs accelerators for compute- or data-intense workloads is low compared to GPUs, which can also be observed in the TOP500 as of November 2022: *none* of the systems are equipped with FPGA-based accelerators.

Designing efficient circuits for FPGAs can be challenging, especially for software developers without expert knowledge about the structure of the chip and digital circuit design. Thus, vendors introduced higher abstractions that are supposed to make programming for FPGAs more accessible [23]. High-Level Synthesis (HLS) allows users to describe an algorithm in a conventional programming language or a Domain-Specific Language (DSL) which is then further compiled to low-level representations suitable for FPGA programming. There exist a multitude of high-level programming languages that can be compiled to an FPGA configuration, including general purpose languages like ANSI C and C++ [24].

Figure 1.2 shows the workflow of compiling HLS applications exemplary for a Xilinx FPGA, but the process translates directly to other vendors. The inputs at the top are source files:

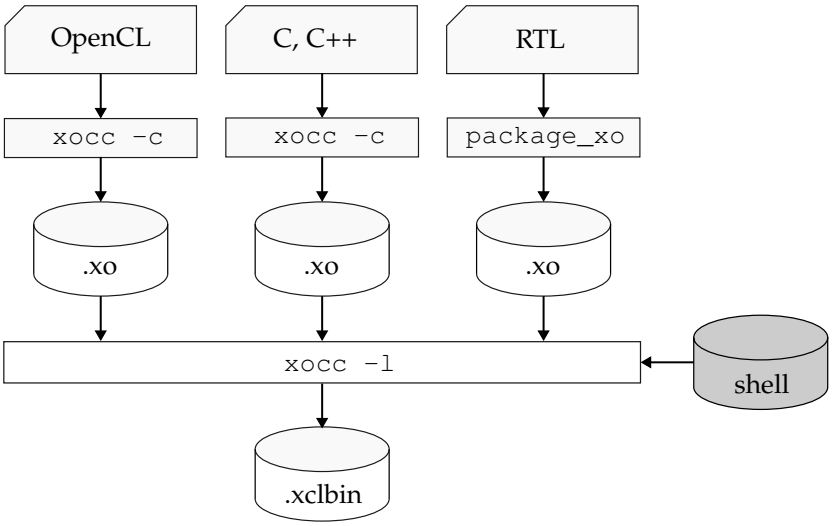



Figure 1.2: The process of HLS compilation for FPGAs [25]

high-level descriptions like OpenCL, C or C++ code or lower-level Register-Transfer Level (RTL) descriptions of the desired functionality. These inputs are compiled to `xo` object either by Xilinx' `xocc` command or using `package_xo` in the case of RTL code. The object files are then linked together into a binary `xclbin` file. Thus, one can easily combine OpenCL, C/C++ and RTL. Additionally, a *shell* is included. The shell serves as a well-defined interface between the application code as supplied by the input files, and the rest of the system like memory interfaces and clocks. The result of the linking process is a binary file that can be copied to the FPGA to implement the functionality of the input files on the chip [26].

The binary file is called a *configuration* or the *bitstream*. This data describes the connections that are active in the programmable switches (i.e. the routing), truth tables of all Lookup Tables (LUTs), initialization values of other on-chip memories and potentially architecture specific values and meta information about the configuration. The format of the file is vendor-specific and the function of each bit is often undocumented. The configuration is either copied to a dedicated memory or streamed to a controller that handles the process of configuring.

Today, FPGAs used in accelerators support either only Dynamic Reconfiguration (DR) or the more evolved Partial Reconfiguration (PR):

1. *DR*: a change of the configuration does not require a restart of the system. This drastically reduces the overhead of applying a new configuration. Data that is needed for the computation is then copied to the accelerator's own memory and a runtime invokes the processing of the function that was offloaded. After the execution, the generated result must be copied back to the host's main memory. When the execution of another function is requested, the associated bitstream must be copied to the device and a reconfiguration must be conducted. The complete workflow is described more in-depth in Section 2.1.
2. *PR* (or Dynamic Partial Reconfiguration (DPR)): a change of the configuration does not require a restart of the system, *and* it is possible to change only a section of the configuration while the rest keeps operating. PR is more advanced and less constrained than DR, because it does not require changing the complete configuration at once.

PR has some advantages over pure DR:

- The number of elements on the chip affected by the reconfiguration is smaller, potentially leading to shorter compilation times.
- The bitstream is smaller, because it must hold information for fewer elements, speeding up the configuration process.
- The FPGA can be divided into regions that operate independently, allowing fine-grained control over the chip's circuitry.
- A runtime can react flexibly to changing load by adjusting the regions' configurations accordingly.

These benefits and in particular their interaction with task scheduling[1] make PR a popular prerequisite for FPGA related research.

A task is a unit of work, for example a function that was invoked. *Task scheduling is the allocation of tasks to processors and the definition of their execution order* [27].

Intuitively, this maps well to the concept of allocating resources on an FPGA and changing them over time. One idealized and simplified way to visualize the mapping of tasks to a chip is shown in Figure 1.3. A collection of tasks together with their dependencies is shown on the left as a Directed Acyclic Graph (DAG). The right diagram depicts an allocation at a fixed time. The tasks are mapped to areas on the chip and arranged so that they are non-overlapping and so that their dependencies are met, i.e. no task is allocated before all of its predecessors have finished. The tasks are shown as 2-dimensional rectangles that are arranged without any overlap, both in time and in space. *All* approaches for scheduling on FPGAs attempt to find such a non-overlapping alignment of tasks, albeit with varying degrees of abstraction or additional constraints.

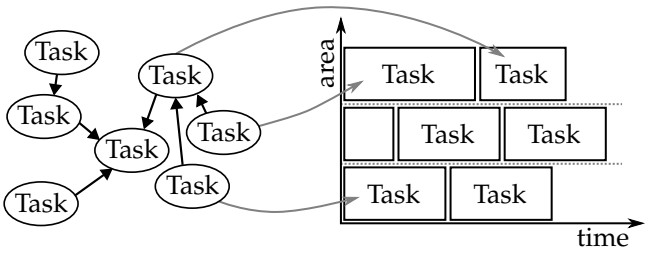



Figure 1.3: A mapping of tasks to FPGA chip area

However, PR also poses disadvantages that affect its adoption:

- The complexity involved with PR results in an increased development overhead (also compared to plain DR).
- Fragmentation can lead to the inefficient use of chip resources, because PR typically cannot be applied arbitrarily. Instead, only certain (rectangular) regions may be changed.
- Due to the smaller search space, the circuits generated for a reconfigurable region may be performing worse when compared to plain DR.
- PR is tied to low-level details like the physical arrangement of the chip's resources. This hinders the portability of any software that relies on it.

In Chapter 2 we show that considerable efforts have been made to improve the outcomes of scheduling algorithms. Either by considering additional restrictions or by optimizing the

[1] In the context of FPGAs, "scheduling" is also a phase in the HLS workflow that is concerned with the decomposition of the algorithm into steps that can be executed in one clock cycle. The scope of this thesis is *task* scheduling, a more abstract process.

technical implementation of PR systems. Even though a lot of research has gone into scheduling on FPGAs supporting PR, it is not a commonly used feature, especially in environments built around high-level programming [18], [24]. Moreover, despite a plethora of existing approaches, virtually none of them are readily available for programmers, researchers or even domain experts. To a large degree this is caused by the strong dependency on low-level and vendor-specific implementations. While they give users of the system precise control over its workings, they also require expert knowledge or specific hardware. The dependence on non-standard, vendor- or even product-specific implementations hinders the portability and thus also the adoption.

Fundamentally, there are two scenarios for offloading techniques that require PR:

1. An approach is *specialized for an architecture*. This leads to well-performing solutions that are difficult to adapt to other architectures. An example for this is described by Pham et al. [28], who present an OpenCL implementation for the ZYNQ UltraScale+ hardware.
2. An approach is *general and architecture-agnostic*. Formulating a scheduling algorithm in such a way makes it more universal, but it also implies that it has to be ported for each specific architecture. For example, an Integer Linear Programming (ILP) formulation as presented by Redaelli et al. [29] can generate optimal schedules for FPGAs that allow 2-dimensional fragmentation of the configuration. However, it is not available for any concrete hardware implementation and must be implemented for each vendor or even for each specific chip model. This implementation requires expert knowledge and possibly a significant development effort.

Porting a scheduling algorithm that uses PR is a major task, because there is no underlying standard, e.g. no equivalent of an x86 or RISC-V instruction set for FPGA programming. Abstractions *do* exist, but they are not widely supported [30].

Sticking to the CPU instruction set example: analogous to PR we can consider vectorization features of modern CPUs. Each instruction set has its own instructions for vectorization as the implementation may differ notably. If the compiler fails to insert these automatically through auto-vectorization, programmers must rely on architecture-specific low-level statements. These can provide superior performance and energy-efficiency but require expert knowledge and are not portable across architectures.

The situation is very similar with PR, except that the equivalent of auto-vectorization is not commonly found in HLS software. As a result, although the concept of task-level PR could lay the foundation for efficient scheduling on FPGAs, it is barely used in practice. In their recent survey *FPGA dynamic and partial reconfiguration: A survey of architectures, methods, and applications* [18] Vipin and Fahmy conclude:

> *"In modern commercial FPGAs, PR is an auxiliary feature rather than something around which the architecture is designed. This means many aspects of PR design are tied to low-level architecture details requiring significant expertise.*
>
> *[. . . ]*
>
> *The design of PR systems remains difficult, and hence, only accessible to FPGA experts. Many published techniques for overcoming the limitations of vendor tools have slowly become obsolete, as a result of the increasing heterogeneity of modern devices and less open access provided by vendors. Since many techniques are also heavily tied to specific architectures, with their evolution, these tools can become unusable. As a result of these difficulties, most systems that use PR at present must be designed at a low level with detailed hardware design expertise required."*

We share the assessment that the dependence on PR poses a high burden on implementors and leads to the deprecation of many existing offloading approaches. Even Vipin and Fahmy's own efforts from 2014 to introduce a dedicated test platform specifically for the evaluation of PR are only usable with adjustments on more recent FPGAs [31]. We analyze the current state of the art, its features and limitations in Chapter 2.

## 1.1 Problem Statement

With the potential for acceleration and possibly energy efficient computing, FPGAs could be a valid alternative to GPUs for compute-intense workloads. The broader motivation of this work is to make the potential of FPGAs available for a wider range of users, without requiring expert knowledge, niche technologies or complicated workflows. Instead, a more pragmatic and user-centered approach is desired.

We identify three main reasons that hinder the adoption of the research in the field:

1) *Lack of generality*: due to the tendency to prefer more optimized solution over universal ones, the approaches tend to be tailored to specific vendors or chips.
2) *Lack of portability*: as a result of the specialization and industry trends, these approaches developed for one technology can be harder or impossible to apply to others.
3) *Lack of comparability*: the variations in abstraction level and technical conditions of the approaches make it more difficult to identify solutions for a particular use case.

As a consequence, we propose an approach that avoids the need for specialized features that cause the problems mentioned above. Specifically supporting partial reconfiguration on a per-task granularity introduces overhead for the software developers, can lead to fragmentation of chip area and hinders the portability of codes [18]. Our central Research Question (RQ) is:

> *How can we optimize the task scheduling of tasks on FPGA-based accelerators without relying on low-level features?*

To answer this question, we divide it into three more Sub-Research Questions (SRQs):

- *SRQ 1*: How can we systematically evaluate reconfiguration-aware scheduling strategies for task-based workloads?

  We first ask how an optimization can be analyzed when the strategies are notoriously complex to implement, heterogeneous and of varying degrees of specialization. For example, how can a strategy that was tuned to a ZYNC Ultrascale+ chip [28] be compared to an abstract ILP formulation of the scheduling problem [29]? Without a notion of comparability, any optimizations lack validity. We describe how such a comparable approach can be formulated, what its benefits and limitations are and how valid its assertions are.

- *SRQ 2*: How can reconfiguration-aware *task scheduling algorithms* allow optimizations of the schedule?

  After answering SRQ 1 and having a foundation to argue about scheduling on FPGAs systematically, is it possible to develop algorithms that do not rely on PR and compare them against established methods? What is the cost of such scheduling strategies, compared to existing ones? How can heuristics help to mitigate the inherent complexity of the problem? How do heuristics perform compared to optimal solutions?

- *SRQ 3*: How can we optimize the high-level code based on a task-graph?
  The HLS workflow is a complex interaction between the high-level code, the compiler(s) and the hardware. Without changing the latter two, can we optimize the code based on insights from SRQ 1 and SRQ 2? Is it possible to generate recommendations for a given code, a set of tasks and the FPGA it is executed on? Can these changes affect the resulting schedule positively?

## 1.2 Contributions

The main contributions of this thesis are the following:

1. A *formal description* to evaluate and optimize reconfiguration-aware scheduling strategies and design space exploration for given workloads

   We introduce a framework that allows the integration of reconfigurable machine models. Two machine models are provided exemplarily: one that supports partial reconfiguration and one that does not. This framework can be used to compare other scheduling approaches for reconfigurable hardware as well.

2. A *scheduling algorithm* for tasks without the requirement for task-level reconfiguration

   An algorithm using heuristics is introduced. It uses list scheduling to generate a schedule from a task graph and the previously introduced machine model. A second variant of the algorithm is presented to handle communication congestion.

3. A *process for Design Space Exploration (DSE)*

   The method aims to optimize high-level programming code targeted for FPGAs. It uses a Genetic Algorithm and existing task graphs to identify opportunities for optimization of the code with minimal effort from the user.

4. An *evaluation* of the framework, the algorithms and the DSE and its effects on applications in terms of performance

   Executions of established applications traced on a high-end FPGA and random generators are used to create cost-accurate task graphs. With these, we apply the above algorithms to the task graphs and evaluate the resulting schedules compared to optimal schedules.

In the tradition of good scientific practice, we also provide the following with this thesis:

1. A *collection of trace-data* on FPGAs and derived *task-graphs* for HPC applications

   This data was collected during the execution of the OpenDwarfs benchmark collection [32] on a modern high-end FPGA and processed to generate representative task-graphs.

2. The *software* that was used to generate, compare and evaluate the data

   The software collection comprises a constraint programming representation of our approach, a framework for the simulation of schedules and programs to generate task-graphs. Additionally, software to instrument OpenCL programs and derive task-graphs is included as well as a program to validate machine models against hardware.

Details and pointers to these resources are provided in Chapter 6.

## 1.3 Preliminary Works

A list of the author's publications together with a description of the content and the author's share follow. The list is divided into three parts:

1. Publications directly related to this thesis, with some of the contents inspiring later chapters. This is denoted in the chapters' introduction, respectively.
2. Publications that are only partially related to scheduling on FPGAs.
3. Publications to which the author contributed a smaller part.

### 1.3.1 Publications Directly Related to the Thesis

- P. Jungblut and D. Kranzlmüller, "Optimal schedules for high-level programming environments on FPGAs with constraint programming," in *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, IEEE, 2022, pp. 96–99

  In this publication we describe a simplified version of a theoretical foundation that is found in Chapter 3. Additionally, two scheduling algorithms are described and compared against an optimal solution based on a constraint programming formulation. Two FPGA machine models for scheduling are introduced: one with and one without support for partial reconfiguration.

  The author developed the idea, implemented the algorithms, performed the evaluation and wrote the paper.

- P. Jungblut and D. Kranzlmüller, "Dynamic spatial multiplexing on FPGAs with OpenCL," in *International Symposium on Applied Reconfigurable Computing*, Rennes, France: Springer, 2021, pp. 265–274

  This publication contains the predecessor to the paper above with a focus on *online* scheduling. A performance model for tasks on FPGAs is combined with a reconfiguration-aware scheduler that was integrated into the existing runtime. The performance is evaluated and it is shown that this runtime scheduling is on-par with manual performance tuning.

  The author developed the idea, implemented the scheduler, performed the evaluation and wrote the paper.

- P. Jungblut, "Task scheduling in reconfigurable computing with OpenCL," in *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, New Orleans, LA, USA, Jun. 2021, p. 1023

  The author presented the research questions of this thesis in the PhD-forum.

- P. Jungblut, "Task scheduling on FPGA-based accelerators without partial reconfiguration," presented at the International Conference on High Performance Computing, Networking, Storage and Analysis (SC) (Dallas, TX, USA), Nov. 17, 2022

  A preliminary version of this work was selected for the doctoral showcase at the Supercomputing Conference 2022.

### 1.3.2 Publications Partially Related to the Thesis

- P. Jungblut and K. Fürlinger, "Integrating node-level parallelism abstractions into the PGAS model," in *Proceedings of the 13th International Symposium on High-Level Parallel Programming and Applications*, vol. 13, Porto, Portugal, 2020, pp. 38–56

The integration of portable programming models for accelerators in a Portable Global Address Space (PGAS) environment is the content of this work. The approach consists of a domain decomposition scheme that is combined with distributed memory programming and accelerator offloading. The method is demonstrated as an integration of DASH [38] and Alpaka [39]. It is shown that the approach performs similar to traditional MPI+X solutions but is more portable.

The author developed the decomposition scheme, programmed the integration of DASH and Alpaka, performed the evaluation and wrote the article.

Karl Fürlinger advised the process and revised the document.

- P. Jungblut and K. Fürlinger, "Portable node-level parallelism for the PGAS model," *International Journal of Parallel Programming*, vol. 49, no. 6, pp. 867–885, Jun. 5, 2021

  This work is an extension to our work above [37]. It contains a more thorough evaluation, especially highlighting the portability of the approach with data from additional platforms.

  The author performed the extended evaluation and wrote the article. The remainder of the paper was taken from the previous original work by the author (see above) [37].

  Karl Fürlinger advised the process and revised the document.

- P. Jungblut, R. Kowalewski, and K. Fürlinger, "Source-to-source instrumentation for profiling runtime behavior of C++ containers," in *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, Exeter, UK: IEEE, 2018, pp. 948–953

  This paper describes a source-to-source compiler based on LLVM that instruments data structures in existing code bases. During the execution, data traces are extracted and analyzed, e.g. using a density based clustering to find outliers in the data structures.

  The author performed the majority of the work which originated from his master thesis: the selection of an appropriate method, the implementation, deriving the model for analysis and the evaluation. The author also wrote the article.

  Roger Kowalewski advised with the implementation, specifically during the evaluation of the preceding master thesis. Karl Fürlinger introduced the idea and motivation and advised the work and document.

### 1.3.3 Other Publications

- R. Kowalewski, P. Jungblut, and K. Fürlinger, "Engineering a distributed histogram sort," in *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, Albuquerque, NM, USA: IEEE, 2019, pp. 1–11

  The publication presents a novel sorting algorithm for distributed memory systems.

  The author contributed a minor part of the implementation and supported the evaluation, especially the implementation and evaluation of a local parallel sorting and partitioning strategies.

- K. Fürlinger, J. Gracia, A. Knüpfer, T. Fuchs, D. Hünich, P. Jungblut, R. Kowalewski, and J. Schuchart, "DASH: Distributed data structures and parallel algorithms in a global address space," in *Software for Exascale Computing-SPPEXA 2016-2019*, Springer, 2020, pp. 103–142

  The paper presents the collective work on DASH.

The author contributed Section 5.1 about the integration with ALPAKA and LLAMA [39] in the context of the MEPHISTO project, which funded his position.

## 1.4 Structure of this Thesis

Figure 1.4 shows the organization of this thesis from top to bottom. It starts with this introduction and Chapter 2, followed by the main part in Chapters 3, 4 and 5. These parts are evaluated and finally concluded with a discussion and an outlook. The arrows indicate dependencies between the chapters. Thus, Chapter 4 depends on Chapter 3, and both Chapters 3 and 4 are a prerequisite for Chapter 5. The main part and the core contributions of this thesis is contained in the dotted rectangle.

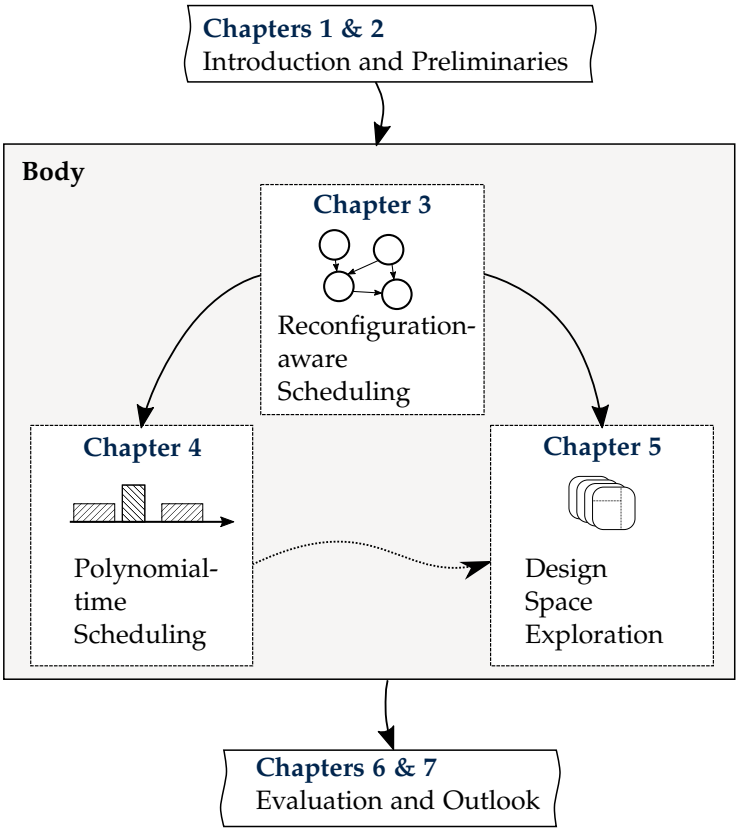



Figure 1.4: Structure and contents of the thesis

This chapter describes the motivation, problem statement as well as preliminary works from the author.

Chapter 2 consists of two parts:

1. An introduction of the basic concepts of reconfigurable computing, FPGAs, task scheduling and high-level programming. We describe the architecture of an FPGA accelerator and how it interacts with its host system. The *OpenCL* programming environment is introduced as a continuous example to illustrate our approach and its applications.
2. After the fundamentals are presented, the current state of the art of scheduling on reconfigurable hardware is presented. We put special focus on the portability and practicality of the existing approaches.

Chapter 2 concludes with the motivation for our approach as a result of a lack of portable, future-proof and pragmatic solutions for scheduling tasks on reconfigurable hardware.

In Chapter 3 we present a theoretical framework to define models for FPGA-based accelerators and to analyze scheduling algorithms on reconfigurable hardware. From these definitions a series of constraints can be derived that must hold for valid schedules on machines adhering to the models. These constraints are further adopted to form a constraint programming formulation, which allows the automatic verification and generation of optimal schedules for a given task graph and machine model. The chapter is mainly concerned with SRQ 1.

With the foundation in place, their application to scheduling strategies is investigated in Chapter 4. A scheduling algorithm is introduced and studied in the context of the previously introduced machines models. How do the scheduling algorithms impact the schedule? What is the influence of the provided configurations, the machine models and their parameters on the result? These are questions we pursue in Chapters 4 and 6 to answer SRQ 2.

After both the framework and concrete scheduling algorithms are defined, we explore the possibilities of DSE in Chapter 5. Given a task graph and a scheduling algorithm: can we recommend changes to the high-level code? Is it beneficial to migrate a kernel from one configuration to another? Or would the execution benefit if the resources of one configuration were balanced differently? This chapter describes strategies to systematically explore such optimizations based on the previous findings to answer SRQ 3.

In Chapter 6 the strategies are evaluated with simulations based on parameters previously obtained from traces on a current high-end FPGA and randomly generated task graphs. The simulations are also compared against the results obtained from schedules executed on the accelerator. We also focus on the influence of the various parameters on the scheduling quality. From that, recommendations for chip designers and runtime developers are derived.

Finally, the thesis is concluded in Chapter 7. Our approach and especially the results of the evaluation are discussed and their strengths and weaknesses are highlighted. We summarize our findings and take a glance into the future of reconfigurable computing within high-level computing environments.

# Chapter 2

# Preliminaries

The research conducted to accelerate applications with FPGAs is expectedly focused on the unique characteristics of the platform. The properties that make FPGAs special compared to other accelerators are a result of their reconfigurability: the capability to change the circuit at runtime offers flexibility but also poses challenges. To understand both the challenges and advantages this chapter is divided into six sections:

Section 2.1 provides a more detailed description of the architecture of an FPGA. The focus lies on the usage of FPGAs as (part of) accelerators, their integration into a host system, the high-level programming models supporting such hardware and the HLS workflow from the program input to the runtime. The hardware is described in a vendor-independent and abstract form for the following reasons:

- An advantage of the HLS workflow is its abstraction from a concrete hardware.
- Lower-level details are often vendor- or model-specific and sometimes even undocumented.
- The exact architecture is not relevant for our research questions and thus related work. The goal is a more abstract but still useful and applicable description of the system.
- The model presented in Chapter 3 is designed for abstract descriptions of the hardware. It supports a more narrow description where necessary.

The role of energy efficiency of FPGAs is examined in Section 2.2. The section focuses on the claims of high energy efficiency with regard to task scheduling in particular.

With the technical prerequisites described, an in-depth analysis and classification of existing scheduling techniques is provided in Section 2.3.

The workings of a genetic algorithm Section 2.4 are highlighted as a prerequisite for Chapter 5, where it is used to optimize FPGA configurations.

A more theoretical description of the scheduling problem follows in Section 2.5. It covers basic definitions to formulate the problems and objectives clearly.

Section 2.6 shows our extensions to these basic definitions to reflect the characteristics of FPGAs. These consist of a notion of *configurations* and *locations* and an extended task graph.

## 2.1 Program Acceleration with FPGAs

Programming FPGAs is different from typical software development. Most computers today are constructed using the *von Neumann architecture*. Besides other components, it consists of a control unit. The unit controls the execution of instructions that are stored as data in the computer's memory. The CPU is capable of reading the instructions and performing the actions they encode, i.e. executing them. This is a flexible approach to computing, since in a von Neumann architecture the memory can be changed by the computer itself [43]. In contrast, FPGAs do typically work with a fixed but reconfigurable circuit, not on instructions stored in memory.

To give software programmers access to their well-known way of expressing algorithms, HLS converts a high-level description of a program or parts thereof to a configuration for an FPGA. Copying the configuration to the FPGA and starting the execution is a complex process. Additionally, the integration of an FPGA into a host system for acceleration is complex, since it consists of many components – all of which can have an influence on the accelerator's behavior. For a comprehensible overview, we organize these components into layers, where the lowest layer represents the hardware (FPGA in this case) and the top layer represents the functions to be accelerated.

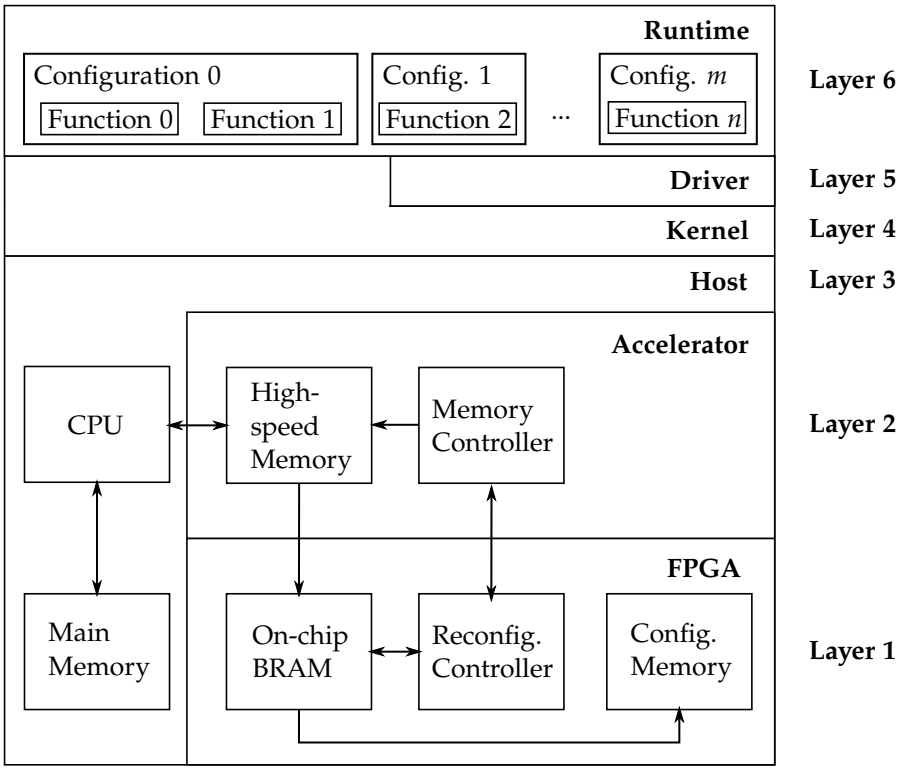



Figure 2.1: Accelerator stack

Figure 2.1 depicts the stack from the FPGA hardware at the bottom on Layer 1 up to the accelerated functions at the top on Layer 6. Some layers are not necessarily present from the perspective of the upper layer, e.g. a runtime may interact directly with the kernel, since it incorporates some functionality of a driver. In this case the respective layer does not over the whole width of the diagram. The layers are:

- Layer 1: An *FPGA* itself is the silicon that implements the circuits.
- Layer 2: An *accelerator* adds supporting interfaces and hardware to the FPGA, e.g. off-chip memory or I/O interfaces.
- Layer 3: A *host* is the controlling computer with its CPU and main memory, connected through a high-bandwidth connection to the accelerator.
- Layer 4: A *kernel* and an operating system are providing low-level primitives to interact with the hardware.
- Layer 5: A *driver* is controlling the accelerator and by extension the FPGA.
- Layer 6: A *runtime* is interfacing with the driver to control the accelerator, manage configurations and acceleration of user-provided functions.

While this is a typical setup, the layer of focus varies considerably. For example, some approaches integrate the acceleration deeply into the operating system [44], [45], removing the need for drivers and explicit runtimes. Others view Layers 1-4 as a given and provide a virtual interface to the accelerator [46]–[48].

The techniques in Chapters 4 and 5 are primarily focused on Layer 6, the runtime and the organization of its components. However, the framework in Chapter 3 makes no assumptions about the abstraction level of an approach. Instead, its flexibility enables comparisons between vastly different techniques.

To understand the interactions from the input program and the resulting configuration and, in turn, its interaction with the runtime, each of the layers is briefly described and its influence on scheduling is highlighted.

### 2.1.1 Layer 1: FPGAs

In contrast to a CPU or an ASIC, an FPGA is *field programmable*. Its circuitry is determined by a configuration that is not etched into silicon at the time of production. Thus, its construction is different from other static integrated circuits. A typical FPGA is built around core components:

- *n-LUTs* that take $n$ (digital) input signals and provide one output signal that depends on the *values* of the input signal. The values can be either high or low (voltage), represented by 1/true or 0/false, respectively.
- *Flip-flops* hold the value of a signal.
- LUTs and flip-flops are combined into *Logic Blocks (LBs)*.
- Special *hard blocks* or hard IPs (intellectual property) provide optimized implementations for typical operations.
- The signals of logic and memory blocks are connected with a *switchable interconnect*.

An n-LUT represents a boolean function that maps each pattern of its $n$ inputs to either true or false on the output. These functions can be represented by a truth table: the output values can (and must) be *configured*. The LUT will load the corresponding output value depending on its inputs and the configuration.

Flip-flops can store the value of the output of a LUT for an arbitrary number of clock cycles. Both are logically bundled into LBs, which are connected with a switchable interconnect. Whether a connection is enabled (meaning the signal can follow the connection) or disabled is also specified in the configuration bits.

Any boolean function can be implemented with these components, provided the FPGA contains enough resources. To decrease the amount of resources that are required for a given circuit, many FPGAs additionally contain hard blocks that implement a dedicated function. One example for such a block is an Arithmetic Logic Unit (ALU) for arithmetic and logic operations. Another common block type is on-chip memory, so called Block Random Access Memory (BRAM).

The truth table, i.e. the boolean function each LUT implements, the initial state of the memory and the active connections of the interconnect must be provided as a *configuration*. The configuration ultimately determines the circuit an FPGA implements. The format of a configuration is binary, thus it is also called the *bitstream*. Which bit corresponds to which resource on the chip is not necessarily disclosed by vendors [49]. This requires reverse engineering efforts. Moreover, it hinders the development of vendor-independent and possibly open software to manipulate bitstreams such as described in [50] or even to generate it from scratch.

The storage of the configuration is based on one of three technologies:

- *Static Random-Access Memory (SRAM)* storage is fast, rewritable and non-persistent. Once the memory loses power, it has to be reinitialized.
- *Flash memory* is a rewritable and non-volatile alternative. A flash memory cell can hold its value without any power supply.
- *Antifuse* is a write-once, non-volatile system, where the link between two conductors is molten by a high current or a laser to form an electrical connection, setting one configuration bit [51].

Accelerators based on FPGAs primarily use SRAM, because the accelerator is constantly supplied with power. Changing the configuration is done by writing to the memory. This process is called *reconfiguration*, independent of the underlying technology. It is possible either via external interfaces like JTAG[1] (and by extension the PCI Express (PCIe) bus for accelerator cards) or with internal interfaces like Internal Configuration Access Port (ICAP) that allow the FPGA circuit itself to set configuration bits [18].

Intuitively, the number of resources and degrees of configurability correlate directly with the size of the required configuration. Each $n$-LUT requires at least $n$ bits of configuration and each switchable connection requires at least one bit. Further, flip-flops require a defined initial state of one bit per flip-flop. Hard blocks may also expect configuration as well as the initial state of BRAMs. As a result, modern FPGAs require bitstreams in the order of megabytes. For example, a bitstream for a Virtex 4 from 2004 is approximately 1 MB [52] while a Intel Stratix 10 GX 10M, the FPGA with the most resources at its introduction in 2019, requires 163.5 MB [53].

The speed of the reconfiguration can vary by orders of magnitudes between devices. It depends on the rate at which the data can be copied to the chip and the size of the configuration. Legacy interfaces like JTAG support slower reconfigurations at 0.03 Gbit/s, whereas optimized and compressed methods like Control via Protocol (CvP) over PCIe support rates of 8 Gbit/s. This means that it takes from 0.01 s up to 2.7 s to apply a bitstream of 10 MB [54], [55].

To reduce the overhead of reconfigurations, a range of techniques are available:

- *Compression* of the bitstream itself [52], [56]–[59].
- *Prefetching* the bitstream from slow memory to faster memory (e.g. BRAM) ahead of time [54], [60], [61].
- *Relocation* of bitstream moving an existing configuration on-chip to other resources, possibly manipulating the configuration in the process [50], [62]–[64].
- *Multi-context FPGAs* as dedicated architectures holding more than one configuration. These offer fast reconfiguration within single clock cycle [65], [66], but were unfortunately never commercially available, due to their high resource and power usage [67].
- *Partial Reconfiguration (PR)* applying only a fraction of the complete bitstream, reducing the size of the bitstream and allowing the unaffected resources on the chip to keep operating during the reconfiguration. This technique can be combined with all the aforementioned.

The most general approach is PR, which is also a prerequisite for almost all scheduling strategies on FPGAs, because it reduces the size of the configuration and provides flexibility as described in Chapter 1.

The influence of the FPGA on task scheduling is extensive:

1. The capabilities of its reconfiguration hardware and the structure of the FPGA limit the speed, the granularity and potential parallelism of the reconfiguration.

[1] Named after the Joint Test Action Group

2. Larger FPGAs need more information in the bitstream, increasing the size.
3. The structure of the FPGA may further restrict the reconfiguration process, especially for PR, since it cannot be applied to arbitrary resources. Instead, only consecutive blocks may be reconfigured. The exact constraints vary per architecture [54].
4. The amount of resources limits the degree of parallelism for accelerated functions.
5. A high utilization may impair the quality of the generated configuration. This could potentially affect the performance of the offloaded functions.

### 2.1.2 Layer 2: Accelerator

An FPGA on its own is of little use, since it needs supporting infrastructure for I/O, power delivery and cooling. This is provided by Layer 2, the accelerator, consisting of an FPGA itself and a range of supporting hardware on a printed circuit board. It contains:

- An interface to the host, typically a connection over a PCIe-bus.
- Off-chip memory modules, e.g. Double Data Rate Synchronous Dynamic Random-Access Memory (DDR SDRAM), and memory controllers.
- Network interfaces that tightly integrate with the FPGA resources
- Further I/O to sensors and antennas
- Power management and cooling infrastructure: a high-end FPGA accelerator emits 255 Watt of heat that must be dissipated [12].

The influence of the accelerator on task scheduling is as follows:

1. The interface to the host limits the data transfer, both for the configuration itself and for data needed for the function execution.
2. The bandwidth of the off-chip memory and the memory controller influence the reconfiguration and the function execution.

### 2.1.3 Layer 3: Host

The accelerator is embedded into a controlling computer, the host system. It includes but is not limited to CPU(s) and main memory. To transfer data to/from an accelerator, a CPU either reads it from main memory and writes it explicitly to the accelerator's memory or vice-versa. Alternatively the accelerator can operate with Direct Memory Access (DMA), removing involvement of the CPU [68, pp. 440–441].

The influence of the host on task scheduling is as follows:

1. The speed, capacity and energy efficiency of the CPUs affect whether offloading is possible and/or beneficial. If the host's CPU performs the task(s) better than an accelerator, then this may be considered by system-wide (hybrid) schedulers[2] like presented by Lui et al. [69] and Tang et al. [70].
2. The bandwidth and latency of the main memory to the CPU and accelerators potentially influences the feasibility of offloading a given task.

[2]Hybrid scheduling is outside the scope of this work.

### 2.1.4 Layer 4: Kernel and Operating System (OS)

The host system runs a kernel and an Operating System (OS) that provides users with an interface to the kernel and by extension to the hardware. Low-level primitives for process and thread management, memory management, synchronization and hardware-software interaction are provided by the OS.

Some approaches for offloading computation integrate FPGAs at this level and in turn implement scheduling:

- Hthreads [45] is a computational architecture that supports reconfigurable devices. Besides of pure *software threads*, it also supports *hardware threads* that are mapped onto an FPGA. It supports HLS by compiling its input to an intermediate representation that is compiled to VHDL [71]. The scheduling is dynamic, not task-based and uses a mix of cooperative and preemptive multitasking and online-scheduling. It implements a priority queue, round-robin and a First In First Out (FIFO) queue. The approach was only implemented on a Virtex II Pro 7 FPGA. It does not use PR [72].
- ReconOS is more evolved. It is a real-time OS with support for hardware threads [44]. Later versions use PR. Different from Hthreads, it is available as open source with support for a range of FPGAs from Xilinx. ReconOS supports preemptive online-scheduling that reads back the bitstream and saves it during a context-switch [73]. Scheduling algorithms are not evaluated, as the scheduling is defined manually.
- FOS is another OS with support for FPGAs [74]. It focuses explicitly on modularity to tackle the aforementioned complexities for software acceleration with FPGAs. The online-scheduling is done non-preemptively in a round-robin fashion. Due to its modularity, FOS is technically agnostic of its underlying hardware. However, it has only been evaluated for three different Xilinx FPGAs, as it is based on earlier works limited to such hardware [28]. It supports PR and is available as open source. FOS also includes drivers and a runtime.

The influence of the OS on task scheduling is as follows:

1. The primitives provided by the OS are the only way for software on the higher layers to interact with the hardware. If the primitives are unsuited for reconfigurable architectures, the performance and flexibility may be affected.
2. The (online) scheduling of (software) processes is typically handled by the OS.

### 2.1.5 Layer 5: Driver

The driver implements model-specific functionality and provides a standardized interface to the users of the system. A proprietary driver is typically provided by the vendor.

The influence of the OS on task scheduling is as follows:

1. Similar to the OS, the interface provided by the by the driver is the only way for software on the higher layers to interact with the hardware. If the driver does not implement certain techniques, the runtime will not have access to them.

### 2.1.6 Layer 6: Runtime

The runtime can act as an interface between the low-level primitives of the OS and the high-level view of a user. It is a process with the purpose of executing user-provided tasks and providing a

uniform and user-friendly interface. It takes dependencies between tasks as an input, starts and monitors their execution, handles the completion of tasks and communicates the tasks' states to the user program. Thus, a runtime may also handle task scheduling itself. Some scheduling approaches are explicitly implemented as a runtime component, others are agnostic to their level of abstraction.

The influence of the runtime on task scheduling is as follows:

1. It limits the interface between the user and the underlying system. If the runtime lacks support for a feature of the underlying system, it is inaccessible by a user.
2. The runtime explicitly handles the scheduling of tasks by one or more scheduling strategies.

## 2.2 Energy Efficiency of FPGA-based Accelerators

After covering the relevant parts of FPGA-based accelerators and their influence on computation offloading in general and task scheduling in particular, we briefly go into the claims about energy efficiency of these accelerators and their relation to task scheduling. Energy efficiency is often cited as one of the most important motivations for the integration of FPGAs into accelerators. Contrary to the impression that could be created, the current state of the art has not formed a consensus about whether FPGAs can provide a level of energy efficiency that would alleviate their disadvantages.

The literature has a range of statements about energy efficiency and does not follow a standardized method of evaluation. Muslim et al. [75] show that the energy efficiency of FPGAs ranges from 0.5× to about 25000× for kernels executed on an FPGA compared to a NVIDIA GTX960 (consumer) GPU. However, the energy usage is measured for the FPGA chip compared to the complete GPU. Struyf et al. [76] find a 6.5× (13.5×) increase in power efficiency over a GPU and the GPU plus the host, respectively. Since the FPGA gets its data from a network-attached host, this comparison highly favors the FPGA.

A more elaborate study was recently performed by Nguyen et al. [77]. In this work systolic arrays are investigated, a technique to structure programs that is favorable for FPGAs. The authors conclude:

> *"We show that although FPGA performance and bandwidth still fall far below GPUs for compute and memory-intensive tasks, the energy efficiency of FPGAs with hardened DSPs is now within a factor of two for SGEMM and SpMV, and in the case of genomics, can exceed that of GPUs by 10%."*

A central motivation for the focus on task scheduling in our work is the correlation between energy usage and performance. Nguyen et al. [77] show that the energy usage of a system does not change drastically per configuration. An observation that is also shown by Cong et al. [78]: in an implementation of the Rodinia benchmark [79], that consists of 15 different kernels, the energy consumption only differs in 2.5 Watt or 12% (21.8 Watt ± 2.5 Watt) between the least and the most energy intensive kernel. At the same time the ratio of performance per Watt of the FPGA ranges from 0.17× to 19.29× compared to a GPU. Qasaimeh et al. [6] describe that FPGAs can implement complex computer vision algorithms with an up to 8× advantage in energy efficiency.

Thus, we do not focus on the modeling and optimization of energy consumption directly, but on the time to solution (or makespan/schedule length) as an approximation. This also includes the energy consumption of the whole system, something that is often ignored in literature regarding the energy efficiency of FPGAs. Measuring and modeling solely the FPGA's energy consumption is not optimal for several reasons:

1. The *total energy consumption* of a system today is much larger than that of an accelerator chip. A typical server power supply is capable of providing upwards of 500 Watt, while the FPGA consumes around 30 Watt up to 150 Watt [78]. Thus, if a schedule's makespan is larger, all other components of the system must be provided with power during the time of the execution, even when they are idle.
2. The *support infrastructure*, i.e. the accelerator, of the FPGA consumes power as well. This seems to be missing in the estimations/measurements in literature, because the power estimation and measurement tools of the FPGA vendors focus only on the chip. Some works like Quasimeh et al. [6] and and Nguyen et al. [77] report the energy to move data separately, while others do not.
3. The *fraction of energy* spent by executing the kernel on FPGAs is rarely set into context of the whole application. For example, if the execution of kernels on FPGAs consumes merely 20% of the energy of a given computation, an energy efficiency advantage of 100% on the device might only result in a 20% total reduction in overall energy usage, even when (2) is ignored.

In the following chapters, we will therefore focus on the scheduling length as the primary metric and argue that it is a good approximation for the total energy consumption.

## 2.3 Scheduling Techniques for FPGA-based Acceleration

The flexibility of FPGAs and their heterogeneity spurred the research into a wide range of techniques for scheduling. We classify them into the following categories:

- Abstraction: on which *stack layer* as described in Section 2.1 – if any – is the approach located?
- Mode: is the scheduling *static* or *dynamic*?
  Static scheduling (or offline scheduling) generates the schedules before its execution and the scheduler has access to all tasks and their dependencies. Dynamic scheduling (or online scheduling) places – potentially newly arriving – tasks during run time.
- Optimality: is the scheduling algorithm *heuristic* or *exact*?
  An exact algorithm or formulation will produce an optimal schedule for the given tasks and hardware (abstraction). Given the NP-completeness of the scheduling problem, this is often only feasible for small problem sizes. Algorithms using a heuristic find solutions that are possibly local optima, but do so in polynomial time.
- Evaluation: on what hardware platform – if any – is the approach evaluated?
  Due to the inherent complexity of scheduling based on PR, many approaches are evaluated via simulation only or on a very limited set of targets. Where an evaluation on hardware was performed, the FPGA models are noted.
- Limited PR: does the approach work with limited PR or does it require task-level PR?
  Since the application of PR maps so well to the scheduling problem, it has been applied in virtually all approaches for task scheduling on FPGAs. However, as detailed in Chapter 1, reliance on PR hinders the portability and adoption by an approach. A limitation that we want to overcome.

We consider approaches for task scheduling on reconfigurable devices that fulfill the following requirements:

- Parallel execution of tasks is possible on one FPGA. Early works did not support the execution of tasks in parallel [80], [81].
  Not only is parallelism a key benefit of FPGAs, but scheduling tasks on a single processor is merely a special case of task scheduling for parallel systems.
- Task scheduling is supported. We exclude approaches that focus on fine-grained structures like loop scheduling or instruction level scheduling.
  These approaches can either be transparently incorporated into coarse-grained tasks or suffer from the same technical hurdles as PR.
- The tasks to be scheduled have no real-time requirement, i.e. no deadline, and do not use preemption.
  Real time (or soft real time) scheduling adds firm time constraints that are unsuitable for offloading tasks.

Table 2.1 lists approaches that meet these criteria ordered ascending by publication date. The first column shows the corresponding publication(s) and the second its year. The "abstraction" column lists the corresponding layer (if any) from Section 2.1. The "mode" is either static, dynamic or both, depending on whether the approach supports static or dynamic scheduling. The "optimality" column shows whether the authors present an exact solution to the scheduling problem, a heuristic-based one or both. The "limited PR" column indicates whether the approach can work with limited PR capabilities as wide-spread HLS platforms support. Finally, the "evaluation" column describes whether the authors evaluate the approach with a simulation or on FPGA-based hardware. If this is the case, the vendor and model is shown. We identify major developments in the field.

1. PR plays a central role in scheduling. *All* approaches require task-level PR. This is unsurprising, since the allocation of resources on a chip directly corresponds to the resource mapping phase during scheduling (see Section 2.3).
2. Numerous approaches are evaluated either by simulation or using a limited number of hardware platforms. This is owed to the fact that the implementation is complex, low-level and vendor-specific.
3. More than 10% of the approaches provide an exact solution together with a heuristic.
4. The integration of task scheduling for FPGAs into operating systems was in focus from 2005, yet it is not commonly found in today's operating systems.
5. Some work makes simplistic assumptions about FPGAs. For example, some approaches model the FPGA as a homogeneous 2d plane and the tasks as arbitrary rectangles to be placed on the plane [82], [118]. However, FPGAs are often structured in rows and columns with heterogeneous interconnects, LBs, hard blocks and I/O [102].

Especially the dependence on PR is a major concern, hindering the adoption of portable scheduling techniques. Vendors must provide open and interoperable interfaces for PR to change this. There is some momentum (and desire) for a more widely supported interoperable Intermediate Representation (IR) for FPGAs [119], [120], but generally there is a tendency towards more closed, proprietary technologies [18].

Table 2.1: Classification of techniques for tasks scheduling on FPGAs

| Publication | | | Mode | | Optimality | | | |
|---|---|---|---|---|---|---|---|---|
| **Ref.** | **Year** | **Layer** | **Static** | **Dynamic** | **Exact** | **Heuristic** | **Lim. PR** | **Evaluation** |
| [82] | 2000 | – | ✓ | ✓ | ✓ | ✓ | ✗ | simulation |
| [83] | 2000 | – | ✗ | ✓ | ✗ | ✓ | ✗ | simulation |
| [84] | 2000 | 2 | ✗ | ✓ | ✗ | ✓ | ✗ | Xilinx XVC400 |
| [85] | 2001 | 6 | ✓ | ✗ | ✓ | ✓ | ✗ | MorphoSys |
| [86] | 2004 | 6 | ✗ | ✓ | ✗ | ✓ | ✗ | simulation |
| [87] | 2004 | 6 | ✗ | ✓ | ✗ | ✓ | ✗ | Xilinx XC2V6000 |
| [88] | 2004 | – | ✓ | ✗ | ✗ | ✓ | ✗ | simulation |
| [89] | 2006 | 6 | ✓ | ✗ | ✓ | ✗ | ✗ | simulation |
| [90] | 2008 | 2 | ✗ | ✓ | ✗ | ✓ | ✗ | simulation |
| [29], [91] | 2008 | – | ✓ | ✗ | ✓ | ✓ | ✗ | simulation |
| [92] | 2009 | 6 | ✓ | ✗ | ✓ | ✓ | ✗ | simulation |
| [93] | 2009 | 6 | ✗ | ✓ | ✗ | ✓ | ✗ | simulation |
| [94] | 2009 | 4 | ✓ | ✓ | ✓ | ✓ | ✗ | simulation |
| [95] | 2010 | 4 | ✗ | ✓ | ✗ | ✓ | ✗ | simulation |
| [96] | 2010 | 6 | ✓ | ✓ | ✗ | ✓ | ✗ | simulation |
| [97] | 2011 | 6 | ✓ | ✓ | ✗ | ✓ | ✗ | Xilinx XC2VP30 |
| [98] | 2012 | – | ✓ | ✗ | ✗ | ✓ | ✗ | Xilinx XC2V6000 |
| [99] | 2012 | 6 | ✗ | ✓ | ✗ | ✓ | ✗ | simulation |
| [69] | 2013 | – | ✓ | ✗ | ✗ | ✓ | ✗ | simulation |
| [100] | 2014 | 4 | ✗ | ✓ | ✗ | ✓ | ✗ | Xilinx XC5VLX110T |
| [101] | 2014 | – | ✓ | ✗ | ✓ | ✗ | ✗ | Xilinx XC5VLX330T |
| [102] | 2014 | – | ✗ | ✓ | ✗ | ✓ | ✗ | simulation |
| [103] | 2014 | 6 | ✗ | ✓ | ✗ | ✓ | ✗ | simulation |
| [104] | 2015 | 4 | ✗ | ✓ | ✗ | ✓ | ✗ | Xilinx XC7Z020 |
| [105] | 2015 | 6 | ✗ | ✓ | ✗ | ✓ | ✗ | Xilinx "Virtex-5" |
| [106] | 2016 | – | ✓ | ✗ | ✓ | ✗ | ✗ | simulation |
| [107], [108] | 2017 | 6 | ✓ | ✗ | ✗ | ✓ | ✗ | simulation |
| [109] | 2017 | – | ✓ | ✗ | ✗ | ✓ | ✗ | simulation |
| [110] | 2017 | 4 | ✓ | ✓ | ✗ | ✓ | ✗ | Xilinx XCKU060 |
| [111] | 2018 | 4 | ✗ | ✓ | ✗ | ✓ | ✗ | Altera 5SGSMD5H2F35I3L |
| [112] | 2020 | – | ✓ | ✗ | ✓ | ✓ | ✗ | Xilinx XC7Z020<br>Xilinx XC7Z045<br>Xilinx XCZU9EG |
| [70], [113] | 2020 | – | ✓ | ✗ | ✓ | ✗ | ✗ | simulation |
| [74] | 2020 | 4 | ✗ | ✓ | ✗ | ✓ | ✗ | Xilinx XCZU3EG<br>Xilinx XCZU9EG |
| [114] | 2020 | – | ✓ | ✗ | ✓ | ✗ | ✗ | simulation |
| [115] | 2021 | – | ✓ | ✗ | ✗ | ✓ | ✗ | simulation |
| [116] | 2021 | – | ✗ | ✓ | ✓ | ✗ | ✗ | simulation |
| [117] | 2023 | – | ✓ | ✗ | ✗ | ✓ | ✗ | simulation |
| This work | 2024 | – | ✓ | ✓ | ✓ | ✓ | ✓ | simulation<br>Xilinx XCU280<br>Intel 1SX280HN2F43E2VG<br>OpenCL |

## 2.4 Genetic Algorithm

In Chapter 5 we present a method to automatically optimize HLS programs for a task graph, a process called DSE. The method uses a Genetic Algorithm (GA) to reduce the search space of possible solutions and to rate existing ones. A GA is an evolutionary algorithm inspired by natural selection. It is particularly well suited for the optimizations of problems that have the following characteristics [121, pp. 116–117]:

- Large search space
- Noisy fitness function
- No "smooth" or unimodal cost function
- No necessity of identifying a global optimum

GAs follow a simple, yet often effective, iterative structure. We describe it briefly here and refer the reader to Chapter 1 of [121, pp. 7–10] for a more thorough introduction to GAs. The peculiarities of our approach are portrayed in Section 5.3. The structure of a GA is the following:

1. A subset of solutions – the *population* – is the starting point. The initial population is often generated randomly. Each chromosome in the population corresponds to a solution to the problem that is being optimized. A chromosome *encodes* information about the solution in *genes*, either as a simple bit string or more complex variables like integers.

2. Each iteration – a *generation* – consist of three steps:

   a) Fitness: each solution in the population – a *chromosome* – is rated according to a *fitness function*. The fitness function is an indicator for the performance of a solution, but does not necessarily have to be the exact target of the optimization. For example, it might not be possible to test the optimality of a solution for each chromosome in the population, because it would be too compute-intensive and thus take too long. Therefore, the fitness function can be a proxy for the optimality of a solution.

   b) Selection: choose chromosomes for the next iteration based on a stochastic process and the fitness. In each generation, pairs of chromosomes – the *parents* – get selected for reproduction. The probability for selection is proportional to each chromosome's fitness.

   c) Reproduction: apply *reproduction operators* (mutation and crossover) to the population. These operators define how a chromosome is changed (mutation) or combined with another chromosome (crossover). With probability $p_c$ each parent pair is crossed over at a random point in the chromosome in the crossover-phase. With probability $p_m$ the value of a gene is changed in the mutation phase.

To generate a set of random solutions for the problem, it must be possible to encode a solution (or an indirect representation) into a chromosome. An *encoding* is a numeric representation of a solution. It determines what information can be mapped by the GA and influences whether the optimal solution may be directly representable or not. The encoding used in this work is *value encoding*. Rather than a bit string, it allows us to encode more complex values into the chromosome, for example integers.

Figure 2.2 shows the process within a generation (step 2 from above). The progress (measured in generations) is depicted from top to bottom. Within each generation and an intermediary step in the middle, three chromosomes are depicted from left to right as rectangles. Each chromosome

consists of five genes, for each of which its current value (integer) is shown. The chromosomes show value encoding. In the crossover phase, a random point in the chromosomes 2 and 3 is selected and the genes after that point are swapped between both parent chromosomes to create the offspring. After that the mutation operator is applied with an (often small) probability $p_m$ to the resulting chromosomes. It replaces a gene randomly in a chromosome. For example, in Figure 2.2 the fourth gene in chromosome 1 is mutated from the value 3 to 1.

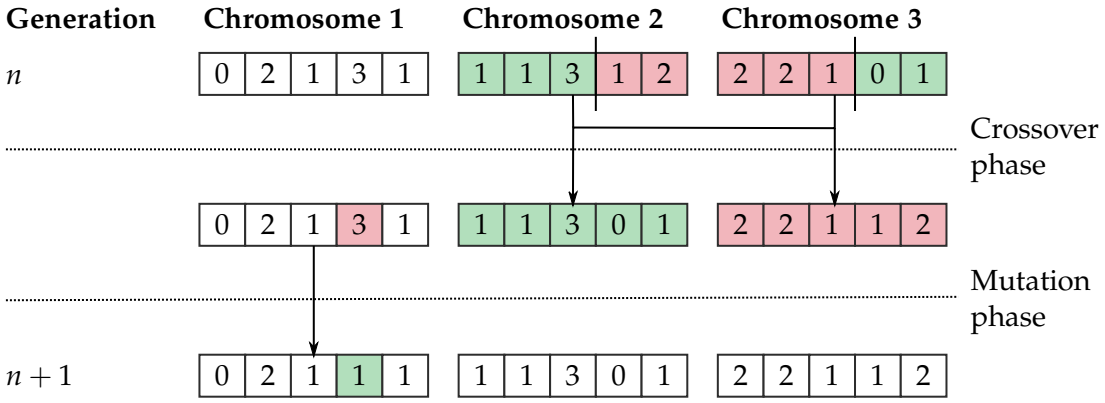



Figure 2.2: Example of two generations in the GA

After a fixed amount of generations, the best solution is selected by its fitness from the resulting population.

Our approach, appropriate encodings, a fitness functions and further details of the selection process are presented in Section 5.3.

## 2.5 Fundamentals of Static Task Scheduling

In this section, we define fundamentals of task scheduling. Existing work regarding task scheduling differs mostly in the notation and other minor details (e.g. whether the first task starts at time 0). We chose to use the notation and mental model from the book:

> *O. Sinnen,* Task scheduling for parallel systems *(Wiley Series on Parallel and Distributed Computing), A. Y. Zomaya, red. Hoboken, NJ, USA: John Wiley & Sons, Inc., Apr. 20, 2007*

Sinnen defines a formal and flexible framework for arguing about scheduling in parallel systems. In particular, the definition allows us to define our own hardware model(s) corresponding to our specific needs. Additionally, adhering to the framework makes it easy to compare existing algorithms. We mark compatible definitions with a black line around the definition and note the corresponding reference in the book. For example, Definition 1.12 would be denoted as Sinnen 1.12 in our definition. Definitions that are merely adjusted are denoted as Sinnen 4.25. Of course, not every definition is directly or indirectly borrowed from Sinnen, since we extend the framework extensively.

**Definition 1** (Task) *A task $v$ is a non-preemptable unit of computational work.*

Since a goal of parallel computing is the parallel execution of applications, these must be split up into smaller *tasks* or units of work, that in turn can be executed in parallel.

Figure 2.3 shows the process of parallelization from an input application to an executable binary from left to right. The respective steps are as follows:

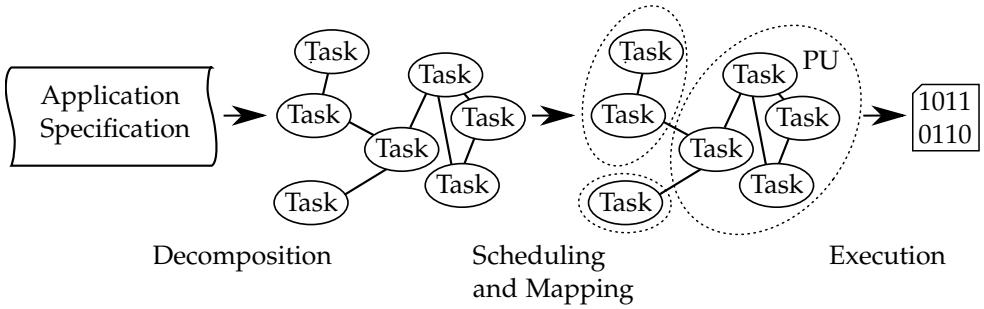



Figure 2.3: The process of parallelization [27, p. 23]

1. The *decomposition* of the application into tasks: this may happen automatically, for example by a compiler, or manually through the directives of a programmer. For our approach, we assume that this decomposition is given.
2. The *scheduling* of tasks: The phase can be split up into the two sub-phases mapping and scheduling:
   - *Mapping* or (resource) allocation is the process of assigning compute resources to a task.
   - *Temporal assignment* is the process of finding an order in which the tasks can be performed.
3. The *execution* on the assigned resources and at the assigned time: the mapping and schedule is taken as an input by a task scheduler software that controls the execution.

The second step, mapping and scheduling, is to decide *where* and *when* the tasks are executed. This process happens before the execution of the tasks starts, because we use static scheduling (see Figure 2.3). Dependencies between tasks imply a partial order between them [27, pp. 23–24] that must be taken into account during scheduling.

**Definition 2** (Processing Element) *A Processing Element (PE) p is a resource that can execute at least one task.*

On conventional computers, tasks are executed on homogeneous processors. Processing on FPGAs (given they do not emulate CPUs) is different:

- The circuit is not directly capable of general purpose processing.
  The hardware function, i.e. the circuit configured by a bitstream, implements a fixed function. Hence, only tasks that expect an implementation for that function can be executed by the circuit under consideration and produce the expected result.
- The resources are not static and not necessarily permanently available.
  Due to reconfigurability, an implementation is only available when a bitstream is loaded (instantiated). Additionally, multiple hardware implementations for one task with varying performance characteristics could exist.

Conventionally, the execution entities are defined as *processors*. The definition of PEs is more general, because the implementation and capabilities on FPGAs is flexible over time, static regarding its function, and heterogeneous. Chapter 3 concerns PEs and their properties in detail.

**Definition 3** (Generic Task Graph) *A program decomposed into tasks can be represented by a generic task graph, a DAG $G' = (V, E)$, where $V$ represent tasks (vertices, nodes) and $E \colon V \times V$ the edges (dependencies). In particular $(v_0, v_1) \in E$ indicates that task $v_0$ communicates to task $v_1$.*

The directed edges, i.e. the communications, define a partial order "$\prec$" between tasks. If $(v_0, v_1) \in E$, then $v_0 \prec v_1$. This relationship is transitive:

$$v_0 \prec v_1 \wedge v_1 \prec v_2 \Rightarrow v_0 \prec v_2 \tag{2.1}$$

Iff a task $v_0$ of a program precedes another task, $v_0 \prec v_1$, $v_0$ has to finish its execution and communicate to $v_1$ before the execution of $v_1$ can be started. This "happened-before" relation "$a \rightarrow b$" is often used in parallel computing to express that $a$ happened before $b$, similar to that $v_0 \rightarrow v_1$ must hold [122]. Nevertheless, we chose to express the order of execution in terms of start and finish times as described in Section 2.6, because it allows us to argue more nuanced about the structure of a schedule. Particularly, it allows us to relax the strict "happened-before" requirement in certain cases, e.g. as demonstrated in Section 3.4.5.4, where the execution of two tasks may overlap. Additionally, all timings of all tasks are known in static scheduling, enabling us to reason about concrete points in time compared to "happened-before".

For a given node $v$ of a task graph $G' = (V, E)$, the nodes with outgoing edges to $v$ are called predecessors $\mathsf{pred}(v)$ and the nodes with incoming edges from $v$ are called successors $\mathsf{succ}(v)$.

$$\mathsf{pred}(v) = \{v_i | (v_i, v) \in E\} \tag{2.2}$$

$$\text{and } \mathsf{succ}(v) = \{v_o | (v, v_o) \in E\} \tag{2.3}$$

To accommodate for the unique characteristics of FPGAs, more concepts must be introduced, before we can formerly define the resource allocation and scheduling itself.

## 2.6 Extensions for Reconfiguration-Aware Scheduling

The generic description of a framework for task scheduling is extended by two concepts:

1. *Configurations and locations* describe bitstreams and their allocation on FPGAs.
2. An *FPGA-aware task graph* contains information that is needed to accurately perform scheduling on a dynamically reconfigurable system.

### 2.6.1 Configurations and Locations

Every PE must be contained in a configuration (a bitstream). A configuration may contain more than one PE, but a PE is always contained in exactly one configuration. Typically, in a PR system, each configuration contains exactly one PE, i.e. it implements one function.

**Definition 4** (Configuration) *A configuration $c = \{p_0, \ldots, p_n\}$ is a set of $n$ PEs $p_0, ..., p_{n-1}$.*

We write $\mathsf{conf}(p) = c$ if $p \in c$.

A configuration must be copied to some part of an FPGA to have an effect, i.e. to make the PEs available for processing. Since systems with PR can have multiple configurations applied at a time or the same configuration multiple times, the concept of a location is introduced.

**Definition 5** (Location) *A location $l \subseteq L$ is a subset of all available locations $L$.*

The available chip area is modeled as a set $L$. The part of the chip, where a configuration can occupy resources is called a location. The granularity at which this is modeled may differ considerably. For example, one could model an FPGA with two slots for PR as $L = \{0, 1\}$ and the locations as location $l_0 = 0$ and location $l_1 = 1$. Or the area can be seen as a two-dimensional plane of dimension of $x_w \times y_h$, then $L = \{(x, y) | 0 \leq x \leq x_w, 0 \leq y \leq y_h\}$.

If a configuration $c$ is configured (instantiated) at a location $l$, we write $\mathsf{loc}(c) = l$. Similarly, $\mathsf{loc}(p) = l$ if $p \in c$ and $\mathsf{loc}(c) = l$.

### 2.6.2 An FPGA-Aware Task Graph

Conventionally – with homogeneous processors – each task has a fixed cost assigned, e.g. its execution time. In a flexible FPGA environment with the set $P$ PEs, the computation and communication depends on many factors, as described above. To account for that, the notion of a task graph is extended by three functions:

- The weight $w\colon V \times P \to \mathbb{N}$ represents a task's *computation cost*.
- The non-negative weight $c\colon E \to \mathbb{N}_0$ represents the *communication cost*.
- The type $t\colon V \to \mathbb{N}$ represents the *class of computations* that a task belongs to.

The weight function $w$ assigns computational costs for combinations of tasks and PEs. An FPGA can provide more than one implementation for a task, similar to how a CPU can contain multiple cores (and execute multiple threads). Different from typical CPUs, the implementations may change in cost. One PE could consume more FPGA resources but execute tasks faster than another one.

The weight function $c$ assigns communication costs to an edge in the task graph. The communication between two tasks is *local* if the PE that executes both tasks is in the same location. Local computations have no communication cost. The meaning of *non-local* communication may vary per use case:

- If each location is connected to a dedicated DDR SDRAM bank, moving the data to a different bank constitutes non-location communication.
- If the FPGA is a network-connected FPGA-based accelerator, each location can model an FPGA and in this case, non-local communication involves network transfer.
- If two or more FPGA-based accelerators are connected to the same host, the host's DDR SDRAM memory can be considered non-local communication, whereas each accelerator's memory is considered local.

The function $t$ assigns each task an identifier for the type of function this task represents. This is useful for tasks that are executed on an FPGA, since its implementation must be generated in a configuration. In contrast to the flexible von Neumann architecture, FPGAs typically only implement a small fixed set of specific functions per bitstream and thus the scheduling algorithm must account for that during the resource allocation.

With the above definitions, an FPGA-specific task graph can be defined as follows.

**Definition 6** ((FPGA-specific) Task Graph) *A program decomposed into tasks can be represented by a task graph, a Directed Acyclic Graph $G = (V, E, w, c, t)$, with:*

- *$V$ and $E$ as defined in Definition 3.*
- *$w$, $c$ and $t$ are functions as described above.*

The definitions based on a generic task graph from Definition 3 can be applied as-is to the task graph from Definition 6.

The process of scheduling tasks consists of two phases:

1. Mapping: tasks need an assigned PE that performs the execution.
2. Temporal assignment: the start (or finish) times for all tasks must be chosen.

**Definition 7** (Resource Allocation) *A resource allocation $\mathcal{A}$ of the task graph $G = (V, E, w, c, t)$ on a finite set $P$ of PEs is the allocation function* $\mathsf{alloc}\colon V \to P \times L$ *of the nodes of $G$ to the PEs of $P$ at locations $L$. The shorthand functions* $\mathsf{proc}\colon V \to P$ *and* $\mathsf{loc}\colon V \to L$ *return the PE and location, respectively.*

A schedule consists of an allocation combined with a function to assign start times to tasks.

**Definition 8** (Schedule) *A schedule $\mathcal{S}$ for the task graph $G = (V, E, w, c, t)$ on a set of PEs $P$ is the function pair $(t_s, \mathsf{alloc})$:*

- $t_s\colon V \times P \to \mathbb{N}$ *is the start time function of the nodes of $G$.*
- $\mathsf{alloc}\colon V \to P \times L$ *is the allocation function of the nodes of $G$ to the PEs of $P$ and locations $L$.*

The node finish time of a task is defined analogous to the start time and depends on its cost and, by extension, the allocation.

**Definition 9** (Node Finish Time) *Let $\mathcal{S}$ be a schedule for task graph $G = (V, E, w, c, t)$. The node finish time of task $v \in V$ on PE $p$ is:*

$$t_f(v, p) = t_s(v, p) + w(v, p) \tag{2.4}$$

Sinnen 6.7

We write the short notation $t_s(v) = t_s(v, p)$ and $t_f(v) = t_f(v, p)$ where it is clear from the context which $p$ is used.

The edge finish time (in contrast to the task finish time) includes the communication cost.

**Definition 10** (Edge Finish Time) *Let $G = (V, E, w, c, t)$ be a task graph. For a given schedule the edge finish time of $e_{ab} = (v_a, v_b) \in E$:*

$$t_f(e_{ab}) = t_f(v_a) + c(e_{ab}) \tag{2.5}$$

The cost function $c$ assumes a uniform cost between all PEs. Below we will also introduce a technique that can be used to reflect the relative distance between the locations and model congestion.

**Definition 11** (Schedule Length) *Let $\mathcal{S}$ be a schedule for task graph $G = (V, E, w, c, t)$. The schedule length is:*

$$\mathsf{sl}(\mathcal{S}) = \max_{v \in V} t_f(v) - \min_{v \in V} t_s(v) \tag{2.6}$$

Sinnen 4.10

Of importance for further analysis of schedules is their length, sometimes called *makespan*. This is the time passed from the start time of the first executed task to the finish time of the last. To make the notation easier, we set the first task to start at 0, i.e. $\min_{v \in V} t_s(v) := 0$, for the rest of this document. With the schedule length we have a metric that can be used to assess the quality of a schedule, since it is an indication of the degree of acceleration and energy consumption as described in Section 2.2. As a basis, the accumulated time of all tasks can be used to normalize the quality of different task graphs.

**Definition 12** (Sequential Time) *Let $G = (V, E, w, c, t)$ be a task graph and $\mathcal{A} = (t_s, \mathsf{alloc})$ an allocation. The graph's sequential time is:*

$$\mathsf{seq}(G) = \sum_{v \in V} w(v, \mathsf{proc}(v)) \tag{2.7}$$

Sinnen 4.12

The parallel time $\mathsf{pt}(\mathcal{S})$ of the schedule is the execution time. The speedup of a schedule $\mathcal{S}$ is the ratio of its sequential time to the parallel time: $\mathsf{speedup}(\mathcal{S}) = \frac{\mathsf{sl}(\mathcal{S})}{\mathsf{pt}(\mathcal{S})}$ [27, p. 222]. It is a measure for the amount of parallelism a schedule achieves and a direct indicator for the amount of acceleration a schedule achieves over the sequential execution of a task graph. Hence, a high speedup is desirable in parallel computing.

The length of a path in a task graph is its accumulated computational and communication cost:

**Definition 13** (Path Length) *Let $G = (V, E, w, c, t)$ be a task graph and a path $p = \langle v_0, v_1, \ldots, \rangle$ with $(v_0, v_1), \ldots \in E$ in $G$, then the length of path $p$ is:*

$$\mathsf{len}(p) = \sum_{v \in p} w(v) + \sum_{e \in p} c(e). \tag{2.8}$$

Sinnen 4.17

The critical path is the longest path in a task graph. It provides a lower bound for the schedule length, since all accumulated costs must be included in a schedule - see Sinnen Lemma 4.4 [27, p. 94].

**Definition 14** (Critical Path) *Let $G = (V, E, w, c, t)$ be a task graph, the critical path $CP$ is the longest path in $G$:*

$$\mathsf{len}(CP) = \max_{p \in G} \mathsf{len}(p). \tag{2.9}$$

Sinnen 4.18

In this chapter we described the fundamentals of program acceleration with FPGAs:

- The *workflow for program acceleration* with FPGAs was briefly described.
- The typical *integration* of an FPGA into a host system for program acceleration was presented with a special focus on each of its component's influence on task scheduling.
- The relation between *task scheduling and energy efficiency* was outlined.
- Existing *approaches for task scheduling* on FPGAs were analyzed.
- The *fundamentals of tasks scheduling* and our extensions ware defined.

With this basis we can now present our first central contribution: a theoretical framework to define FPGA-based accelerators for task scheduling, which allows us to further describe, analyze and optimize task scheduling approaches for FPGAs in Chapters 4 and 5.

# Chapter 3

# Reconfiguration-Aware Scheduling

Since evaluating the potential of task scheduling (algorithms) on FPGAs is hard, as established in Chapter 2, many approaches are evaluated solely through simulations. Others rely on the result and implementation of one FPGA model or vendor. In contrast, we aim to provide a universal and flexible method to analyze and compare scheduling approaches for tasks on FPGAs. The following three chapters constitute the body of this thesis and describe our methodology to analyze task scheduling algorithms for FPGAs, our approach to scheduling on FPGAs and code optimization.

## 3.1 Overview and Methodology

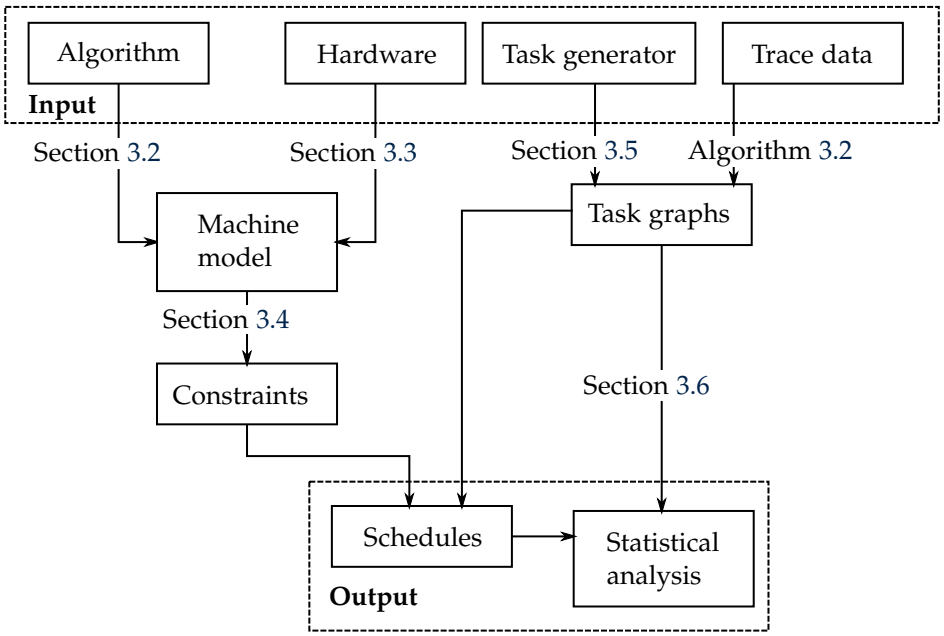



Figure 3.1: Methodology overview

Figure 3.1 depicts the structure of our methodology to evaluate scheduling algorithms. At the top the inputs are shown: scheduling algorithms, hardware descriptions, task generators and trace data. Arrows describe transformations of the inputs to generate intermediate steps during the process. For example, both the scheduling algorithm and a hardware description are transformed into a machine model. Similarly, trace data can be used to supply parameters for task generators. The arrows are annotated to show where the transformation is described in detail.

The outputs can be found at the bottom: statistical analysis can be performed on the generated schedules. The methodology, as depicted in Figure 3.1 to derive, analyze and compare scheduling algorithms is as follows:

- A *machine model* for the target FPGA-based accelerator is defined. We show two sources for the structure and parameters of machine models:
  - The *premises* of a scheduling algorithm about the underlying hardware are converted to a machine model as described in Section 3.2.
  - The *properties* of an existing (or future) piece of hardware are converted to a machine model as described in Section 3.3.
- A set of *task graphs* is gathered. We present two ways to do this:
  - The *task generators* are algorithms that produce task graphs with a predefined structure and parameters as described in Algorithm 3.2.
  - A collection of *trace data* can be used to construct task graphs directly from it as shown in Section 3.5 or to provide parameters (e.g. the cost of tasks) for the task generators.
- The *constraints* are derived from the machine model. Constraints must hold for schedules as described in Section 3.4.
- The constraints are used to *generate schedules* for the task graphs using constraint programming.
- The resulting schedules are *statistically analyzed* in Section 3.6.

These steps are described in more detail below. To compare scheduling approaches with vastly different abstractions, our methodology is based on machine models. A machine model is an abstraction of the target FPGA-based accelerator, its communication system and optionally parts of the host system. It defines which PEs are available to execute tasks and also the performance characteristics of the entire system.

The machine model must account for several factors.

1. The *assumptions* a scheduling algorithm has about the hardware. Scheduling approaches make (or omit) assumptions about the underlying hardware of the target accelerator. For example some approaches assume a homogeneous 2-dimensional plane as a FPGA substrate. Others incorporate more structural information about the chip into their scheduling decisions.
2. The *limitations hardware and software* have. Both the hardware and software itself influence the execution of tasks. For example, if an approach is evaluated for a certain FPGA model, its memory bandwidth should be reflected in the model.
3. The *functions* a bitstream includes. The configuration directly describes the functionality of the FPGA. Each bitstream may implement one or more PEs.
4. The *features and limitations* that result from reconfigurability. It enables fast dynamic replacement of PEs and at the same time introduces a reconfiguration-overhead.

## 3.2 Modeling

The hardware, its capabilities and – if a certain scheduling approach should be analyzed – its assumptions about the hardware are described with models. We strive to describe models that can be held as simple as possible and extended where needed. This is founded in the wide range

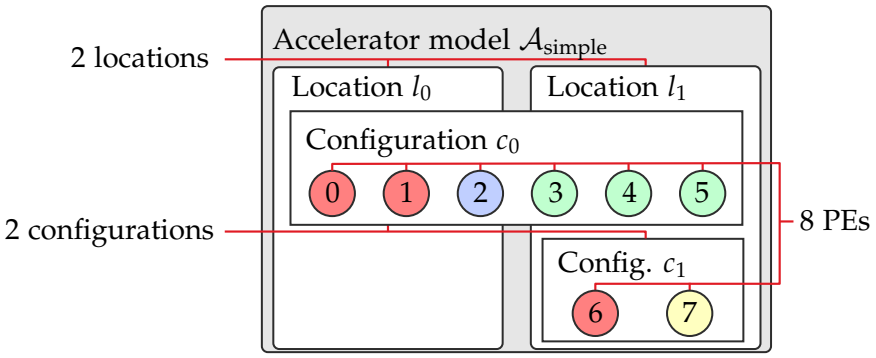



Figure 3.2: An accelerator model $\mathcal{A}_{\text{simple}}$ with two locations, two configurations and a total of eight PEs (circles)

of abstractions that existing task scheduling approaches for FPGAs use (see Section 2.3). On the one hand, a user should be able to construct a simple description of the accelerator hardware and evaluate a scheduling approach on it. On the other hand, a vendor with deep knowledge, incentive and resources could invest a lot of effort into a machine model to match its real-world characteristics as close as possible. Both can then be used as a part of the software stack to aid optimizations, e.g. for task scheduling. Furthermore, if such a model exists for an existing hardware, one can easily simulate changes in the hardware that would otherwise take hours in the case of a synthesis of a configuration and possibly years in the case of new accelerator design.

### 3.2.1 Accelerator Model

The first building block for a machine model is an accelerator model, which describes the computational capabilities and their organization in configurations.

**Definition 15** (Accelerator Model) *An accelerator model $\mathcal{A} = (C, L)$ consists of:*

- *$C$: a set of* configurations. *Each configuration in $C$ consists of a set of PEs. The set $P = \bigcup_{c \in C} c$ denotes the PEs of $\mathcal{A}$. It holds that $\forall c_x, c_y \in C : x \neq y \Rightarrow c_x \cap c_y = \emptyset$, i.e. a PE must only be element of one configuration.*
- *$L$: a set of* locations *(on the FPGA).*

An accelerator model consists of a set of configurations $C$, each of which contains a set of unique PEs for task execution. Locations denote regions to which a configuration can be copied. The notion of locations is intentionally uncertain, since what constitutes a location varies widely between scheduling approaches and software abstractions.

Figure 3.2 depicts an *accelerator model diagram* for an exemplary accelerator model $\mathcal{A}_{\text{simple}}$. Such diagrams can show a simple overview of any accelerator model as an intuitive illustration. Two locations $l_0$ and $l_1$ are shown as rectangles with rounded corners and (two) configurations $c_0$ and $c_1$ are shown as rectangles. The PEs $P_0, \ldots, P_7$ are grouped into configurations and shown as circles containing the PE's index and color representing its *type*. In particular two PEs $P_0$ and $P_1$, represented with a red color in configuration $c_0$, have a matching type with PE $P_6$ in configuration $c_1$. The semantics of types of PEs are described in Section 3.3, as some more definitions are needed. For now, it suffices that the type reflects what functions (see stack layer 6) a PE executes.

**Definition 16** (Instance) *For an accelerator model $A$, an instance $\mathcal{I} = (c, l, i)$ is a 3-tuple of a configuration $c \in C$, a location $l \in L$ and an interval $i = [b, e)$ with begin and end time $b, e \in \mathbb{N}$.*

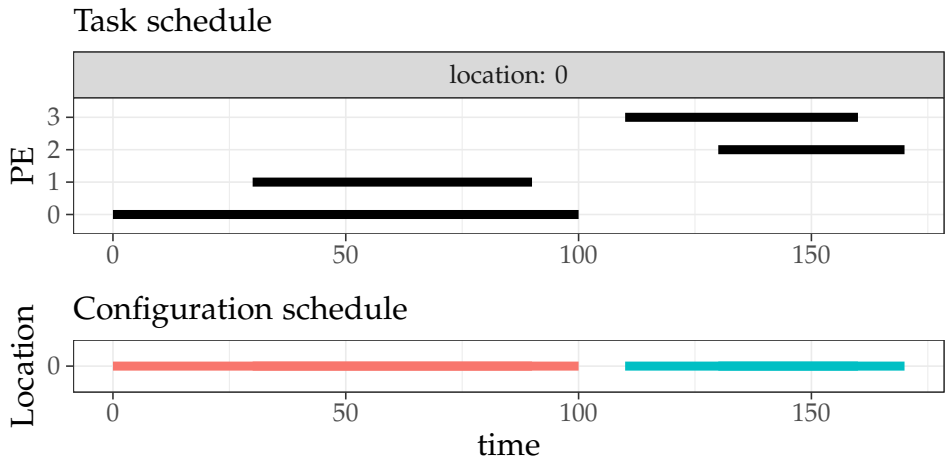



Figure 3.3: Two instances over time with the task schedule on top and the associated instances shown at the bottom

When a configuration (and hence a number of PEs) is configured to a location, so its PEs can execute, it is instantiated. The combination of a configuration, a location and its active interval, is thus called an *instance*. The PEs of a configuration only become available for execution when a configuration is copied to a location and are no longer available when another configuration is written to the same location. A configuration can be instantiated at more than one location simultaneously. The associated PEs are then available for parallel processing.

Figure 3.3 shows two instances with four PEs in total. The time is depicted on the x-axis. The top shows the only location $l_0$ and the tasks that are executed by PEs 0, 1, 2 and 3. Below is the configuration schedule that shows the instances over time, one from time 0 to 100 and one from 110 to 170. One can see that PEs 0 and 1 are contained within one configuration denoted by the red bar, whereas PEs 2 and 3 are contained in the second one shown as the blue bar. Note that the configuration has to be instantiated during the execution on any PE, but the reverse is not true.

#### 3.2.1.1 Applicability of Accelerator Models

The accelerator models are expressive and not limited to FPGAs or even single nodes. The following three examples show the flexibility of the model:

1. A single-region FPGA-based accelerator
2. A hybrid CPU-FPGA architecture
3. A multi-FPGA cluster

**Single-Region FPGA-based accelerator** A *single-region FPGA-based accelerator* is modeled such that the number of configurations $|C| \geq 1$ and the number of locations $|L| = 1$.

There is *one* reconfigurable region (one location $l \in L$) in $\mathcal{M}_\mathrm{R}$ and possibly many configurations that can be instantiated on it. This does not allow partial reconfiguration, but dynamic reconfiguration – even with multiple PEs at a time – is possible.

Figure 3.4 depicts an example accelerator model $\mathcal{M}_\mathrm{R}$: shown is a single region $l$ that contains two configurations $c_0$ and $c_1$. Configuration $c_0$ contains six PEs, three of which have the same type (same, red color). Configuration $c_1$ contains two PEs with a different type each.

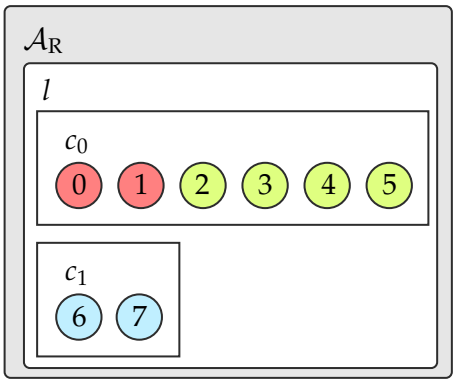



Figure 3.4: Accelerator model diagram for $\mathcal{A}_R$

**Hybrid CPU-FPGA architecture** In a *hybrid CPU-FPGA architecture* $\mathcal{A}_{\text{hybrid}}$ is modeled as follows. The set of configurations $C$ consists of $c_{\text{CPU}}$ and the remaining $c_1, \ldots c_n$. $c_{\text{CPU}} = \{p_{c_0}, p_{c_1}, \ldots p_{c_n}\}$ represent the CPU-Cores.

The model has at least two locations $L = \{l_{\text{CPU}}, l_1, \ldots\}$. The CPU cores are instantiated permanently on $l_{\text{CPU}}$. This can be ensured through the property $\mathcal{P}_c^L(l_{\text{cpu}}) := \{c_{\text{CPU}}\}$ and $\mathcal{P}_c^L(l_i) := C \setminus \{c_{\text{cpu}}\}$ for $l_i \in L \setminus l_{\text{CPU}}$.

This means that all PEs from the configuration $c_{\text{CPU}}$ must reside on one location and the remaining configurations must be configured (more flexibly) on the remaining locations. Communication cost between the CPU and FPGA can be introduced with a cost matrix $c$, that encodes this cost per edge and location pair.

Figure 3.5 depicts an example accelerator model $\mathcal{A}_{\text{hybrid}}$. It shows (the subset of) three locations $l_{\text{CPU}}, l_1, l_2$ and three configurations. The CPU is modeled as the location $l_{\text{CPU}}$ and a configuration $c_{\text{CPU}}$ that contains four PEs (i.e. CPU-cores). Since the cores are flexible w.r.t the work they can perform, their type is not specified, so they are not colored. The other two locations $l_1$ and $l_2$, representing the FPGA(s), can be configured with configuration $c_1$ and location $l_2$ can additionally be configured with location $c_2$. In contrast to $l_{\text{CPU}}$, the PEs in $c_1$ and $c_2$ have associated types and are thus colored.

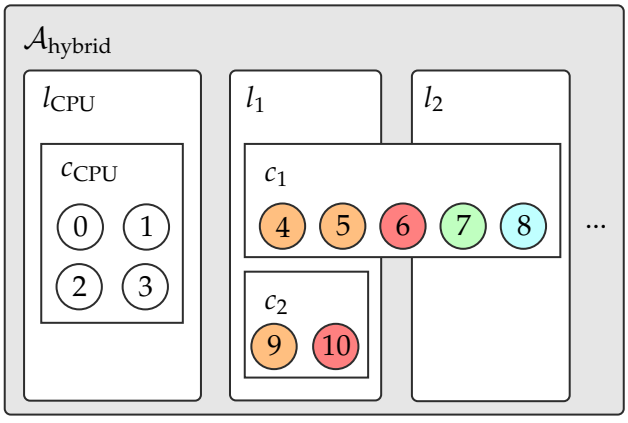



Figure 3.5: Accelerator model diagram for $\mathcal{A}_{\text{hybrid}}$

**Multi-FPGA Cluster** A *multi FPGA cluster* $\mathcal{A}_{\text{multi}}$ is modeled so that each of the $n$ FPGAs in the cluster is represented by a location, so $L = \{l_0, \ldots, l_{n-1}\}$. If the cluster is composed of homogeneous FPGAs, then all configurations can be instantiated on all locations: $\mathcal{P}_{c,p}(p) \coloneqq L$.

Similar to the CPU-FPGA architecture, the communication cost between the locations can be encoded in a cost matrix $c$.

Figure 3.6 depicts an example accelerator model $\mathcal{A}_{\text{multi}}$. It contains a set of locations $l_0, l_1, \ldots$ and two configurations $c_1$ and $c_2$ that can be configured on (all of) them. Each location represent a distinct FPGA. Note that a FPGA cluster with multiple slots on each FPGA can be similarly modeled.

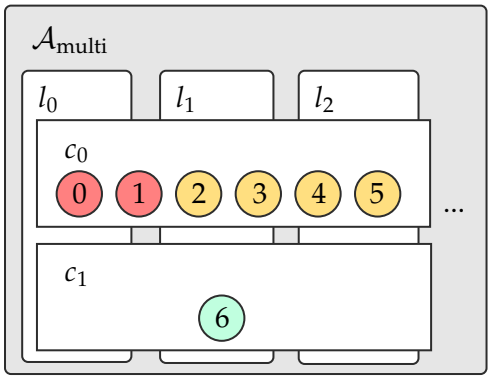



Figure 3.6: Accelerator model diagram for $\mathcal{A}_{\text{multi}}$

### 3.2.2 Communication Topologies

Accelerator models by themselves can only be used for scheduling algorithms that do not account for communication between tasks at all or make simplistic assumptions about the underlying hardware. A typical FPGA-based accelerator can have a wide range of connections, especially but not limited to the case where it is equipped with network interfaces. Other connections include the (accelerator-) global RAM, the PCIe-link and interfaces of PEs to a location's global RAM.

The structure of these connections and their capabilities have a large influence on the accuracy of schedules, i.e. the similarity between the generated (projected) schedules and the execution in a real-world setup. To increase the accuracy, two factors have to be incorporated into the model, a topology and a network scheduling model:

1. The *topology* describes the structure and the capabilities (in terms of bandwidth between communicating nodes).
2. The *communication scheduling model* describes the influence of the topology on the (viable) schedules.

Since the accelerator models are an extension to Sinnen's framework for scheduling, both the topology and the communication scheduling model can be borrowed and adjusted from [27].

The connections between the communicating entities (PEs) are not always point-to-point connections. They are instead described as a network, which in turn is modeled as a set of vertices and edges. Note that the topologies in FPGA-based accelerators are typically simple compared to full-fledged network applications.

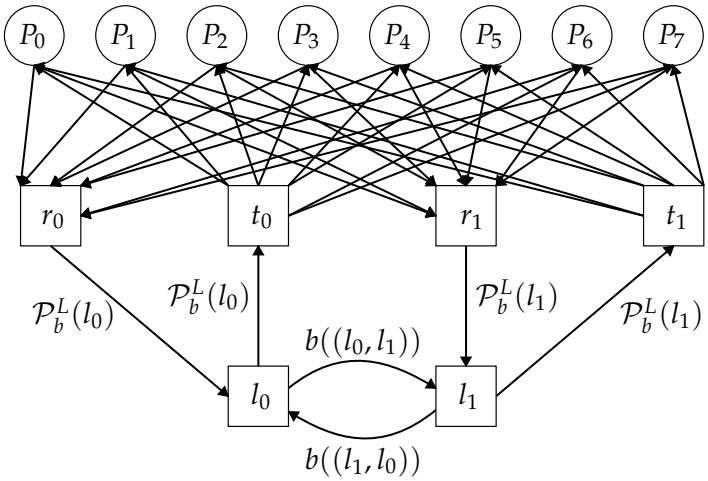



Figure 3.7: A default topology $\mathcal{T}_{\text{default}}$ for the accelerator model $\mathcal{A}_{\text{simple}}$ in Figure 3.2

**Definition 17** (Topology) *A (communication) topology is modeled as a graph $TG = (N, O, D, b)$, where $N$ is a finite set of vertices, $O$ is a subset of $N$, $O \subseteq N$, and $D$ is a finite set of directed edges. A vertex $n \in N$ is referred to as a network vertex, of which two types are distinguished: a vertex $o \in O$ represents a PE, while a vertex $s \in N, \notin O$ represents a switch. A directed edge $D_{ij} \in D$ represents a directed communication link from network vertex $n_i$ to network vertex $n_j$ and $n_i, n_j \in N$. The weight $b(d)$ associated with a link $d \in D$, $b\colon D \to \mathbb{Q}^+$, represents its relative data rate.*

Sinnen 7.3

The topology is described as a graph consisting of the vertices $N$ and a subset of graph $P$, which in turn describes the PEs, in contrast to the switches. The set of links $K$ describes paths through the network as a combination of vertices and edges.

#### 3.2.2.1 Default Topology

The accelerator model can now be accompanied by a topology to describe the underlying communication hardware. If not specified explicitly, a *default topology* is used as described below. Given an accelerator model $\mathcal{A} = (C, L)$, a default topology $\mathcal{T}_{\text{default}}$ consists of:

- $N = P \cup L \cup R \cup T$: the set of network vertices consists of PEs, locations and one *transmit node* $t \in T$ and one *receive node* $r \in R$ per location $l \in L$, i.e. $|T| = |R| = |L|$.
- $O = P$: the set of non-switch nodes is equal to the set of PEs.
- $D = PR \cup RL \cup LT \cup TP \cup LL$, where
  - $PR = \{(p, r) \mid p \in P, r \in R\}$,
  - $RL = \{(r, l) \mid r \in R, l \in L\}$,
  - $LT = \{(l, t) \mid l \in L, t \in T\}$,
  - $TP = \{(t, p) \mid t \in T, p \in P\}$, and
  - $LL = \{(l_f, l_t) \mid l_f, l_t \in L, l_f \neq l_t\}$.
- $b$ is set according to the properties $\mathcal{P}_b^P$ and $\mathcal{P}_b^L$ as described in Section 3.3.

Figure 3.7 shows an example of an accelerator model $\mathcal{A}$ and its corresponding default topology $\mathcal{T}_{\text{default}}$. The set of non-switch nodes $O$ (equal to the set of PEs $P$) is depicted as circles and

switches (all other nodes $N \setminus O$) as squares. Both location $l_0$ and location $l_1$ are connected with a directed communication link and both have transmit nodes $t_{\{0,1\}}$ and receive nodes $r_{\{0,1\}}$, respectively. The transmit and receive nodes allow the modeling of bandwidth limitations and congestion independently of the PEs. Also, often PEs do not communicate directly (for instance via OpenCL Pipes) but write data to their location's global memory. For example, all PEs $P_0, \ldots, P_7$ could each have a maximum bandwidth of $u$, but the global memory at location $l_0$ provides $2 \cdot u$ in total. Therefore, if all 8 PEs were to communicate at the same time, the necessary bandwidth of $8 \cdot u > 2 \cdot u$ would lead to congestion. The bandwidth is noted at the edges, except for edges to and from PE nodes: for clarity the labels $b((P_m, r_n))$ and $b((t_n, P_m))$, where $0 \leq m \leq 7$ and $0 \leq n \leq 1$, are omitted.

Typically, the bisection bandwidth is used to identify bottlenecks in a communication network. It is the minimum bandwidth across any cut that splits the network into two subnets with an equal number of non-switch nodes [123, p. 1156]. This is typically used in undirected graphs, but we can extend the bisection bandwidth for directed graphs using a similar description.

**Definition 18** ((Directed) Bisection bandwidth) *Let $\mathcal{T} = (N, O, D, b)$ be a topology and the sets $S$ and $T$ with $S \cup T = O$ and $S \cap T = \emptyset$ form two partitions of the PE nodes $O$ with a difference of at most one in size, i.e. $0 \leq ||S| - |T|| \leq 1$. The (directed) bisection bandwidth is the minimal bandwidth available between any two sets $S$ and $T$ according to $b$.*

Suppose the bandwidth $b$ is 1 for all edges a default topology $\mathcal{T}_{\text{default}}$, then the bisection bandwidth is 1. The communication from any PE $P_s \in S$ to PE $P_t \in T$ must flow through a single edge from its location's receiver node and transmit node, respectively. If the topology $\mathcal{T}_{\text{default}}$ consists of more than one location, the communication must flow through the connection between locations (the edges in $LL$). In a typical setup, the (single) edge from receiver/transmit nodes will indeed be a bottleneck, since it models the connection to the global memory.

#### 3.2.2.2 Ring-Connected Topology

A different topology, for modeling inter-FPGA communication networks, is shown below. Suppose accelerator model $\mathcal{A}_{\text{ring}}$ consists of three FPGA-based accelerators that are connected in a full duplex ring topology, represented by three locations $l_0, l_1$ and $l_2$. For brevity, we assume only one configuration with one PE per location can be instantiated. The topology $\mathcal{T}_{\text{ring}}$ is then:

- $N = \{P_0, P_1, P_2\} \cup \{r_0, r_1\} \cup \{t_0, t_1\}$: same as the default topology
- $O = \{P_0, P_1, P_2\}$: same as the default topology
- $D = \{((P_0, r_0), (r_0, l_0), (l_0, t_0), (t_0, P_0), \ldots, (t_2, P_2), (P_0, P_1), \ldots, (P_1, P_0), \ldots\}$
- $b$ is set equal to the default topology and additional values for the links $(P_0, P_1), \ldots (P_1, P_0)$.

Figure 3.8 shows a corresponding topology $\mathcal{T}_{\text{ring}}$ for accelerator model $\mathcal{A}_{\text{ring}}$. The PEs $P_0, P_1$ and $P_2$ are connected in a ring topology. Each of the PEs is connected to its location accumulation node, which is in turn connected to the location node (representing the global memory). This topology could be simplified by removing the accumulation nodes altogether, since the connections to the location nodes are not shared. Conversely, if there were more PEs per location, accumulation nodes could be inserted between the PEs ring connection (red).

The bisection bandwidth for $\mathcal{T}_{\text{ring}}$ is independent of the location bandwidth. If the ring's edges (depicted as red arrows in Figure 3.8) have bandwidth $b$ of 1, then the bisection bandwidth is 2 – similar to a ring topology in an undirected graph.

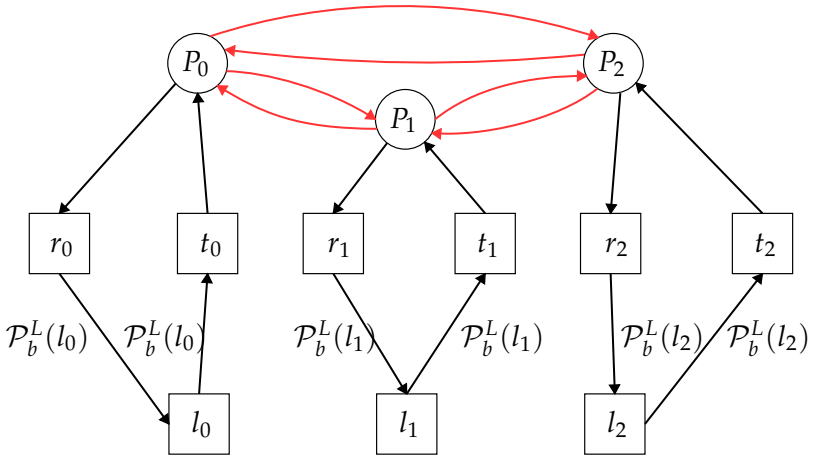



Figure 3.8: Ring topology $\mathcal{T}_{\text{ring}}$ applied to the accelerator model $\mathcal{A}_{\text{ring}}$

#### 3.2.2.3 CPU-FPGA Topology

Another example is the topology $\mathcal{T}_{\text{hybrid}}$ for a hybrid CPU-FPGA model $\mathcal{A}_{\text{hybrid}}$ as depicted in Figure 3.5. As described in Chapter 2, the FPGA-based accelerator card is connected with, for example, PCIe or Compute Express Link (CXL) to the host system. The CPU and the accelerator share data across this connection and thus it must be part of the topology. There are several scenarios one can represent as a topology:

1. A system with *coherent memory*

   Modern interconnects like CXL offer cache coherency across different memory subsystems like the host memory and the accelerators global memory [124, pp. 83 ff.]. In this case the topology can be described with additional connections from $l_{\text{CPU}}$ to the accelerator's global memory.

2. A system with *explicit memory transfers*

   Explicit memory transfers from host to the accelerator and vice-versa can be described by an additional switch node, so the connection from the host memory to the accelerator (global memory) can be parameterized and modeled.

3. A system with a *complex cache hierarchy*

   Current CPUs contain multiple levels of caches (the so-called cache hierarchy). Each level has its own bandwidth characteristics which can be described as a topology and therefore considered during scheduling.

Figure 3.9 shows the first scenario for a hybrid accelerator model $\mathcal{A}_{\text{hybrid}}$. The CPU is modeled as a separate location $l_{\text{CPU}}$ with the corresponding receive and transmit nodes $r_{\text{CPU}}$ and $t_{\text{CPU}}$, respectively. The PEs $P_0$ and $P_1$ are connected to this location, whereas PEs $P_2$ and $P_3$ are connected to location $l_0$ through its receive and transmit nodes. The locations $l_{\text{CPU}}$ and $l_0$ are connected and the relative bandwidth is shown as "CXL", representing the bandwidth a CXL link provides relative to the other components. Different from a default topology, the PEs are not connected to the receive and transmit nodes of both locations, since a CPU-core cannot be instantiated at an FPGA-based accelerator and vice-versa.

Suppose every link in the topology $\mathcal{T}_{\text{hybrid}}$ has bandwidth 1 and the CXL-link a bandwidth of $C$, then the bisection bandwidth is $\min(1, C)$.

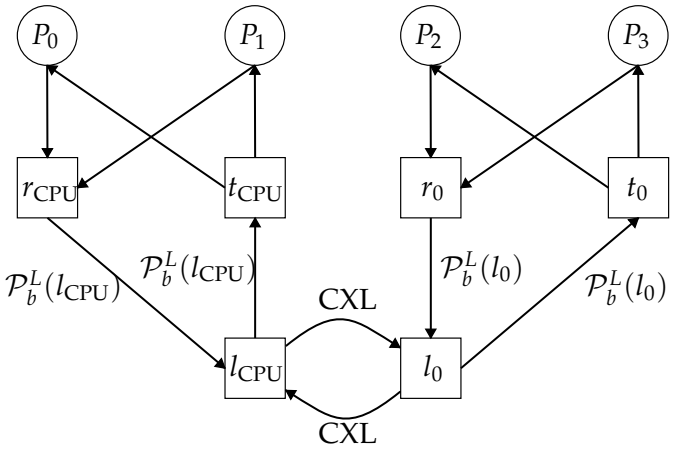



Figure 3.9: Hybrid CPU-FPGA topology $\mathcal{T}_{\text{hybrid}}$ applied to the accelerator model $\mathcal{A}_{\text{hybrid}}$

### 3.2.3 Machine Model

An accelerator model alone or a topology alone are of limited value to solve the scheduling problem accurately. Instead, we combine them both and introduce a machine model that both includes the topology and an accelerator model. The accelerator model describes the compute capabilities of the machine and the topology its communication capabilities. The information about the modeled system is only of structural nature, i.e. describing the organization of PEs and their connections to each other and other parts of the system. The modeled hardware, however, has additional characteristics. These range from link bandwidths, constraints on the capabilities of the PEs, up to reconfiguration time overhead.

**Definition 19** (Machine Model) *A tuple $\mathcal{M} = (\mathcal{A}, \mathcal{T}, \mathcal{P})$ is a machine model, where:*

- *$\mathcal{A} = (C, L)$ is an accelerator model,*
- *$\mathcal{T} = (N, O, D, b)$ is a topology, and*
- *$\mathcal{P}\colon X \multimap Y$ is a multifunction, where $X = \mathcal{A} \cup N \cup D \cup C \cup P \cup L$. A multifunction or correspondence $f\colon A \multimap B$ maps elements of $A$ to subsets of $B$, i.e. $f\colon A \to \mathbb{P}(B)$. We write $\mathcal{P}_Z^{X'}$ to denote a function $X' \to Y'$, where $X' \subset X$ and $Y' \subset Y$. $Y$ is the codomain for a property of $Z$, which is described below.*

  *$\mathcal{P}_Z$ assigns a set of properties $Z$ to accelerator models, network nodes, network links, configurations, PEs, and locations, respectively. If not specified differently, $O = P$ is assumed throughout this document.*

The third and final piece of a machine model is therefore a property function $\mathcal{P}$ that allows us to assign any kind of property to any entity of the machine model. For example, defining the bandwidth $b$ of $x$ for a link $d \in D$ in the topology $\mathcal{T}$, we can specify $\mathcal{P}_b^D(d) = x$. By using the information provided by properties, algorithms may enhance the search for optimal solutions. The properties of machine models are described below.

## 3.3 Properties of Machine Models

The machine model and, in particular, two of its subparts, the accelerator model and the topology, describe the structure of the accelerator. It may also contain elements of the host system if it is

necessary, e.g. to achieve higher accuracy of scheduling algorithms or to reflect a central role of the host system in the scheduling process as in a machine model constructed with the hybrid CPU-FPGA accelerator model $\mathcal{A}_{\text{hybrid}}$. Machine models describe which PE is contained in which configuration in $C$. But this information is not sufficient to accurately describe an FPGA-based accelerator complete with its host interface. For example, the types (colors in the diagrams) have not been described by the machine models above.

Properties are a flexible way to describe the quantitative, non-structural characteristics of a machine model (and by extension of the accelerator model and topology). The property multifunction $\mathcal{P}$ takes an element from a machine model $\mathcal{M}$ as an input and returns a set of values, describing the properties of the object. In particular $\mathcal{P}_Z$ describes the property $Z$ and $\mathcal{P}_Z^X(x)$ gives the value of property $Z$ for an element $x \in X$. Note that $\mathcal{P}_Z^X$ may be defined for two disjunct sets $X$. In other words: a type of property ($Z$) may be used for more than one entity. For example, the bandwidth $b$ is used for PEs and links.

Be aware that this list is not considered complete. If anything, users applying our approach are expected to add their own properties as needed. Also, properties are described informally in terms of their real-world representation. Their (more formal) effect on schedules as well as the process of introducing new properties is described in Section 3.4.

A set of properties used in this work are described below. Properties can be defined for all entities (sets) in a machine model, but we limit ourselves to properties for PEs $P$, configurations $C$, locations $L$, and links $D$. It is also possible to construct properties recursively. For example, the properties describing the resource usage of PEs in one configuration can simply be summed up to get the resource usage of the configuration. This example is described in Section 5.4.

### 3.3.1 Processing Element Properties

Since machine models are build around PEs, their definition contains a set of *PE properties* $\mathcal{P}^P$. Properties are used to define the characteristics of the hardware or assumptions the software makes about the hardware. For example, a PE may have a limited bandwidth to the main memory that is used to communicate with PEs instantiated to another location. This limitation can be described as a property.

| Property | Identifier | Codomain | Default | Example |
|---|---|---|---|---|
| Collision | $\mathcal{P}_c^P$ | $P$ | $\emptyset$ | $\{p_3, p_8\}$ |
| Dependency | $\mathcal{P}_d^P$ | $P$ | $\emptyset$ | $\{p_1\}$ |
| Type | $\mathcal{P}_t^P$ | $\mathbb{N}$ | 0 | 4 |
| Start overhead | $\mathcal{P}_s^P$ | s | 0 s | 112 ns |

Table 3.1: Possible PE properties for a set of PEs $\{p_0, p_1, \ldots, p_n\} = P$

Table 3.1 shows one possible PE-property per row. The identifier column shows the name of the property (function). All properties in the table apply to PEs, so the upper index is $P$, i.e. $\mathcal{P}^P$. The lower index identifies the property, e.g. $c$ for the collision property. The codomain for each property is shown in the respective column. For example, collisions are described in terms of elements from the set of PEs $P$ – the colliding PEs. The default value is used if no explicit value is specified for a property. For example, if a PE $p$ has no collisions specified, then $\mathcal{P}_c^P(p) = \emptyset$. The last column shows an example for each property. The PE $p$ collides with PE $p_3$ and $p_8$ in the example $\mathcal{P}_c^P(p) = \{p_3, p_8\}$. The list of properties may be extended as described in Section 3.4.

The semantics of the properties from Table 3.1 are the following:

- The *collision* property $\mathcal{P}_c^P$ denotes PEs that may not be instantiated simultaneously.
- The *dependency* property $\mathcal{P}_d^P$ denotes PEs that must be instantiated simultaneously.
- The *type* property $\mathcal{P}_t^P$ denotes the type of tasks that a PE can execute. If a type is not set, all tasks can be executed.
- The *start overhead* property $\mathcal{P}_s^P$ denotes the overhead of executing an empty task on a PE.

### 3.3.2 Configuration Properties

Configurations in the accelerator model define how PEs are combined into bitstreams. *Configuration properties* describe characteristics of such a configuration. Similar to PE properties, configuration properties are used to specify hardware characteristics – in this case of a configuration.

| Property | Identifier | Codomain | Default | Example |
|---|---|---|---|---|
| Collision | $\mathcal{P}_c^C$ | C | $\emptyset$ | $\{c_1\}$ |
| Dependency | $\mathcal{P}_d^C$ | C | $\emptyset$ | $\{c_1\}$ |
| Placement | $\mathcal{P}_p^C$ | $L$ | L | $\{l_0, l_1\}$ |
| Size | $\mathcal{P}_s^C$ | bit | 0 | 433 kB |

Table 3.2: Configuration properties with $c_1 \in C$

Table 3.2 shows possible configuration properties. The columns are equal to Table 3.1. The property identifiers have the upper index $C$, denoting the applicability to configurations.

The semantics of the properties from Table 3.2 are the following:

- The *collision* property $\mathcal{P}_c^C$ denotes configurations that must be instantiated simultaneously (at other locations).
- The *dependency* property $\mathcal{P}_d^C$ denotes configurations that must not be instantiated simultaneously at other locations.
- The *placement* property $\mathcal{P}_p^C$ denotes all possible locations at which a configuration may be instantiated.
- The *size* property $\mathcal{P}_s^C$ represents the size of the bitstream file.

### 3.3.3 Location Properties

Similar to PE- and configuration properties, location properties describe characteristics of a certain location. For example, a location may only accept new bitstreams with a given bandwidth or a fixed overhead.

| Property | Identifier | Codomain | Default | Example |
|---|---|---|---|---|
| Bandwidth | $\mathcal{P}_b^L$ | bit/s | $\infty$ | 15 Mbit/s |
| Reconfiguration delay | $\mathcal{P}_r^L$ | s | 0 s | 13 ms |
| Configurations | $\mathcal{P}_c^L$ | $C$ | $C$ | $\{c_1, c_3\}$ |

Table 3.3: Location properties with $c_1, c_3 \in C$

Table 3.3 shows possible configuration properties. The columns are equal to Table 3.1. The property identifiers have the upper index $L$, denoting the applicability to locations.

The semantics of the properties from Table 3.2 are the following:

- The *bandwidth* property $\mathcal{P}_b^L$ denotes the bandwidth at which the location reads/instantiates configurations.
- The *reconfiguration delay* property $\mathcal{P}_r^L$ denotes the time to instantiate a configuration.
- The *configurations* property $\mathcal{P}_c^L$ denotes a list of configurations that can be instantiated at the location, i.e. it is the counterpart of $\mathcal{P}_l^C$.

The arrangement as seen in the accelerator model diagrams in Figures 3.4, 3.5 and 3.6 can now be described in terms of properties and vice versa. When a configuration overlaps with a location in such a diagram, the configuration can be instantiated at this location. This means if for location $l$ and a configuration $c$: $c \in \mathcal{P}_c^L(l)$, then $c$ will overlap $l$ in an accelerator model diagram. PEs from a configuration can thus be configured to all locations that are overlapped by the configuration box. Going back to Figure 3.2 on page 33, it shows that configuration $c_0$ can be instantiated at location $l_0$ and $l_1$, whereas configuration $c_1$ may only be instantiated at location $l_1$. Similarly, the color of a circle representing a PE $p$ shows its type that is equal to its type property $\mathcal{P}_t^P(p)$.

## 3.4 Constraints on Schedules

Clearly, not every schedule is valid. Consider the precedence relation $\prec$. If a task $v_1$ starts executing before a preceding task $v_0$ executes, then it cannot receive the communication from $v_0$ and will yield an invalid result. Thus, a schedule $\mathcal{S}$ is subject to constraints to make it viable. These can be split up into three categories for our purposes:

1. *General constraints* inherent to task scheduling.
2. *Machine constraints* resulting from the restrictions of the target machine model, i.e. from PEs, configurations, locations and topologies.
3. *Communication constraints* resulting from the communication setting.

The following sections go over the derivation of constraints for all three categories. Section 3.4.1 highlights general constraints that must hold for all schedules. Sections 3.4.2, 3.4.3, 3.4.4 and 3.4.5 describe how machine constraints are derived for each entity (PEs, configurations, locations and topologies). In Section 3.4.5 we also illustrate a range of communication settings and their effect on the set of constraints. The summary in Section 3.4.6 and Table 3.7 show an overview.

### 3.4.1 General Constraints

The first category of conditions must hold for all schedules, independently of the target machine model. There are two general constraints:

- Exclusive PE allocation
- Exclusive instantiation

The *exclusive PE allocation* constraint ensures that at any time each PE $p \in P$ executes at most one task $v \in V$. If a PE $p$ was capable of executing multiple tasks concurrently (or even truly in parallel), it must be modeled as multiple PEs $p_0, \ldots, p_n$. Two tasks overlap, if there exists a point in time, when both are executing. The predicates $\mathsf{overlap}_b$ and $\mathsf{overlap}$ denote this, with and without a buffer between the tasks, respectively. Given two tasks $v_a, v_b \in V$, then

$$\begin{aligned}\mathsf{overlap}_b(v_a, v_b, b) =& t_s(v_b) + b \geq t_s(v_a) \wedge t_s(v_b) \leq t_f(v_a) + b \vee \\ & t_s(v_a) \geq t_s(v_b) + b \wedge t_s(v_a) \leq t_f(v_b) + b \text{ and} \qquad (3.1)\\ \mathsf{overlap}(v_a, v_b) =& \mathsf{overlap}_b(v_a, v_b, 0). \qquad (3.2)\end{aligned}$$

**Constraint 1** (Exclusive PE allocation) *Let $\mathcal{S}$ be a schedule for task graph $G = (V, E, w, c, t)$ on PEs $P$. For any two nodes (tasks) $v_a, v_b \in V$:*

$$\mathsf{proc}(v_a) = \mathsf{proc}(v_b) \Rightarrow \neg\, \mathsf{overlap}(v_a, v_b) \qquad (3.3)$$

A location can hold exactly one configuration, which is described in the following constraint.

**Constraint 2** *Let $\mathcal{S}$ be a schedule for task graph $G = (V, E, w, c, t)$ on machine model $\mathcal{M} = (\mathcal{A}, \mathcal{T}, \mathcal{P})$. For any two instances $i_a = (c_a, l_a, [b_a, e_a))$ and $i_b = (c_b, l_b, [b_b, e_b))$:*

$$c_a = c_b \Rightarrow b_a > e_b \vee e_a < b_b \qquad (3.4)$$

This is similar to constraint 1, only instead of tasks it mandates no overlap of instantiated configurations at a given location.

**Definition 20** (Generally feasible schedule) *A schedule $\mathcal{S}$ is called generally feasible, if it fulfills both constraints 1 and 2.*

We require all schedules to fulfill constraints 1 and 2, since any schedule violating either one could not reliably compute the desired result. We call these schedules *generally feasible*. However, other constraints need to be fulfilled by a schedule that can be considered *valid*. The following sections describe such constraints and how they are constructed.

In Section 2.3 we established that not all possible schedules are valid. Aside from constraints 1 and 2, the machine model also influences the solution space for valid schedules. Thus, the machine model $\mathcal{M}$ must be converted to a set of *machine constraints* $\mathcal{C}$ on the schedule.

**Definition 21** (Feasible schedule) *A generally feasible schedule $\mathcal{S}$ is called* feasible *if it fulfills all machine constraints $\mathcal{C}$.*

Each property may be transformed into one or more constraints. After the conversion, all feasible schedules violate none of the machine model's properties, i.e. a feasible schedule $\mathcal{S}$ can be executed on a machine modeled by the input model $\mathcal{M}$. We demonstrate this process in a case study in Section 3.7.

A set of *constraint derivation rules* $\mathcal{R}$ is used to generate constraints in a systematical manner. The constraint derivation (multi-)function $\mathsf{const}_\mathcal{R} : \mathcal{P} \multimap \mathcal{C}$ maps each property to a set of constraints using the rule set $\mathcal{R}$.

A machine model can be converted to a set of constraints with the procedure depicted in Algorithm 3.1. The procedure takes a machine model and a set of constraint derivation rules as input. It iterates over all entities of the machine model and applies $\mathsf{const}_\mathcal{R}$ for all properties for each entity. The resulting constraints are combined into a set of constraints $K$.

While Algorithm 3.1 shows the universal process to generate machine constraints from a machine model, the concrete constraint derivation rules depend on the properties of the machine model. The following sections show the derivation rules for the properties introduced in Section 3.3.

**Input:** Machine model $\mathcal{M} = (\mathcal{A}, \mathcal{T}, \mathcal{P})$; constraint derivation rule set $\mathcal{R}$
**Result:** Set of constraints $K$
$Z \leftarrow$ all properties in $\mathcal{M}$ 1
**foreach** $x \in X$ **do** 2
  **foreach** $z \in Z$ **do** 3
    $K \leftarrow K \cup \mathsf{const}_{\mathcal{R}}(\mathcal{P}_z^X(x))$ 4
  **end** 5
**end** 6
**return** $K$ 7

Algorithm 3.1: Generating the complete set of machine constraints $K$ for a given machine model $\mathcal{M}$

### 3.4.2 Machine Constraints from PEs

Table 3.4 shows the derivation rules for properties of PEs. Each row contains one role, corresponding to one property. For each property it lists the property name, its identifier and the constraint derivation rule that must be applied to create the constraint set. For example, the first row shows the constraint to be generated from the (PE) collision property $\mathcal{P}_c^P$. Assuming a collision constraint $\mathcal{P}_c^P(p) = \{p_2, p_3\}$ is defined for a PE $p$, then the resulting constraint rule is

$$\forall x \in \{p_2, p_3\} \colon \neg\, \mathsf{overlap}(p, x). \tag{3.5}$$

In other words, all feasible schedules must ensure that as long as PE $p$ is instantiated, neither $p_2$ nor $p_3$ is instantiated at the same time.

The other rules in the table – and the following tables – are derived similarly, although some make use of other information, such as the task graph. For example, the rule for the (PE) type property $\mathcal{P}_t^P$ uses the set of locations $L$ from the machine model and the set of tasks $V$ from the task graph to create a constraint.

| Property | Identifier | Rule |
|---|---|---|
| Collision | $\mathcal{P}_c^P$ | $\forall x \in \mathcal{P}_c^P(p) \colon \neg\, \mathsf{overlap}(p, x)$ |
| Dependency | $\mathcal{P}_d^P$ | $\forall x \in \mathcal{P}_d^P(p) \colon\ \mathsf{overlap}(p, x)$ |
| Type | $\mathcal{P}_t^P$ | $\forall l \in L, \forall v \in V \colon t(v) \neq \mathcal{P}_t^P(p) \Rightarrow \mathsf{alloc}(v) \neq (p, l)$ |
| Start overhead | $\mathcal{P}_s^P$ | $\forall v \in V \colon t_f(v, p) \geq t_f(v, p) + \mathcal{P}_s^P$ |

Table 3.4: Constraint derivation rules for a PE $p$ and a task graph $G = (V, E, w, c, t)$

### 3.4.3 Machine Constraints from Configurations

Table 3.5 shows the derivation rules for properties of configurations. The columns are identical to Table 3.4. Each row corresponds to one configuration property from Table 3.2. Applying the rules from the third column with Algorithm 3.1 creates the machine constraints from the configurations' properties. The following provides an explanation of the rules:

- *Collision*: ensure for all instances of configurations in the collision property $\mathcal{P}_c^C(c_0)$ that they do not overlap with instances of $c_0$.

- *Dependency*: ensure for all instances of configurations in the dependency property $\mathcal{P}_d^C(c_0)$ that they do overlap with instances of $c_0$.
- *Placement*: ensure for all tasks allocated on a PE from configuration $c_0$ that it must be instantiated only at locations from the placement property $\mathcal{P}_p^C(c_0)$.
- *Size*: ensure for all locations $L$ that whenever configuration $c_0$ is instantiated at location $l \in L$, it does not overlap with a buffer of $\frac{\mathcal{P}_s^C(c_0)}{\mathcal{P}_b^L(l)}$, respecting the reconfiguration overhead and configuration bandwidth property $\mathcal{P}_b^L(l)$ of location $l$.

For example, suppose the placement property is $\mathcal{P}_p^C(c) = \{l_0, l_3\}$ for a configuration $c$. Then the resulting rule is:

$$\forall v \in V, p \in c\colon \mathsf{alloc}(v) = (p, l) \rightarrow l \in \{l_0, l_3\}$$

| Property | Identifier | Rule |
|---|---|---|
| Collision | $\mathcal{P}_c^C$ | $\forall x \in \mathcal{P}_c^C\colon \neg\,\mathsf{overlap}(x, c_0)$ |
| Dependency | $\mathcal{P}_d^C$ | $\forall x \in \mathcal{P}_d^C\colon \mathsf{overlap}(x, c_0)$ |
| Placement | $\mathcal{P}_p^C$ | $\forall v \in V, p \in c\colon \mathsf{alloc}(v) = (p, l) \Rightarrow l \in \mathcal{P}_p^C(c_0)$ |
| Size | $\mathcal{P}_s^C$ | $\forall l \in L\colon \neg\,\mathsf{overlap}_b\left(x, c_0, \frac{\mathcal{P}_s^C(c_0)}{\mathcal{P}_b^L(l)}\right)$ |

Table 3.5: Constraint derivation rules for a configuration $c_0$ and a task graph $G = (V, E, w, c, t)$

### 3.4.4 Machine Constraints from Locations

Table 3.6 shows the derivation rules for properties of locations. The columns are equal to Table 3.4. Each row corresponds to one location property from Table 3.3. Applying the rules from the third column with Algorithm 3.1 creates the machine constraints from the locations' properties.

- *Bandwidth*: ensure for all pairs of configuration $c_0, c_1$ that they do not overlap with a buffer (the reconfiguration duration) depending on the respective size property $\mathcal{P}_s^C(c_0)$ and $\mathcal{P}_s^C(c_1)$, respectively, if they are both instantiated at location $l$.
- *Reconfiguration overhead*: ensure for all pairs of configurations that they do not overlap with a buffer of the size of the reconfiguration overhead property $\mathcal{P}_r^L$, if they are both instantiated at location $l$.
- *Configurations*: ensure that whenever a configuration $c_0$ is instantiated at location $l$, it must be any of the configurations listed in $\mathcal{P}_c^L(l)$.

| Property | Identifier | Rule |
|---|---|---|
| Bandwidth | $\mathcal{P}_b^L$ | $\forall c_0, c_1 \in C\colon \mathsf{loc}(c_0) = \mathsf{loc}(c_1) = l \Rightarrow \neg\,\mathsf{overlap}_b\left(c_0, c_1, \frac{\max(\mathcal{P}_s^C(c_0), \mathcal{P}_s^C(c_1))}{\mathcal{P}_b^L(l)}\right)$ |
| Reconfiguration overhead | $\mathcal{P}_r^L$ | $\forall c_0, c_1 \in C\colon \mathsf{loc}(c_0) = \mathsf{loc}(c_1) = l \Rightarrow \neg\,\mathsf{overlap}_b\left(c_0, c_1, \mathcal{P}_r^L(l)\right)$ |
| Configurations | $\mathcal{P}_c^L$ | $\forall c \in C\colon \mathsf{loc}(c) = l \Rightarrow c \in \mathcal{P}_c^L(l)$ |

Table 3.6: Constraint derivation rules for a location $l$ and a task graph $G = (V, E, w, c, t)$

### 3.4.5 Communication Constraints from Communication Settings

The last set of constraints are the ones imposed by the communication costs and the topology. The constraints stemming from communication between tasks are not constructed through rules from the machine model. Instead, the *communication setting* defines which constraints must hold for a valid schedule. In the following, we will look into five communication settings, detail their semantics and define the constraints based on those semantics.

Communication can be modeled in numerous ways, depending on the use case and resources at hand. For example, one could decide to ignore the communication entirely to trade accuracy for a faster time to solution. We define five communication settings, each of which leads to a different set of constraints. They can be split up into two categories, with and without configuration:

1. *Without communication*

   Communication costs are not considered.

2. *With communication*

   Communication cost are considered.

   a) *Direct communication*

      Transferring data between tasks introduces communication cost that are dependent on the locality. If the tasks are executed in different locations, the communication cost is non-zero and may delay the dependent's task start time.

   b) *With congestion*

      Transferring data from one task (and PE) occupies links and therefore might disturb other communication and prolong the execution.

   c) *With computation-communication overlap*

      Computation-communication overlap is an optimization technique to achieve better utilization of computing resources.

   d) *With PE-to-PE communication*

      Rather than writing data to memory and reading it later, it is sometimes written directly from one PE to another on FPGAs.

Each of the communication settings defines a set of constraints and includes a definition based on those constraints. If a schedule is valid and adheres to the set of constraints of a communication setting, we say that the schedule is valid *with* a certain setting. For instance, a schedule $\mathcal{S}$ for machine model $\mathcal{M}$ can be valid *with congestion*.

#### 3.4.5.1 Without Communication

The simplest setting ignores communication entirely. If the communication is not considered, the schedule must only consider the precedence of tasks, exclusivity and machine constraints. In this case neither the topology nor the communication cost function $c$ are used. This may be useful if the communication costs are small compared to computational costs, because it can lead reduced algorithmic (scheduling) cost.

**Constraint 3** (Precedence without Communication) *Let $\mathcal{S}$ be a schedule for task graph $G = (V, E, w, c, t)$ and $A = (t_s, \mathsf{alloc})$ a resource allocation. For $v_d \in V$,*

$$\forall v_s \in \mathsf{pred}(v_d) \colon t_s(v_d) \geq t_f(v_s) \tag{3.6}$$

Therefore, constraint 3 merely requires the precedence relation to be honored and the communication costs are ignored.

**Definition 22** (Communication-Free Schedule) *A schedule $\mathcal{S}$ for a task graph $G = (V, E, w, c, t)$ is communication-free if it is feasible and fulfills constraint 3.*

Feasible schedules that fulfill constraint 3 are called "communication-free schedules". However, this communication setting should be used with care, since it potentially ignores a large part of the cost of executing a task graph. This problem is addressed with the following setting.

#### 3.4.5.2 Direct Communication

The communication setting *direct communication* assumes a fixed communication cost for a pair of two tasks and a pair of locations, the precedence constraint must be changed to incorporate the communication cost function $c$.

**Constraint 4** (Precedence with Direct Communication) *Let $\mathcal{S}$ be a schedule for task graph $G = (V, E, w, c, t)$ and $A = (t_s, \mathsf{alloc})$ a resource allocation. For $e_{sd} = (v_s, v_d) \in V$,*

$$\forall (v_s, v_d) = e_{sd} \in E \colon t_s(v_d) \geq t_f(e_{sd}) \tag{3.7}$$

**Definition 23** (Communication-Aware Schedule) *A schedule $\mathcal{S}$ for a task graph $G = (V, E, w, c, t)$ is communication-aware if it is feasible and fulfills constraint 4.*

Feasible schedules that fulfill constraint 4 are called "communication-aware schedules". However, this communication setting ignores that simultaneous access to the same resources is not possible, i.e. it does not handle congestion. The next communication setting explicitly handles congestion using topologies and a dedicated scheduling strategy for communication.

#### 3.4.5.3 With Congestion

Modeling congestion is more involved than applying the first two communication settings. The reason is that each edge in the task graph, i.e. each dependency, cannot be considered in isolation. Instead, the communication of results of a task may have an effect on the scheduling of any other edge.

Defining a topology in the first place has the purpose of modeling the congestion that occurs whenever more data per time unit is sent than the communication channel allows. A simple *endpoint congestion* model can be used already with the properties $\mathcal{P}_b^P, \mathcal{P}_b^C$ and $\mathcal{P}_b^L$. However, this can lead to inaccurate results, i.e. the planned schedule does not reflect the actual execution [27, p. 222]. To offer congestion-aware scheduling algorithms that respect the network topology, we borrow the approach of *edge scheduling* from [125] – that is also described in [27, pp. 203–209] – and integrate it into our approach. Edge scheduling was chosen for three reasons:

1. It allows us to formulate the congestion awareness as a *constraint problem*, too.
2. It integrates with *machine models* and other techniques used in this work.
3. It integrates with our *polynomial time scheduling algorithm* in Chapter 4.

In this subsection, we resort to some of the definitions from Sinnen's work [27] again. However, they are simplified and adjusted to our use case and thus marked with a dashed box as introduced in Section 2.5.

Generally, congestion-aware scheduling relies on three aspects: topology, a *communication model* and scheduling.

1. The topology itself must be defined to accurately model communication congestion.
2. The communication model defines four parameters of the communication.
3. A strategy to schedule communication and its integration into task scheduling

Topologies in general and default topologies in particular have been introduced in Section 3.2.2. To apply congestion-aware scheduling, what remains is to define the communication model and *edge scheduling* as a communication scheduling strategy.

**Communication Model** The communication model defines parameters, such as the routing and switching methods. For brevity and simplicity we assume the following communication model:

- *Static* rather than dynamic routing: routes between two communication endpoints do not change over time.
- *Circuit switching* rather than packet switching: a communication channel is established between two communication endpoints. Note that the bandwidth is *not* reserved for a switched circuit: just the route is static for that communication.
- *Cut-through* rather than store-and-forward: all links on a route are used simultaneously and nodes do not store any messages.
- *No routing delay* rather than some delay per hop: no entity introduces a delay in transmitting data.

This is similar to Sinnen's definition of a communication model in Sinnen 7.5 [27, p. 202]. For a more in-depth explanation of the parameters see Section 7.2.2 [27, pp. 198 ff].

The third component to enable congestion-aware scheduling is the scheduling strategy for communication. We use well-established *edge scheduling* to model the allocation and occupation of resources due to communication that leads to congestion.

**Edge Scheduling** As explained in Section 2.5, an edge from the task graph represents a communication (dependency) between two tasks. The communication is either local (the same location and PE) or it – the edge – must occupy some part of the network topology for some duration. We assume the link is occupied and reserved for that edge exclusively.

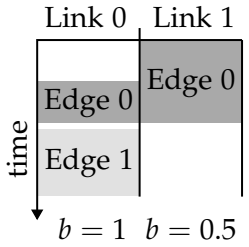



Figure 3.10: An edge schedule of two edges, Edge 0 and Edge 1, on two links, Link 0 and Link 1, each with different relative bandwidth $b$

We chose a strategy to schedule communication that is akin to task scheduling: similar to the mapping of tasks to PEs, edge scheduling is concerned with the mapping of communication (edges) onto links in a topology. Figure 3.10 shows an edge schedule. The x-axis shows two links, Link 0 and Link 1 and the y-axis time flowing from top to bottom. The gray rectangles depict the allocation of communication edges from a task graph on the links. For example, Edge 0 first occupies both Link 1 and Link 0 and later Edge 1 is allocated on Link 0 only. The relative

bandwidth $b$ is denoted on the bottom of each link. Link 1 has half of the bandwidth of Link 0, which means that edges occupy Link 1 twice as long as Link 0.

Tasks executed on two different PEs that have a dependency must communicate using the topology. A path the communication takes in the topology is called a *route*, consisting of a list of links. For example, the route Edge 0 takes in Figure 3.10 goes along Link 0 and Link 1.

**Definition 24** (Route) *Let $\mathcal{T} = (N, O, D, b)$ be a topology. For any two distinct processors $P_s$ and $P_d$, $P_s, P_d \in P$, the routing algorithm of $T$ returns a route $R \in T$ from $P_s$ to $P_d$ in the form of an ordered list of links $R = \langle d_1, \ldots, d_n \rangle$, $d_i \in D$ for $i = 1, \ldots, l$.*

Sinnen 7.8

Integrating task scheduling with edge scheduling is a matter of re-defining the start and finish times of tasks for this communication setting. Instead of a static communication time directly taken from the communication cost function $c$, the start times of tasks have to take the congestion into account. To properly define the start time of a task with congestion, we first need to have a notion of the finish time of an *edge*:

**Definition 25** (Edge Start, Communication and Finish Time on Link) *Let $G = (V, E, w, c, t)$ be a task graph and $\mathcal{T} = (N, O, D, b)$ be a topology.*
*The start time $t_s(e, d)$ of an edge $e \in E$ on a link $d \in D$ is the function $t_s \colon E \times D \to \mathbb{Q}_0^+$.*
*The communication time of $e$ on $d$ is:*

$$\zeta(e, d) = \frac{c(e)}{b(d)}. \tag{3.8}$$

*The finish time of $e$ on $d$ is:*

$$t_f(e, d) = t_s(e, d) + \zeta(e, d) \tag{3.9}$$

Sinnen 7.6

The communication time of an edge on a certain link, i.e. the duration for that a link is occupied by an edge, depends on the link's relative bandwidth. Only when an edge finished its communication, the successor task may start its execution. The point in time when all incoming edges have been communicated to a task is called the *data ready time*.

**Definition 26** (Data Ready Time) *Let $\mathcal{S}$ be a schedule for task graph $G = (V, E, w, c, t)$ on machine model $\mathcal{M} = (\mathcal{A}, \mathcal{T}, \mathcal{P})$. The data ready time of a node $v_j \in V$ on PE $p \in P$ is*

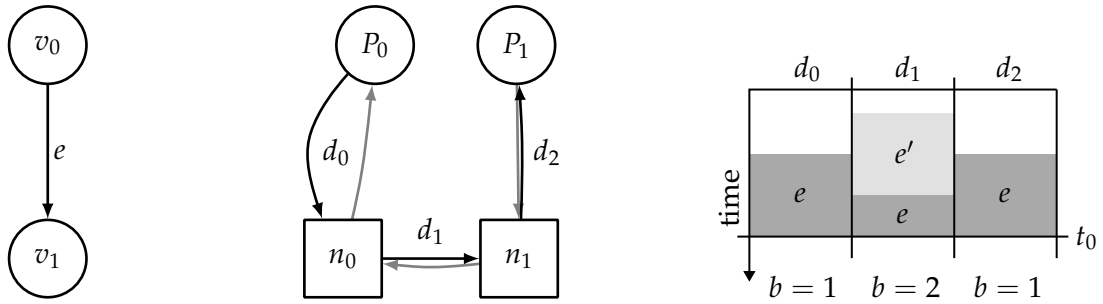
$$t_d(v_j, \mathsf{loc}(p)) = \max_{v_i \in \mathsf{pred}(v_j)} t_f(e_{ij}, \mathsf{loc}(v_i), \mathsf{loc}(p)) \tag{3.10}$$

Sinnen 4.8

Figure 3.11 shows three sub-figures as an example of an edge schedule consisting of a task graph, a topology and the edge schedule itself.

Figure 3.11 (a) on the left shows a task graph G = (V, E, w, c, t) with two tasks $v_0$ and $v_1$. Task $v_1$ depends on task $v_1$, so edge $e = (v_0, v_1) \in E$.

Figure 3.11 (b) in the middle shows a topology connecting two PEs $P_0$ and $P_1$. The route $R = \langle d_0, d_1, d_2 \rangle$ from PE $P_0$ over the switches $n_0$ and $n_1$ to PE $P_1$ is highlighted.




(a) A task graph G = (V, E, w, c, t) consisting of two tasks $v_0$ and $v_1$

(b) Excerpt of topology $\mathcal{T}$ with two PEs $P_0$ and $P_1$ and two switches $n_0$ and $n_1$

(c) A schedule of two edges $e$ and $e'$ on three links $d_0$, $d_1$ and $d_2$ with bandwidth $b = 1$ and $b = 2$, respectively [125]

Figure 3.11: A topology and a corresponding edge scheduling for two edges

Figure 3.11 (c) on the right shows the edge schedule of task graph $G$ on topology $\mathcal{T}$, similar to Figure 3.10. The three links $d_0, d_1$ and $d_2$ along route $R$ from the topology $\mathcal{T}$ are depicted on the x-axis and the time from top to bottom on the y-axis. In the edge schedule the edge $e$ is scheduled to communicate over the route $R$. Each of the three links are occupied by edge $e$ just before the time $t_0$. Note that link $d_1$ has twice the bandwidth $b$ of links $d_0$ and $d_2$, so it is occupied half of the duration in comparison to the other two links.

Another edge $e'$ – not part of task graph $G$ – occupies link $d_1$ for some time to show the effect of congestion. The edge $e'$ is scheduled on $d_1$ and occupies it before $e$ is scheduled. If $e'$ occupies $d_1$ longer, the finish time of $e$ (that is $t_f(e, d_2)$) would move past $t_0$ and thus finish later. One possibly counterintuitive detail about the schedule is that the finish time of $e$ is the same on all three links, namely $t_0$. Further, it seems as if $t_s(e, d_2) < t_s(e, d_1)$, i.e. the communication starts on a subsequent link $d_2$ before it starts on the preceding link $d_1$. Recall that edge scheduling models congestion. Even if we remove edge $e'$ from the picture and only consider $e$, $t_s(e, d_1) > t_s(e, d_2)$ and $t_f(e, d_1) = t_f(e, d_2)$ must hold. If this was not the case, for example $t_s(e, d_0) = t_s(e, d_1) = t_s(e, d_2)$ then edge $e$ would finish on link $d_1$ before it is finished on link $d_0$. Further, if $t_s(e, d_1) = t_s(e, d_2)$, then a *faster link* $d_1$ *would prolong* $t_f(e, d_2)$ [125].

**Constraints for Schedules with Congestion** Although the communication setting with congestion is more involved than the previous two settings, it is expressed by two constraints to form a valid schedule:

1. The *Link constraint* ensures that an edge occupies a link exclusively.
2. The *Causality constraint* ensures that the finish time of an edge is consistent along the route.

The link constraint, similar to constraint 1 for PEs, guarantees that a link is only occupied by one edge at a time.

**Constraint 5** (Link) *Let $G = (V, E, w, c, t)$ be a task graph and $\mathcal{T} = (N, O, D, b)$ be a topology. For any two edges $e, f \in E$ scheduled on link $d \in D$:*

$$t_s(e,d) \leq t_f(e,d) \leq t_s(f,d) \leq t_f(f,d) \vee t_s(f,d) \leq t_f(f,d) \leq t_s(e,d) \leq t_f(e,d), \tag{3.11}$$

*or equivalently*

$$\neg\, \mathsf{overlap}(e,f,d). \tag{3.12}$$

Sinnen 7.1

Analogous to constraint 4, the precedence constraint for tasks, the communication of an edge on a link later on the router must not start before the allocation on first link. Similarly, the communication on the first link must not start before the predecessor's node finish time and the successors start time must be larger than the finish time of the edge on the last link of the route.

**Constraint 6** (Causality) *Let $G = (V, E, w, c, t)$ be a task graph and $\mathcal{T} = (N, O, D, b)$ be a topology. For the start and finish times of edge $e = (v_s, v_d) \in E$: on the links of the route $R = \langle d_0, d_1, \ldots, d_l \rangle$, $R \in T$,*

$$t_s(e,d_1) \geq t_f(v_s), \tag{3.13}$$
$$t_f(e,d_{k-1}) \leq t_f(e,d_k), \tag{3.14}$$
$$t_s(e,d_0) \leq t_s(e,d_k), \tag{3.15}$$
$$t_s(v_d) \geq t_f(e,d_l), \tag{3.16}$$

*for $0 < k \leq l$.*

Sinnen 7.2

Constraints 5 and 6 must hold for a congestion-aware schedule. The first one guarantees that the communication is overlap-free on each link. In reality, the link *can* use packet switching, but as an approximation to model congestion, we assume circuit switching characteristics. This means that the data is not split into small chunks that are individually sent but as a contiguous data stream from source to destination. The second condition ensures the causality between the communication links. Additionally, it ensures that the communication only starts after the predecessor task has finished executing, and the destination task starts executing only after the communication on the last link of the route finished. Note that Equations (3.13) and (3.16) are not included in Sinnen 7.2, but implicit in their definition for edge scheduling instead.

Since $t_s(e,d)$ and more so $t_f(e,d)$ for any edge $e$ and link $d$ depend on the relative bandwidth, this property can be constructed from the normalized $\mathcal{P}_b^X$. In other words, using the available bandwidth of PEs (and by extension configurations) and locations, the edge weights of the topology graphs can be computed. The constraints for congestion-aware scheduling are fully described by constraints 5 and 6, because the properties are already contained in the topology description.

**Definition 27** (Congestion-aware Schedule) *A schedule $\mathcal{S}$ for a task graph $G = (V, E, w, c, t)$ and a topology $\mathcal{T} = (N, O, D, b)$ is congestion-aware if it is feasible and fulfills constraints 5 and 6.*

Feasible schedules that fulfill constraints 5 and 6 are called "congestion-aware schedules". However, this communication setting ignores two possible optimizations that are commonly found in FPGA programs:

1. Computation-communication overlap
2. Direct PE-to-PE communication

The next communication settings each address one of these.

#### 3.4.5.4 With Computation-Communication Overlap

Up to this point, we assumed that a PE is either scheduled, executing or finished and produced data that can be communicated. Since FPGAs do not implement the conventional Von Neumann architecture, they can communicate and execute at the same time. In fact, it is a design pattern in the HLS workflow to construct a PE such that it works in three phases:

(1) *Consumption* of data: the input data for the PE is read, for example from a global memory, a network interface or another PE (see Section 3.4.5.5). This allows the interface to coalesce transfers and achieve a higher bandwidth through less overhead. The data can be stored in a local buffer.

(2) *Processing* of data: the data is processed and stored into a local buffer.

(3) *Production* of data: the output data is written from the local buffer to global memory, a network interface or another PE.

These three phases can be pipelined: while phase (1) keeps ingesting data, phase (2) may already write output to the local buffer and possibly allow phase (3) to produce output to the target. The implication for scheduling are the following:

- A task $v$ may start its communication before its finish time $t_f$, but its computational cost ($w(v)$) remains unchanged.
- A task $v$ may start executing before all data has arrived, i.e. the $t_d(v)$ conforms with Definition 26.
- A task may execute and simultaneously consume and produce data.
- The start edge time $t_s((v_s, v))$ of the incoming data must come before the start edge time of the outgoing edge $t_s((v, v_d))$ for tasks $v_s, v, v_d$. One could construct a case where a PE produces output data before the arrival of the first input data, but we consider the former case.
- Notably the start time $t_s(v_d)$ may be before the finish time $t_f(v_s)$, where $v_d \in \mathsf{succ}(v_s)$.

**Constraint 7** (Communication Overlap) *Let $G = (V, E, w, c, t)$ be a task graph and $\mathcal{T} = (N, O, D, b)$ be a topology. For the start and finish times of edge $e = (v_s, v_d) \in E$: on the links of the route $R = \langle d_1, d_2, \ldots, d_l \rangle$, $R \in T$,*

$$t_s(e, d_1) \geq t_f(v_s) - \mathcal{P}_o^P(\mathsf{proc}(v_s)), \tag{3.17}$$

$$t_s(e, d_1) \geq t_s(v_s), \tag{3.18}$$

$$t_s(v_d) \geq t_f(e, d_l) - \mathcal{P}_o^P(v_d), \tag{3.19}$$

*for $1 < k \leq l$.*

The communication overlap constraint allows the earlier execution time. Note that, this will not result in a feasible schedule as defined by Definition 20, because $t_s(v_d) < t_f(v_s)$ may be true, violating constraint 4. However, the definition relies on the edge finish time as defined for location-based communication cost (Definition 10), which does not make much sense for a topology-based communication setting. Thus, Definition 10 and by extension constraint 4 are ignored in this setting.

**Definition 28** (Overlap-aware Schedule) *A schedule $\mathcal{S}$ for a task graph $G = (V, E, w, c, t)$ and a topology $\mathcal{T} = (N, O, D, b)$ is computation-communication overlap aware, if it is feasible and fulfills constraint 7*

Feasible schedules that fulfill constraint 7 are called "overlap-aware schedules". Another optimization is having direct PE-to-PE communication.

#### 3.4.5.5 With PE-to-PE Communication

A common theme in implementations of software on FPGA-based accelerators is the usage of direct communication channels. With a producer-consumer model, where a PE *consumes* data from a *producer* PE, data is directly send from one PE to another. A PE may also be both, a consumer and producer. For FPGA-based accelerators it is a popular approach, because the data does not need to be saved to the global memory, but can use the direct on-chip interconnect and local buffers.

The channel communication mode is supported by relevant platforms, for example by Xilinx's HLS implementation Vitis [25] called *streams* and by Intel's oneAPI [126] named *pipes*. It was adopted as pipes into the OpenCL specification since version 2.0 [127], that are semantically equivalent to buffered or unbuffered FIFO queues. OpenCL's pipes are used – similar to Unix pipes – to directly connect PEs with no or only minor buffering. Storing intermediate results is thus not necessary. Listing 3.1 shows OpenCL code with one kernel function that takes two arguments, an `in_data` read pipe and an `out_data` write-only pipe. In lines 2–3 it reads data from `in_data` and saves it into a *private* buffer `buf`. In lines 5–6 the function `compute_val` is applied to all elements in the buffer `buf` and the remaining code writes the result to `out_data` which in turn can be read by another PE. This structure also corresponds to the three phases as presented in Section 3.4.5.4 and thus may be pipelined, potentially also enabling computation-communication overlap.

```
1 __kernel void consumer_producer(pipe int in_data, write_only pipe
    ↪ int out_data) {
2   int buf[100];
3   for (int i = 0; i < 100; i++)
4     read_pipe(in_data, &buf[i]);
5   for (int i = 0; i < 100; i++)
6     buf[i] = compute_val(buf[i]);
7   for (int i = 0; i < 100; i++)
8     while(write_pipe(out_data, &buf[i]));
9 }
```

Listing 3.1: OpenCL kernel that uses three phases: reading, computing and writing data via pipes

Optionally, this communication setting can be combined with the congestion-aware setting. However, all participating PEs must be present at the same time which in turn has an influence on the instantiation of configurations. Contrary to the communication settings above, no dedicated

constraints have to be introduced for this setting. Instead, the dependencies of the PEs must be modeled with the topology and the dependency property $\mathcal{P}_d^P$. If two PEs $p_s$ and $p_d$ use PE-to-PE communication, e.g. via pipes, the following must hold:

- The topology $\mathcal{T} = (N, O, D, b)$ must contain a link $d = (p_s, p_d) \in D$, possibly with $b(d) = \infty$.
- The destination PE $p_d$ must have a dependency on the source $p_s$ or vice-versa: $\{p_d\} \subseteq \mathcal{P}_d^P(p_s)$.

A schedule that follows this communication mode is called pipe-aware.

**Definition 29** (Pipe-aware Schedule) *A schedule $\mathcal{S}$ for a task graph $G = (V, E, w, c, t)$ and a machine model $\mathcal{M} = (\mathcal{A}, \mathcal{T}, \mathcal{P})$ is pipe-aware if:*

- *it is feasible.*
- *there exists at least one direct PE-to-PE pipe: $\exists (o_s, o_d) \in D \colon o_s \neq o_s \wedge \{o_s, o_d\} \subseteq O$.*
- *those PEs are dependent: $\forall (o_s, o_d) \in D \colon o_s \in \mathcal{P}_d^P(o_d) \vee o_d \in \mathcal{P}_d^P(o_s)$.*

### 3.4.6 Summary of the Schedule Constraints

Summarizing, constraints are a central component of this work. They are being used to limit possible schedules based on the properties of the modeled hardware. We introduced three types of constraints: general constraints, machine constraints and communication constraints.

Table 3.7 shows the communication settings and the constraints that must be fulfilled by a schedule to be considered valid with that setting. The communication setting names are listed in the first column and the remaining 7 columns represent one of the constraints each. Each row shows a communication setting and the constraints that must hold for a schedule that is valid with the row's setting are marked with ✓. Optional constraints are marked with (✓). For example, a pipe-aware schedule must fulfill the two general constraints 1 and 2, the (communication) constraint 7 and may optionally fulfill constraints 5 and 6.

| | **Constraint** | | | | | | |
|---|---|---|---|---|---|---|---|
| **Setting** | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Without communication | ✓ | ✓ | ✓ | | | | |
| Direct communication | ✓ | ✓ | ✓ | ✓ | | | |
| Congestion-aware | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | |
| Overlap-aware | ✓ | ✓ | | | (✓) | (✓) | ✓ |
| Pipe-aware | ✓ | ✓ | | | (✓) | (✓) | ✓ |

Table 3.7: The constraints that must hold for the communication settings

To reiterate Chapter 3 so far, we introduced accelerator models and communication topologies and combined them to machine models. We're able to describe their properties and derive constraints on schedules using the structure from machine model and these very properties.

The next step in the methodology, as described in Section 3.1, addresses the problem of working with existing applications. With the previously described approach it is possible to decide whether a schedule is valid or not on a particular machine model. Still missing are realistic workloads to analyze. The next section focuses on gathering workloads from existing applications, possibly already accelerated by an FPGA-based accelerator.

## 3.5 Generating Task Graphs

Nontrivial analysis of scheduling approaches requires representative input workloads. Especially statistical analysis relies on input data for simulations. Such workloads can be obtained:

- as *statistical models* [128],
- through *static code analysis*,
- through *profiling* or, as in our case,
- through *tracing applications* to construct task graphs.

Task graphs and particularly their cost function $c\colon V \times P \to \mathbb{N}$ are not always available, especially for FPGA based accelerators. Hence, we propose using tracing to generate task graphs from program executions. With tracing, a task based application is deployed on an FPGA and its execution is logged from the host.

The advantages of tracing over workload modeling or static analysis are the following.

- Information that is only available at runtime can be obtained.
- The logging of measured execution data is agnostic to the FPGA implementation.
- Observing the execution from the host introduces no overhead on the accelerator.
- The application under investigation does not need to be instrumented.
- Traces provide an exact sequence of the program execution, rather than a statistical model.

The disadvantages of tracing over workload modeling or static analysis are the following.

- An application must be executed – or simulated – at least once on a target architecture.
- Varying parameters, e.g. input sizes or precision, require repeated execution of the application. The degrees of freedom and number of the parameters may inflate the exploration space prohibitively and make it impracticable to generate all traces.
- Tracing must be implemented at runtime.

Tracing an application is the process of recording events during the execution of a program. At certain points during the execution of a program, an often small set of information is collected, so the timeline of the execution can be reconstructed later. For our purposes a trace must collect a tuple $(i, I, p, o_s, o_f, d)$ during the execution of a task, where:

- $i \in \mathbb{N}$ is a unique task identifier
- $I \subset \mathbb{N}$ is the set of direct predecessor task identifiers
- $p \in P$, where P is the set of available PEs and $p$ is the assigned PE
- $o_s \in \mathbb{N}$ is the start time offset of the task $i$
- $o_f \in \mathbb{N}$ is the finish time offset of the task $i$
- $d \in \mathbb{N}$ is the amount of data transferred during the task's execution

After the data is gathered, a task graph can be generated with Algorithm 3.2. The tuples obtained from the execution of the program are used as an input. If the program generates tasks deterministically and stable identifiers are assigned to the tasks, the algorithm can be invoked with tracing data of multiple program executions. The algorithm iterates over all trace tuples and inserts tasks into the task graph based on this information. It also creates dependencies based on the predecessor tasks $I$ and sets the computation and communication costs in lines 2 – 8. In line 10 it handles the case of multiple traces of the same application. The motivation for tracing

**Input:** Set $T$ of trace tuples; binary reduction functions $\odot_{\{\omega,w\}}$
**Result:** Task graph $G = (V, E, w, c, t)$
**foreach** $(i, I, p, o_s, o_f, d) \in T$ **do**
  **if** $i \notin V$ **then** insert task into task graph
    $V \leftarrow V \cap \{i\}$
    $w(i, p) \leftarrow o_f - o_s$
    **foreach** $i_p \in I$ **do**
      $E \leftarrow E \cap \{(i_p, i)\}$
      $c((i_p, i)) \leftarrow d$
    **end**
  **else** update task cost
    $w(v) \leftarrow \min(w(v), \omega(i, p))$
  **end**
  **return** $(V, E, w, c, t)$
**end**

Algorithm 3.2: Task graph generation from trace data

multiple application runs and combining their traces into a single task graph is an increase in accuracy. Algorithm 3.2 uses a min function to select the minimal runtime cost of a task over all runs.

This algorithm is implemented as a module of the RESCH framework as described in Section 6.1.2.

With a description of the hardware (a machine model) and appropriate workloads (task graphs gathered through tracing) the next step according to our methodology from Section 3.1 is the analysis of schedules.

## 3.6 Analyzing Schedules

Recall that SRQ 1 asks how reconfiguration-aware schedules can be systematically evaluated. We have now laid the groundwork for the proper analysis of task scheduling algorithms on reconfigurable hardware. With this set of tools, we enable statistical analysis of schedules and also, by extension, the analysis of machine models.

After generating schedules for a range of input task graphs, for example using constraint programming, the result can be systematically analyzed. The approach allows us to build one-time abstractions of the hardware and then analyze schedules for a wide range of workloads and parameters. A possible alternative to this are formal proofs. However, while these provide universally valid statements, proofs must be rather conservative in its assumptions, whereas statistical analysis allows a direct relationship to real-world problems and their solutions. This method allows us to quickly iterate the solution space and assess the quality of a schedule in Chapter 5.

Our approach to analyze the machine model, scheduling algorithms and their interactions is to generate valid – and possibly optimal – schedules and then analyze the outcome with statistical methods. The process requires three steps:

1. Choose a *target metric*.
2. Generate *feasible and optimal schedules* according to the criterion.

3. Accumulate the *result metric(s)*. This can be the same that we used to optimize or a different set, for example the achieved parallelism, energy usage or the speedup.

For example, using a set of task graphs generated with the previous step, it is now possible to generate optimal schedules according to the target metric. The mean of the result metric can now be used to assess the quality of an approach.

The following section demonstrates our approach on the basis of a simple example in a case study. In particular, it shows how to isolate and examine the effect of PR.

## 3.7 Case Study: Two Machine Models

With respect to SRQ 1, we want to compare two machine models and demonstrate the process described in this chapter. Recall that we are particularly interested in the effect of partial reconfiguration on scheduling tasks and the costs of avoiding task-level PR all together in favor of an easier development process. As an example comparison, we present two similar artificial machine models $\mathcal{M}_{\text{PR}}$ and $\mathcal{M}_{\text{R}}$, with one difference: the first supports PR while the latter does not.

This case study follows the methodology from Section 3.1 and is structured as follows:

1. The machine models are defined in Section 3.7.1.
2. The constraints are derived from the models in Section 3.7.2.
3. The task graph is generated from (fictional) trace data in Section 3.7.3.
4. The analysis is applied to the task graph in Section 3.7.4.

We prefer brevity and simplicity over realism in this example to make the case study more accessible, as it should serve as an illustration of the application of our methodology. More complex scenarios are evaluated in Chapter 6.

### 3.7.1 Definition of the Machine Models

In this scenario, we have three PEs in total $\{p_0, p_1, p_2\} = P$. Each PE can execute two tasks of the task graph depicted in Figure 3.12. Each node represents a task, with the task label (a name) on the left and its cost on the right. The edges are annotated with the communication cost. For the sake of simplicity, all tasks have an execution time of 100 ms on all PEs, the reconfiguration overhead is 10 ms for all PEs and properties like the bandwidth are ignored. The two machine models $\mathcal{M}_{\text{PR}}$ and $\mathcal{M}_{\text{R}}$ can be described as the following:

1. $\mathcal{M}_{\text{PR}} = (\mathcal{A}_{\text{PR}}, \mathcal{T}, \mathcal{P}_{\text{PR}})$ describes an FPGA with two PR-slots, i.e. two PEs can execute simultaneously.
2. $\mathcal{M}_{\text{R}} = (\mathcal{A}_{\text{R}}, \mathcal{T}, \mathcal{P}_{\text{R}})$ describes an FPGA without PR capability and two bitstreams:
    1. A bitstream with two PEs $\{p_0, p_1\}$.
    2. A bitstream with one PE $\{p_2\}$.

Both machine models $\mathcal{M}_{\text{PR}}$ and $\mathcal{M}_{\text{R}}$ contain the same set of PEs $P$. Disregarding PR specifics for now, we define properties that are true for both machine models.

- All $v \in V$ have $w(v, p) = 100\,\text{ms}$ for all $p \in P$.
- All $l \in L$ have $\mathcal{P}_r^L(l) = 10\,\text{ms}$.

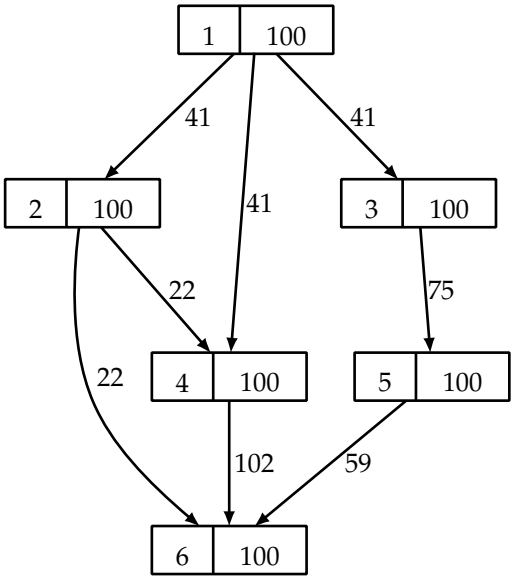



Figure 3.12: Case study task graph $G_c$: the task label is denoted on the left within each node and its cost in ms on the right. The labels on edges denote communication cost in MB.

- Each $p_n \in P$ has $\mathcal{P}_t^P(p_n) = \{v_n, v_{n*2+1}\}$, i.e. $p_1$ can execute tasks $v_2$ and $v_3$.

The PEs of both machine models have model-specific properties. These describe the characteristics specific to PR or the lack thereof, respectively. PR is described by the set of available locations $L$ of the machine model.

- Machine model $\mathcal{M}_{\text{PR}}$ has two slots, so the set of locations $L_{\text{PR}} = \{s_0, s_1\}$ of its accelerator model $\mathcal{A}_{\text{PR}}$ reflects that. To allow all PEs to be configured in both slots, one property has to be added:
  - Both $c \in C$ have $\mathcal{P}_p^C = \{s_0, s_1\}$.
- Machine model $\mathcal{M}_{\text{R}}$ has one slot, so the set of locations $L_{\text{R}} = \{s_0\}$ of its accelerator model $\mathcal{A}_{\text{R}}$ reflects that. The PEs contained in one bitstream are instantiated at the same time, since there is no partial reconfiguration. Equivalently, the PEs not in the same bitstream are exclusive, since only one bitstream can be loaded at a time. These characteristics are already described by the machine model and in particular constraint 2, so no additional properties have to be set.

With these properties, both machine models $\mathcal{M}_{\text{PR}}$ and $\mathcal{M}_{\text{R}}$ are fully defined. The next step is the derivation of constraints. The rules $\mathcal{R}$ from Tables 3.4, 3.5 and 3.6 are applied according to Algorithm 3.1.

### 3.7.2 Derivation of Constraints

The derivation process for constraints is repetitive and can be automated. For illustration, we apply one rule for an exemplary property; all other properties and PEs are processed equivalently. The goal of this step is to determine a set of constraints that must hold for feasible schedules. The constraint set is specific for a machine model. However, since the PEs of machine models

$\mathcal{M}_{\text{PR}}$ and $\mathcal{M}_{\text{R}}$ share all properties, the application of the rules for these properties is the same for both of them.

The property $\mathcal{P}_r^L = 10\,\text{ms}$ is valid for all locations. According to Algorithm 3.1 (line 2) we iterate over all PEs $p \in P$ and for each PE $p$ over all properties $\mathcal{P}^p$. In this example we only consider $\mathcal{P}_r^L$ and $p_0$. In line 4 $\text{const}_{\mathcal{R}}(\mathcal{P}_r)$ maps the property $\mathcal{P}_r$ to a rule and inserts it into the constraint set $C$:

$$C \leftarrow \emptyset \cup \text{const}_{\mathcal{R}}(\mathcal{P}_r^L) = \{\forall l \in L\colon t_f(v,l) \geq t_r(v,l) + 10\,\text{ms}\}.$$

After the application of Algorithm 3.1, $C$ contains all constraints to limit feasible schedules for $\mathcal{M}_{\text{PR}}$ and $\mathcal{M}_{\text{R}}$ respectively.

Information that is typically only available at runtime is the execution cost. In this case study all locations have a consistent configuration cost of 10 ms. This information is gathered in a separate step, where the task graph along with the computation and communication cost is generated.

### 3.7.3 Task Graph Generation

To generate the task graph from a program execution, we collect the tuple $(i, I, p, o_s, o_f, d)$ of tracing data per executed task. Table 3.8 shows the tuples for a trace of a fictional example application. Each column corresponds with an element from the tuple. Each row corresponds to one tracing record, i.e. one event during the execution of the fictional case study application. For example, the task with identifier 3 has two predecessors 1 and 2, was executed on PE $p_1$ during the interval from 10,160 ms to 10,260 ms and transmitted 75 MB of data.

Table 3.8: Trace data for a fictional example task application

| $i$ | $I$ | $p$ | $o_s$ | $o_f$ | $d$ |
|---|---|---|---|---|---|
| 1 | $\emptyset$ | $p_0$ | 10,042 ms | 10,142 ms | 41 MB |
| 2 | $\{1\}$ | $p_0$ | 10,188 ms | 10,288 ms | 22 MB |
| 3 | $\{2\}$ | $p_1$ | 10,160 ms | 10,260 ms | 75 MB |
| 4 | $\{1,2\}$ | $p_1$ | 10,341 ms | 10,441 ms | 102 MB |
| 5 | $\{3\}$ | $p_2$ | 10,316 ms | 10,416 ms | 59 MB |
| 6 | $\{2,4,5\}$ | $p_2$ | 10,580 ms | 10,680 ms | 0 |

With the application of Algorithm 3.2, the task graph in Figure 3.12 is generated. Although a (feasible by definition) schedule was observed during the execution, we cannot make any assumptions about the quality of the schedule. Especially, the lower bound of the makespan is yet unknown.

To find a lower bound and analyze the application systematically, we generate an optimal schedule and analyze it below.

### 3.7.4 Analyzing Schedules

To generate an optimal schedule, the constraint set constructed in Section 3.7.2 is transformed for a constraint programming language or toolkit. Constraint programming languages or toolkits take a set of variables and constraints on the variables as an input. They produce a set of assignments to the variables that fulfill the constraints. Additionally, they may also minimize some cost

function (while maintaining the fulfillment of constraints). This is the mode of operation that is used below.

The software used for the case study is OR-Tools [129]. However, the procedure is similar for other constraint programming languages and toolkits:

1. Define PEs.
2. Define tasks, dependencies and cost as part of the task graph.
3. Define the optimization target. In this case it is the makespan of a schedule.
4. Translate constraints from the previous steps to constraints on the schedule.

The first three steps are the same regardless of the machine model but only dependent on the application under investigation. Listing 3.2 shows code for the first two steps using the Python programming language: in line 1, the tasks with labels and cost are defined. The dependencies – corresponding to the task graph from Figure 3.12 – are defined in line 2 as a simple two-dimensional array, i.e. `dependencies[2]` returns the dependencies of task 2 as a list of task identifiers.

```
1 tasks = [[1, 100], [2, 100], [3, 100], [4, 100], [5, 100], [6, 100]]
2 dependencies = [[], [1], [1], [1,2], [3], [4,5]]
```

Listing 3.2: Definition of the task graph and its dependencies.

The information specific for a machine model is defined in Listing 3.3 in terms of PEs, configurations and locations (`pe_to_config`, `config_to_location`). Additionally, the task types (`task_to_pes`), i.e. $\mathcal{P}_{f,t}$, are defined for each task. The listing shows the definitions for machine model $\mathcal{M}_\mathrm{R}$: `pe_to_config` specified that $p_0$ and $p_1$ are contained in the same configuration (0) and $p_2$ is contained in another configuration (1). There is only one location 0, see `config_to_locations`). The tasks are assigned to PEs as described in Section 3.7.1, i.e. task 1 and 2 are executable on $p_0$ and so on.

```
1 pe_to_config = [0, 0, 1]
2 config_to_locations = [[0], [0], [0]]
3 tasks_to_pes = [[0], [0], [1], [1], [2], [2]]
```

Listing 3.3: Definition of the configurations, locations and task types.

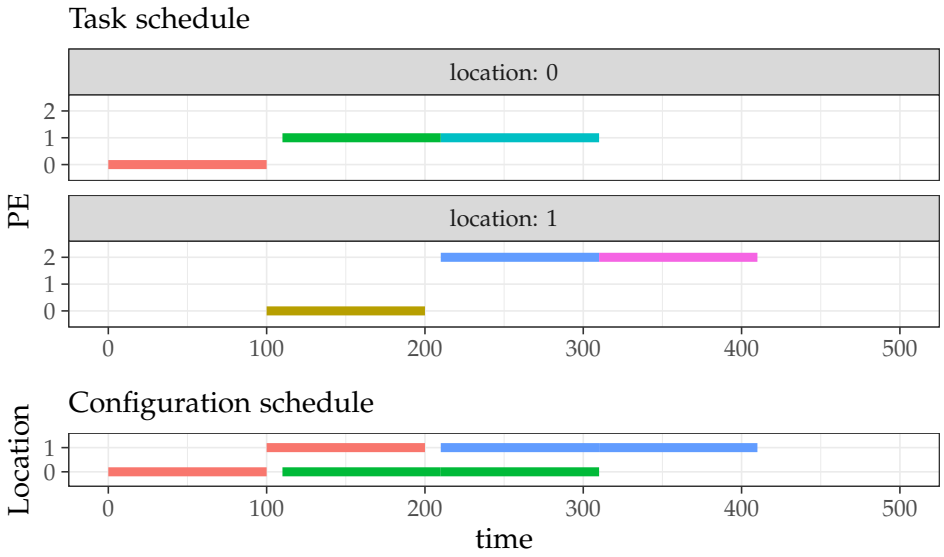
These definitions are the input to a program[1] using OR-Tools to identify the schedule. The program encodes the rules from Tables 3.4, 3.5 and 3.6 and applies them to the input task graph and machine model.

The output is an optimal schedule and a list of non-optimal but feasible schedules, both of which can be transformed into a graphic representation for easy inspection.

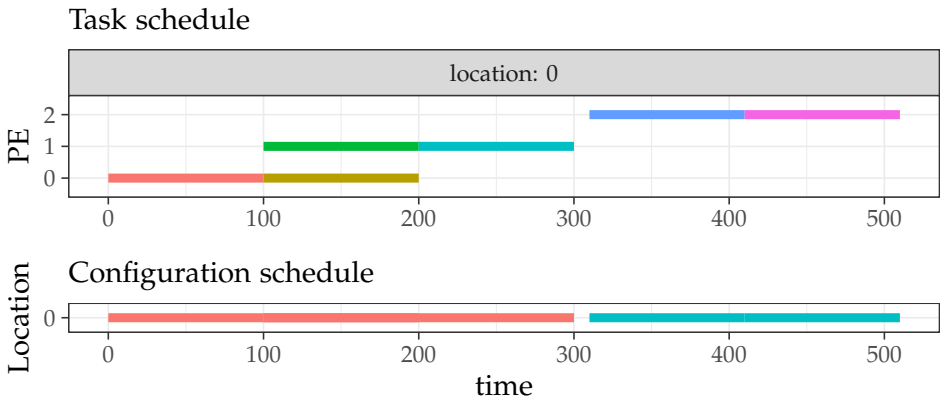
Figure 3.13 shows the optimal schedules $\mathcal{S}_\mathrm{PR}$ and $\mathcal{S}_\mathrm{R}$, respectively, for both machine model $\mathcal{M}_\mathrm{PR}$ (Figure 3.13 (a)) and machine model $\mathcal{M}_\mathrm{R}$ (Figure 3.13 (b)) for task graph $G_c$. Both figures depict the timing of the schedule, the locations, the assignment of tasks to PEs and the assignment of configurations to locations. On the top of both diagrams in Figure 3.13, the task schedule is shown with each of the six tasks scheduled to the PEs. The *configuration schedule* below shows the instantiations for each location. The color denotes the task (index) in the task schedule (upper) part and the active configuration in the configuration schedule (lower) part.

Notable is the difference in the schedule length: in this example, the schedule for machine model $\mathcal{M}_\mathrm{PR}$ reconfigures location $s_1$ (the second slot) with configuration $c_2$ and (after a reconfiguration

[1]The code of this example is available at https://github.com/pascalj/resch




(a) Schedule $\mathcal{S}_{\text{PR}}$ for $\mathcal{M}_{\text{PR}}$




(b) Schedule $\mathcal{S}_{\text{R}}$ for $\mathcal{M}_{\text{R}}$

Figure 3.13: Optimal schedules generated for $G_c$ (from Figure 3.12)

overhead of 10ms) starts executing task 5 on PE $p_2$. On $\mathcal{M}_{\text{R}}$, the execution of task 4 must finish before the single location $s_0$ can be configured with configuration $c_1$ that contains PE $p_2$. Thus, the schedule length of $\mathcal{S}_{\text{PR}}$ is $\mathsf{sl}(\mathcal{S}_{\text{PR}}) = 410$ while the schedule length of $Sc_{\text{R}}$ is $\mathsf{sl}(\mathcal{S}_{\text{R}}) = 510$.

With the constraint programming technique, it is possible to generate feasible schedules as well as the optimal one, since the constraint solver can iterate all possible schedules for the given inputs. We can trivially provide an upper bound on the schedule, so the search space is finite. There are several issues that can be addressed with further analysis:

- The optimal schedule is only valid for one task graph and one machine model.
- Small changes to the task graph, e.g. its cost, can affect the schedule significantly. Therefore, the impact of noise in the program execution, inaccuracies and properties that are not correctly modeled is not quantifiable using solely one optimal schedule.
- Statements about the machine model in general cannot be made.
- Any optimizations we apply to either the machine model or the task graph are affected by the same limitations as listed above.

With deductive analysis, we can further investigate any potential for optimization and assess the quality of the machine model(s) at hand.

#### 3.7.4.1 Statistical Analysis

For a statistical analysis a number of task graphs are required. In this case study, only one task graph ($G_c$, Figure 3.13) is available. To get a better picture of the influence of the machine model on the schedule, we generate similar task graphs by adding noise to the task and communication costs.

We generate 100 versions of task graph $G_c = (V, E, w, c, t)$ by adding a random sample from a normal distribution with $\mu = w(v)$ and $\sigma = 0.1 \cdot w(v)$, i.e. the *imbalance* is 10%. Optimal schedules for all 100 task graphs are generated for both machine model $\mathcal{M}_{\text{PR}}$ and machine model $\mathcal{M}_{\text{R}}$ using the constraint programming solution from the previous section.

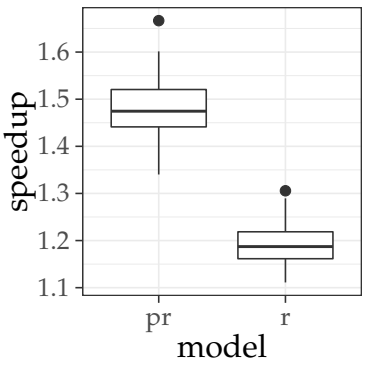


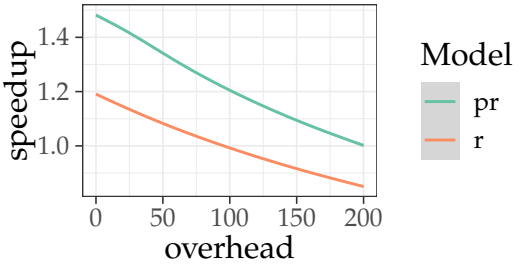

Figure 3.14: Speedup of 100 variations of $G_c$ for two machine models $\mathcal{M}_{\text{PR}}$ and $\mathcal{M}_{\text{R}}$

With this data, more information can be gained about the two machine models. Figure 3.14 shows the speedup for the 100 executions of both models as a box plot. It shows that machine model $\mathcal{M}_{\text{PR}}$ can achieve a higher speedup and more parallelism with partial reconfiguration. It achieves an average speedup of 1.46 while machine model $\mathcal{M}_{\text{R}}$ has an average speedup of 1.17. While it is intuitively clear that a system that supports PR will lead to shorter schedules (everything else being equal), we can now quantify the effect and argue about whether the increased development time, resource usage and complexity is worth the effort.

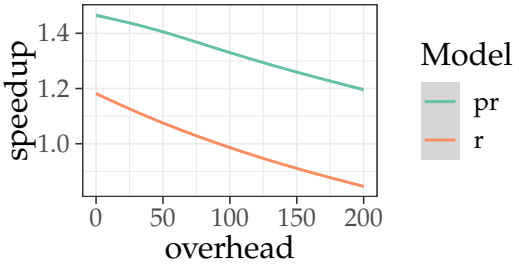
Further, it is now possible to examine changes to the system. The following questions are short examples for issues that might come up during the analysis of a system. A user of our approach can adapt the machine model or the communication setting accordingly and simulate scheduling for further insights.

- What is the influence of reconfiguration delay $\mathcal{P}_r^L$ to the speedup? What if its time changes relative to the average task execution time? Figure 3.15 (a) depicts the mean schedule length relative to the configuration overhead for both machine model $\mathcal{M}_{\text{PR}}$ and machine model $\mathcal{M}_{\text{R}}$. It is apparent that as the overhead increases to 200% of the average task size, the difference between the PR and R model decreases.
- What if the reconfiguration overhead was more realistically modeled relative to the changed area? In contrast, Figure 3.15 (b) shows the behavior under this constraint, everything else being equal. Clearly machine model $\mathcal{M}_{\text{PR}}$ performs better than in the former case, even widening the gap to the performance of machine model $\mathcal{M}_{\text{R}}$.

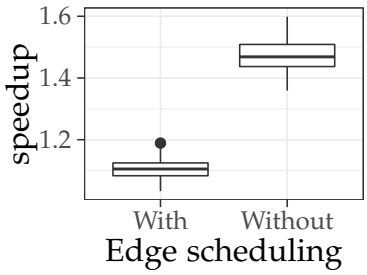
- What if we consider edge scheduling and congestion? The difference is shown in Figure 3.15 (c). For both models it shows the speedup with and without the usage of edge scheduling. Clearly, when considering congestion-aware scheduling in this setting, the speedup is lower. This can be an indication that PEs might be combined, equipped with more local memory or make use of direct PE-to-PE configuration.




(a) Speedup of optimal schedules of all graphs in $G_c$ for two machine models $\mathcal{M}_{\text{PR}}$ and $\mathcal{M}_{\text{R}}$




(b) Speedup of optimal schedules of all graphs in $G_c$ for two machine models $\mathcal{M}_{\text{PR}}$ and $\mathcal{M}_{\text{R}}$ for a range of reconfiguration overheads.




(c) Speedup of optimal schedules of all graphs in $G_c$ for two machine models $\mathcal{M}_{\text{PR}}$ and $\mathcal{M}_{\text{R}}$ for a range of reconfiguration overheads.

Figure 3.15: Analysis of a set of 100 graphs structured as $G_c$ with the cost normally distributed around 100 with a standard deviation of 10.

Figure 3.15 shows the data for some simulations we performed for the set of task graphs. With this method it is possible to compare how a system with a certain structure and properties would perform with the same input workload.

This chapter introduced a methodology to derive a machine model for task scheduling on FPGAs. From a description of the hardware and the assumptions a scheduling approach makes about it, a set of constraints is derived. The constraint set is used to generate feasible and optimal schedules. The machine model and its ability to describe FPGA-based accelerators are utilized in Chapters 4 and 5 to optimize schedules and the organization of configurations.

Chapter 4 is concerned with the optimization of schedules at runtime. Given a (subset of a) task graph and a machine model, how can an efficient and feasible schedule be generated in polynomial time?

# Chapter 4

## Polynomial Time Reconfiguration-Aware Scheduling

The scheduling of tasks is an NP-complete problem even for homogeneous processors and ignoring task cost for a bounded number of processors [130]. For heterogeneous processors or even reconfigurable architectures such as FPGAs, the search space is inherently more complex. Finding the optimal schedule can be excessively time-consuming, even for relatively small task graphs.

Hence, even for static scheduling, i.e. without any soft real-time requirement, it can be impractical to find an optimal solution. Therefore, many approaches rely on algorithms that are not guaranteed to result in a global optimum, but use heuristics to trade optimality for practicality. A wide range of approaches exist to perform static scheduling on heterogeneous resources. For an overview see the literature reviews of Boudet et al. [131], Oh et al. [132] and the list of heuristics in Sinnen's work [27, pp. 157–158]. The approaches can be classified into the following categories:

1. *Heuristic-based* algorithms, which are further divided into two categories:
    a) *List scheduling*: a task in the set of ready tasks (the *ready set*) is selected according to some heuristic, assigned a start (or finish time) and a resource to execute on – also according to some heuristic.
    b) *Clustering*: tasks are grouped into clusters that are to be executed on the same resource to minimize communication cost. The clusters are assigned to processors and subsequently the tasks in each cluster are assigned their start (or finish) time.
2. *Guided random search-based* algorithms use random choices to navigate the search space, combining it with information obtained from earlier choices [133].

Heuristic-based algorithms employ a certain strategy to generate solutions that are as near to the optimum as possible but find solutions in polynomial time. The two sub-categories are based on the algorithms' structures. List scheduling algorithms are built around the ready set – a set of tasks that already have all (recursive) predecessors scheduled and are thus *ready* to be scheduled. Tasks from the ready set are selected based on a sorting criterion, for example by giving tasks with higher cost a higher priority in the scheduling process. In contrast, clustering algorithms first group tasks by resource, i.e. the allocation is decided first. Subsequently, the start time per task is selected.

Guided random search-based algorithms directly work on the set of possible schedules instead of constructing one from scratch. The algorithms select schedules from a randomly generated set. The selection process is guided by schedules from earlier random sets.

Algorithms in both categories offer a polynomial runtime which allows scheduling, even in complex scenarios [27, pp. 108 ff.].

In this work we focus on heuristic-based algorithms, since random search-based algorithms yield good results, but suffer from relatively long runtimes. Topcuoglu et al. describe the difference in runtime of random search-based scheduling algorithms as an order of magnitude, compared to heuristic-based algorithms [133]. List scheduling in particular provides a good compromise between low runtime and good results. Clustering performs worse for a finite set of processors (which is relevant to us) [133].

List scheduling algorithms exist for homogeneous and heterogeneous systems. The heterogeneous versions utilize heuristics that incorporate the differences in computation or communication cost in their rank functions. We present an extension for the scheduling on FPGAs based on hardware models that are described in Chapter 3. With the model as a starting point a compatible version of a well-known list scheduling algorithm is introduced.

Based on our previous work [33], we implement two scheduling algorithms with heuristics. For a given problem size, both returned a (local) optimum in under 30 ms. The same problem fed into a constraint programming solver took more than 10 h to find a solution with an equal cost.

The remainder of this chapter is structured as follows:

1. *List scheduling algorithms for heterogeneous machines* in general are introduced in Section 4.1. Variants of them are typically similar apart from minor details to accommodate specific use cases.
2. *Heterogeneous Earliest Finish Time (HEFT)*, a list scheduling algorithm for heterogeneous systems, is described in Section 4.2. The algorithm is well-known and suitable for heterogeneous systems with communication costs.
3. Our new HEFT-based list scheduling algorithm for reconfigurable computing, *Reconfigurable Earliest Finish Time (REFT)*, is introduced in Section 4.3. The section explores in depth the peculiarities of task scheduling with a list scheduling algorithm.

## 4.1 List Scheduling

List scheduling is a popular heuristic-based algorithm class. It describes a range of algorithms that follow the same two-phase structure:

1. *Rank* all nodes of the task graph in a sorted list $L$ according to a priority scheme and the precedence relation.

   This ensures that all nodes scheduled in the second phase are sorted, such that all precedence constraints are fulfilled. This means that all predecessors $\mathsf{pred}(v)$ are guaranteed to appear before task $v$ in $L$.

2. *Map* each node to a processor and *schedule* it.

   The mapping and scheduling apply some form of heuristic, for example the one of Earliest Finish Time (EFT) or as soon as possible (ASAP) techniques. Clearly, both mapping and scheduling, must take the constraints into account to produce a valid schedule.

Algorithm 4.1 shows the general structure of list scheduling algorithms with their two phases. In the first phase (lines 1–3) the tasks $V$ are sorted into a list $L$ by some ranking or sorting function `Rank`. In the second phase (lines 4–9) a processor is assigned for each task and a start time on this processor is scheduled for that task. This is repeated for all tasks in the sorted list $L$ and in the end the allocation and start times are returned as the schedule.

List scheduling algorithms generally also support online/dynamic scheduling, i.e. constructing a schedule as new tasks arrive. In that case, the first phase is skipped, and the second phase is executed for each newly arriving task. This makes this class of algorithms applicable for scheduling at the runtime level as well. However, that is not investigated further here.

**Input:** Task graph $G = (V, C, w, c)$
**Result:** Schedule $\mathcal{S}$
**begin** Phase 1: rank all nodes according to precedence constraints and priority
  $L \leftarrow$ `Rank` (V)
**end**
**begin** Phase 2: map and schedule all nodes in the sorted list $L$
  **foreach** $v \in L$ **do**
    $\mathsf{proc}(v) \leftarrow$ `Allocate` $(v)$
    $t_s(v) \leftarrow$ `Schedule` $(v, \mathsf{proc}(v))$
  **end**
**end**
**return** $(t_s, \mathsf{proc})$

Algorithm 4.1: Generating a schedule $\mathcal{S}$ with list scheduling after [27, p. 109]

Major effects on the resulting schedule are the priority scheme, the mapping and scheduling strategies:

1. The *priority scheme* sorts tasks into a list. Calculating the priority of a node affects the sorting of the list.
2. The *mapping strategy* assigns processors to tasks. Selecting the processor for a node affects the scheduling of the rest of the ready set.
3. The *scheduling strategy* assigns start times to tasks. The point in time at which the task is planned to execute affects the schedule directly.

For each of these components of a list scheduling algorithm, many alternatives and combinations have been described in the literature [27, p. 117].

Originally, list scheduling was introduced for homogeneous systems where all processors have the same capability such that each task has the same cost on all processors [134]. Since we are dealing with heterogeneous processors – PEs in our case – a look into a variant of list scheduling supporting such architectures is warranted.

## 4.2 Heterogeneous Earliest Finish Time

A simple and well-known implementation of list scheduling for heterogeneous machines is the Heterogeneous Earliest Finish Time (HEFT) algorithm by Topcuoglu et al. [133]. Being a list scheduling algorithm, it has the same structure as Algorithm 4.1. Its two adjustments concern the ranking of tasks in phase one and the allocation in phase two of the algorithm:

1. The *upward rank* is used as a ranking function.
2. The *EFT* is used as a heuristic for the allocation.

In the first phase, tasks are ranked by their upward rank, which is defined recursively. It consists of its average cost on all processors and the maximum of its successors' upward rank in addition to the respective communication cost.

**Definition 30** (Upward rank) *Let $G = (V, E, w, c, t)$ be a task graph and task $v_i \in V$, then the upward rank $rank_u(v_i)$ of task $v_i$ is given by*

$$rank_u(v_i) = \overline{w_i} + \max_{v_j \in \mathsf{succ}(v_i)} \{\overline{c_{i,j}} + rank_u(v_j)\}$$

The functions $\overline{w}$ and $\bar{c}$ represent the *average* computation cost over all processors and the *average* communication cost over all processor-pairs, respectively.

$$\overline{w_i} = \frac{\sum_{p \in P} w(i,p)}{|P|} \text{ and} \tag{4.1}$$

$$\overline{c_{ij}} = \frac{\sum_{p_1,p_2 \in P} c(i,j,p_1,p_2)}{|P|^2} \tag{4.2}$$

In the second phase, a processor is selected for the tasks. This is done with the EFT strategy. The function $\mathsf{EFT}(\mathcal{S}_p, v)$ returns the processor that minimizes $v$'s finish time $t_f(v)$. Given a *partial schedule* $\mathcal{S}_p$ (where not all tasks have been scheduled yet) and a task $v$:

$$\mathsf{EFT}(\mathcal{S}_p, v) = \arg\min_{p \in P} t_f(v,p) \tag{4.3}$$

This strategy has a polynomial $\mathcal{O}(|E| \times |P|)$ complexity and typically yields good results on heterogeneous systems [135].

However, HEFT is not sufficient for our purposes of scheduling tasks on FPGA-based accelerators. The unique characteristics of FPGAs in particular and reconfigurability in general are not considered by the algorithm. Therefore, we present our extension to HEFT for reconfigurable systems utilizing the definitions from Chapter 3. The following section explains the motivation for our new algorithm and defines it.

## 4.3 Reconfigurable Earliest Finish Time

The flexibility and constraints of FPGAs must be reflected by the list scheduling algorithm(s). Especially the constraints the hardware imposes on the schedule must be taken into account to generate feasible schedules. For example, if the reconfiguration imposes an overhead, this is typically not considered by scheduling algorithms. In particular, the following properties should be considered by a scheduling algorithm for FPGAs:

- *Conflict* ($\mathcal{P}_c^P$): certain PEs cannot execute simultaneously with a conflicting set of PEs.
- *Reconfiguration overhead* ($\mathcal{P}_r^L$): the cost of *making a PE available* is not supported.
- *Type* ($\mathcal{P}_t^P$): the influence of the hard-coded function in a PE is not supported.

This is the motivation behind our reconfiguration-aware list scheduling approach: Reconfigurable Earliest Finish Time (REFT) is our list scheduling algorithm with support for reconfigurable hardware. It supports all machine models as described in Chapter 3.

The structure of the REFT algorithm is similar to HEFT – or any list scheduling algorithm – but it explicitly supports:

- *Locations*: the communication cost is based on locations, rather than processors or PEs.

- *Configurations and reconfigurability*: the notion of instances ensures conflict-free operation.
- *Reconfiguration overhead*: the cost of reconfiguration is handled in the schedule.

REFT uses three strategies to generate feasible and preferably optimal schedules: instances, a new ranking function and a reconfiguration-aware EFT function. These strategies and the REFT algorithm are described in the following sections:

1. Invalid *overlaps* in the schedules are prevented in REFT as described in Section 4.3.1.
2. A *ranking function* specifically handling the characteristics of reconfigurable systems is described in Section 4.3.2.
3. The *REFT* algorithm itself is detailed in Section 4.3.4.
4. The *communication congestion* is handled by a second variant of REFT which is presented in Section 4.3.5.
5. The *complexity* of both REFT variants is discussed in Section 4.3.6 .

### 4.3.1 Using Instances to Avoid Undesirable Overlaps

To support locations and the allocation of configurations, REFT relies on instances from Definition 16. Every allocated task requires an active instance, because some PE must execute it and hence a configuration must be instantiated at some location. When a task is allocated and/or the EFT is evaluated for a PE $p$ at location $l$, REFT performs the following steps:

1. Set the disjoint interval $A_{p,l} = [0, \infty)$.
2. For each scheduled instance $\mathcal{I} = (c_i, l, i)$, remove the interval $i$ plus optional reconfiguration overhead from interval $A_{p,l}$ if the configuration $c_i$ is not equal to $\mathsf{conf}(p)$.
3. For each scheduled task, remove its interval from $A_{p,l}$ if the PE is $p$.

The remaining disjoint interval $A_{p,l}$ represents all available slots for the execution of tasks, since it is ensured that PE $p$ is available and either configuration $\mathsf{conf}(p)$ or no configuration is instantiated. This method avoids overlaps of configurations on locations and overlaps of tasks on PEs but allows reconfigurations and ensures that the schedule is feasible, since the properties of the hardware model are considered.

### 4.3.2 Ranking

Recall that HEFT uses the upward rank $\mathsf{rank}_u$ to sort the tasks into a priority list. A high upward rank indicates that the finish time of a task has a significant influence on the total schedule. More specifically, the entry task $v_e$ has the highest upwards rank and all (recursive) successors have a rank of at most $\mathsf{rank}_u(v_e)$. The tasks on the critical path – of which $v_e$ is a part by definition – all have the highest upper rank amongst all their siblings. In other words: the upper rank describes the minimum distance between the start time of a task and the makespan of a schedule, i.e. its length.

Since the critical path provides a lower bound on the schedule length, sorting tasks by the upper rank gives priority to the scheduling of those tasks. However, $\mathsf{rank}_u$ does not take into account any artifacts of reconfigurations and the machine model. A set of new ranking functions are introduced to accommodate for these characteristics.

The *pressure* is defined to represent the weighted cost of instantiating the resources to execute a task $v_b$ after a task $v_a$ was executed previously. Since this cost is calculated *before* the resource allocation is done, it can only be a heuristic.

| Configuration | Location | Reconfiguration |
|---|---|---|
| same | same | unlikely |
| | different | possible |
| different | same | necessary |
| | different | possible |

Table 4.1: Reconfiguration possibilities

Table 4.1 shows the four possible scenarios – one per line – that can occur when scheduling two adjacent tasks. The first column denotes whether both tasks are executed on a PE that is contained in the same or a different configuration. The second column describes whether the location at which both tasks are executed is the same or a different one. The third column describes whether a reconfiguration is unlikely, possible or necessary in this scenario. As depicted in row three, for example, two tasks are executed in the same location, however have a different configuration on PEs, therefore a reconfiguration must be performed, since the configuration containing the second PE must be instantiated. Or alternatively, even if both tasks share the same location and the same configuration as in row one of Table 4.1, a third task could be scheduled between them on the location using a different configuration, which would result in two reconfigurations. If the location is not the same, i.e. the tasks are executed on different locations, we cannot make any assumptions about the reconfiguration behavior at this point.

To incorporate this dynamic behavior into the ranking function, we introduce the reconfiguration *pressure* between two tasks. For two adjacent tasks $v_a \in V$ and $v_b \in \mathsf{succ}(v_a)$, pressure is defined as:

$$\mathsf{pressure}(v_a, v_b) = \frac{\sum_{(l_a,l_b,p_a,p_b)\in P_a\times P_b} 0^{|l_a-l_b|}\cdot\left(1-0^{|\mathsf{conf}(p_a)-\mathsf{conf}(p_b)|}\right)\cdot \mathcal{P}_c^L(l_a)}{|P_a|\cdot|P_b|\cdot|L_a|\cdot|L_b|} \tag{4.4}$$

For all *possible* combinations of locations and configurations for the two given tasks, it evaluates the weighted cost of reconfiguration. This is an optimistic approach in two regards:

1. A scenario with an equal location and an equal configuration will add 0 cost.
2. A scenario where only the location differs will add 0 cost.

The ranking function used in REFT is the upper rank with pressure, $\mathsf{rank}_{u,p}$:

$$\mathsf{rank}_{u,p}(v_i) = \overline{w_i} + \max_{v_j\in\mathsf{succ}(v_i)}\{\overline{c_{i,j}} + \mathsf{pressure}(v_i, v_j) + \mathsf{rank}_u(v_j)\}. \tag{4.5}$$

This gives those tasks a higher rank that are more likely to introduce a reconfiguration overhead and thus would increase the makespan, everything else being equal. The rationale behind this has to do with the EFT that REFT selects for each task. If a task $v_r$ is more likely to introduce reconfiguration overhead than task $v$, giving $v_r$ a higher priority will allow REFT to schedule it before task $v$. Thus, REFT may select an earlier EFT (and start time) for task $v_r$. When $v$ is scheduled later REFT might find an EFT that is later than if it had been scheduled before $v_r$, but the maximum EFT of either of the tasks is lower, since reconfiguration may have been avoided. Basically, the ranking gives REFT additional information about a task's possible negative effect on the schedule length.

### 4.3.3 Determining the EFT

Finding the EFT for a given task and a given partial schedule is deterministic and possible in polynomial time. Thus, it does not need to be optimized for performance, but for conformance with the machine model. In particular, the EFT must respect the properties like reconfiguration overhead and the communication overhead that depend on the location.

To identify the EFT for a given task from task graph $G = (V, E, w, c, t)$ for machine model $\mathcal{M} = (C, L, \mathcal{P})$, REFT evaluates the lowest consecutive interval that has at least $w(v, p)$ length for a task $v \in V$, the PE $p \in P$ and a location $l \in L$:

$$\mathsf{EFT}(v, p, l) = \max\left\{[a, b) \mid b - a = w(v, p), a \geq t_d(v), [a, b) \subseteq A_{p,l}\right\} \text{ and} \tag{4.6}$$

$$\mathsf{EFT}(v, p) = \min_{l \in L_v} \mathsf{EFT}(v, p, l). \tag{4.7}$$

Different from HEFT's evaluation of EFT is the usage of the intervals $A_{p,l}$ that each hold an interval denoting whether PE $p$ on location $l$ is free, i.e. able to execute work.

### 4.3.4 The REFT Algorithm

The REFT algorithm is a list scheduling algorithm with reconfiguration-aware enhancements and adapts HEFT to the characteristics of FPGAs-based accelerators as described above. Algorithm 4.2 shows the complete algorithm. It takes a task graph $G = (V, E, w, c, t)$ and a machine model $\mathcal{M} = (\mathcal{A}, \mathcal{T}, \mathcal{P})$ as input and produces a feasible schedule $\mathcal{S} = (t_s, \mathsf{proc})$ as output. In the first step, in lines 1–3, the tasks $V$ are ranked, similar to HEFT. This ranking is done using the upper rank with pressure $\mathsf{rank}_{u,p}$ to reflect additional cost for reconfigurations. In the second phase, in lines 4–13, it iterates over the free set and selects the processor that minimizes the EFT using the modifications described above for the highest rated task (line 10). When a task is scheduled, REFT keeps track of the allocation for each configuration-location combination (line 11).

Recall that we defined several communication settings in Section 3.4.5. REFT, that is Algorithm 4.2, outputs schedules that are valid *with communication*. This means that communication is modeled, but congestion is not.

### 4.3.5 REFT with Communication Congestion

Algorithm 4.2 does not model the congestion of the communication – in fact it does not consider a possible topology at all. The communication itself is represented through the cost function $c$ and the edge finish time. A task can only start executing once its data is ready, which is encoded in its EFT. As described in Section 3.2.2, merely attributing a cost to an edge in the task graph may not represent the real world behavior accurately, since it leaves out the topology of the underlying hardware and ignores contention on communication links.

Three changes are made to REFT to introduce congestion-awareness:

1. A *topology* is an additional input to the algorithm.
2. All *routes* of the communicating PEs are computed.
3. The EFT function considers *edge scheduling*.

The congestion-aware REFT algorithm consists of three phases rather than two, which are described separately below due to the increased complexity.

**Input:** Task graph $G = (V, C, w, c, t)$
Machine model $\mathcal{M} = (C, L, \mathcal{P})$
**Result:** Schedule $\mathcal{S} = (t_s, \mathsf{proc}, \mathsf{loc})$
/* $(L, \mathcal{R})$ is an ordered set with the relation
$\mathcal{R} = \{(v_a, v_b) \in V \times V : \mathrm{rank}_{u,p}(v_a) \leq \mathrm{rank}_{u,p}(v_b)\}$) */
**begin** Phase 1: rank all nodes according to precedence constraints *and* priority
  $L \leftarrow V$
**end**
**begin** Phase 2: find the EFT for all tasks and schedule them
  **foreach** *task* $v \in L$ **do**
    $p \leftarrow \arg\min_{p \in P_v} \mathsf{EFT}(v, p)$
    $l \leftarrow \arg\min_{l \in L_v} \mathsf{EFT}(v, \mathsf{proc}(v), l)$
    $\mathsf{proc}(v) \leftarrow p$
    $\mathsf{loc}(v) \leftarrow l$
    $t_s(v) \leftarrow \mathsf{EFT}(v, \mathsf{proc}(v), \mathsf{loc}(v)) - w(v, p)$
    /* Remove the interval from the new instance from $A_{p,l}$ */
    $A_{p,l} \leftarrow A_{p,l} \setminus [t_s(v), t_f(v)]$
  **end**
**end**
**return** The schedule $\mathcal{S} = (t_s, \mathsf{proc}, \mathsf{loc})$

Algorithm 4.2: REFT

1. *Generate routes* between all pairs of PEs.
2. *Rank* all nodes of the task graph just as before in Algorithm 4.2.
3. *Map* each node to a processor, each edge to links along its corresponding route and *schedule* both using the EFT heuristic.

#### 4.3.5.1 Phase 1 of REFT

The newly introduced first phase uses edge scheduling as described in Section 3.4.5.3 to make REFT congestion-aware. First, a topology is needed to define the communication paths. The topology graph $\mathcal{T} = (N, O, D, b)$ is an input to the extended REFT algorithm. If none is explicitly provided, we assume the default topology (see Section 3.2.2) that models an FPGA-based accelerator with a global memory. In the phase, a set of routes for all pairs of PEs within the topology is computed. Any fitting path finding algorithm can be used for this purpose. For example, Dijsktra's well-known algorithm to find the shortest path between two points in a graph with (positively) weighted edges [136] is a good fit. The routes are static throughout the scheduling, so they can be computed once upfront. Additionally, for any given default topology, the routes can be specified without invoking a path finding algorithm. For any two PEs $p_s$ and $p_d \in O$ with $\mathsf{loc}(p_s) = l_s$ and $\mathsf{loc}(p_d) = l_d$, the route $R$ from $p_s$ to $p_d$ is

$$R = \begin{cases} \langle (p_s, r_s), (r_s, l), (l, t_s), (t_s, p_d) \rangle & \text{if } \mathsf{loc}(p_s) = \mathsf{loc}(p_d), \\ \langle (p_s, r_s), (r_s, l_s), (l_s, l_d), (l_d, t_d), (t_d, p_d) \rangle & \text{otherwise} \end{cases}. \tag{4.8}$$

#### 4.3.5.2 Phase 2 of REFT

The second phase, ranking, is the same as the first phase of Algorithm 4.2.

#### 4.3.5.3 Phase 3 of REFT

In the third phase we follow a similar strategy as instance tracking to adjust the EFT, i.e. to keep track of the active instantiation and their PEs for a schedule. Algorithm 4.2 uses the EFT to insert tasks in the earliest possible idle gap of a PE. It tracks the availability (and conflicts) of PEs and configurations with the interval $A_{p,l}$. A new set of intervals $B_d$ is introduced. It tracks the occupancy of link $d \in D$ by edges in the task graph, similar to the instance tracking as described in Section 4.3.1. If an edge $e \in E$ of the task graph $G = (V, E, w, c, t)$ is to be scheduled in the link $d \in D$, REFT performs the following three steps:

1. Initialize $B_d = [0, \infty)$ once for each $d \in D$.
2. Find the sub-interval $[a, b)$ in $B_d$ such that $b - a \leq \zeta(e, d)$ holds and that minimizes $b$.
3. Remove the interval $[a, b) = \big[t_s(e, d), t_f(e, d)\big)$ from $B_d$.

This change to REFT affects the EFT for all tasks. To create a congestion-aware schedule, the algorithm must especially take constraints 5 and 6 (Link and Causality) into account. Constraint 5 is satisfied by the occupancy interval $B_d$ as described above. For constraint 6 it is sufficient to redefine the edge finish time $t_f$ accordingly. Let $G = (V, E, w, c, t)$ be a task graph, $\mathcal{T} = (N, O, D, b)$ a topology, an edge $e_{ij} = (v_i, v_j) \in E$, PEs $p_s, p_d \in P$ and route $R = \langle \ldots, d_l \rangle$ from $p_s$ to $p_d$, then for a congestion-aware REFT version, the edge finish time is either equal to the task's finish time $t_f(v_i, p_s)$ if the communication is local or equal to the finish time on the last link otherwise:

$$t_f(e_{ij}, l_s, l_d) = \begin{cases} t_f(v_i, p_s) & \text{if } p_s = p_d, \\ t_f(e_{ij}, d_l) & \text{otherwise} \end{cases}. \tag{4.9}$$

Algorithm 4.3 shows the REFT variant generating congestion-aware schedules according to Definition 27. In the new first phase (lines 1–3), routes between all PE nodes in $T$ are statically computed. The second phase (lines 4–6) is identical and in the third phase (lines 7–22), the definition of $t_f$ from above ensures that the EFTs of all tasks are adjusted to account for congestion, because it factors in the link occupancy. The occupancy is tracked with the intervals $B_d$, that are set accordingly at the end of the algorithm (lines 13, 18).

Figure 4.1 shows output of congestion-aware REFT for a random task graph $G = (V, E, w, c, t)$ with $|V| = 25$ and $|E| = 15$ for a machine model $\mathcal{M}$ generated with the method as described in Section 6.1.1. The figure shows three aspects of the same schedule from top to bottom:

- A *task schedule*
- An *edge schedule*
- A *configuration schedule*

The task schedule shows the occupancy of three PEs (0, 1, 2) on two locations (0 and 1) over time. The colored bars in the task schedule denote whether a task is executing on a PE. The color of the bars serves the purpose of distinguishability of tasks.

**Input:** Task graph $G = (V, E, w, c, t)$
Machine model $\mathcal{M} = (\mathcal{A}, \mathcal{T}, \mathcal{P})$
Routing algorithm $r\colon P \times P \to \mathbb{P}(D)$ for $\mathcal{T}$
**Result:** Schedule $\mathcal{S} = (t_s, \mathsf{proc}, \mathsf{loc})$
/* $(L, \mathcal{R})$ is an ordered set with the relation
$\mathcal{R} = \{(v_a, v_b) \in V \times V\colon \mathrm{rank}_{u,p}(v_a) \leq \mathrm{rank}_{u,p}(v_b)\}$) */
**begin** Phase 1: for all pairs of PEs $p_s, p_d \in O$, generate a route
  $R_{sd} \leftarrow r(p_s, p_d)$
**end**
**begin** Phase 2: rank all nodes according to precedence constraints *and* priority
  $L \leftarrow V$
**end**
**begin** Phase 3: find the EFT for all tasks
  **foreach** *task* $v_j \in L$ **do**
    $p \leftarrow \arg\min_{p \in P_v} \mathsf{EFT}(v, p)$
    $l \leftarrow \arg\min_{l \in L_v} \mathsf{EFT}(v, \mathsf{proc}(v), l)$
    $\mathsf{proc}(v) \leftarrow p$
    $\mathsf{loc}(v) \leftarrow l$
    $t_s(v) \leftarrow \mathsf{EFT}(v, \mathsf{proc}(v), \mathsf{loc}(v)) - w(v, p)$
    /* Remove the interval from the new instance from $A_{p,l}$ */
    $A_{p,l} \leftarrow A_{p,l} \setminus \left[t_s(v), t_f(v)\right)$
    **foreach** *task* $v_i \in \mathsf{pred}(v_j)$ **do**
      **foreach** *link* $d \in R_{ij}$ **do**
        $e \leftarrow (v_i, v_j)$
        $B_d \leftarrow B_d \setminus \left[t_s(e, d), t_f(e, d)\right)$
      **end**
    **end**
  **end**
**end**
**return** The schedule $\mathcal{S} = (t_s, \mathsf{proc}, \mathsf{loc})$

Algorithm 4.3: The REFT algorithm variant to generate congestion-aware schedules

The edge schedule shows the occupancy of 19 links of the topology over time. The colored bars show whether an edge is being communicated on a given link at any given time. The edges are color-coded, i.e. each edge in the task graph has its own color.

The configuration schedule shows the occupancy of two locations over time, i.e. when a configuration is instantiated on a location. This corresponds to the combined occupancy shown in the task schedule. It shows the active configuration for both locations, encoded by color. Since we only used one configuration for brevity, the plot merely shows the active intervals for the configurations.

The machine model $\mathcal{M}$ used in Figure 4.1 has the number of locations $|L| = 2$ and one configuration $c \in C$ with 3 PEs, i.e. $|c| = 3$. For all $v \in V$ the cost is uniformly $w(v, p) = 100$, while all edges $e \in E$ have the cost $c(e) = 10$. The configuration schedule shows REFT using both locations and location 1 having an active instance up until time 420. The uniform task cost of 100 creates phases of congestion on the communication links. For example, at around 100 the output

data from the first tasks is transferred to their respective successors. This congestion results in gaps in the task schedule, clearly visible for example at time 100 on PE 2 at location 0.

### 4.3.6 Complexity of REFT

Recall that – in contrast to the constraint programming solution presented in Chapter 3 – both REFT variants are algorithms of polynomial time complexity. Nonetheless, it is important to understand the runtime behavior of the algorithms, especially with regard to the changes to HEFT. The asymptotic complexity of both algorithms is described below.

#### 4.3.6.1 Complexity of Algorithm 4.2

The asymptotic complexity of REFT as shown in Algorithm 4.2 for creating a schedule from a task graph $G = (V, E, w, c, t)$ and a machine model $\mathcal{M} = (\mathcal{A}, \mathcal{T}, \mathcal{P})$ is deduced as follows:

- For each task, the upper rank with pressure $\mathsf{rank}_{u,p}$ must be computed. Since this must only be computed once per task, the complexity for this step is $\mathcal{O}(V + E)$.
- The original HEFT algorithm has a complexity of $\mathcal{O}(E \times P)$ (or $\mathcal{O}(V^2 \times P)$ for a *dense* graph with $E \propto V^2$) [133].
- REFT introduces locations, so the complexity grows proportional to the number of locations: $\mathcal{O}(E \times P \times L)$.

This makes the total complexity of the algorithm $\mathcal{O}((V + E) + E \times P \times L)$, or $\mathcal{O}(V + E \times P \times L)$. The number of PEs and locations are additional factors compared to HEFT's complexity. In practice, these factors are not expected to have a large impact, since neither the number of PEs $|P|$ nor the number of locations $|L|$ will grow considerably (especially compared to the number of tasks $|V|$ and edges $|E|$). Therefore, the overall impact of our changes to HEFT is expected to be small.

#### 4.3.6.2 Complexity of Algorithm 4.3

The asymptotic complexity of REFT with support for congestion, as shown in Algorithm 4.3, is deduced as follows:

- The complexity of the first phase depends on the complexity of the routing algorithm. Suppose algorithm $r$ has the complexity $K$, then the first phase is an $\mathcal{O}(O \times K)$ operation.
- Scheduling the communication graphs at the end of phase 3 is an *additional* $\mathcal{O}(V \times N)$ operation.

This makes the additional complexity $\mathcal{O}(O \times K + V \times N) = \mathcal{O}(P \times K + V \times N)$, so in total the algorithm requires $\mathcal{O}(P \times K + V + E \times P \times L + P \times K + V \times N) = \mathcal{O}(V \times N + P \times (E \times L + K))$. Again, $P$, $L$, and $N$ are dependent on the machine model and the topology alone. Thus, they are typically small and static and could arguably be removed from the asymptotic consideration.

In essence, both algorithms *do* increase the complexity compared to HEFT, but are still linear with respect to the number of tasks and edges.

This chapter introduced list scheduling, a well-known class of polynomial time task scheduling algorithm. An established list scheduling algorithm for heterogeneous architectures, HEFT, and its building blocks were detailed. Building on this existing algorithm, we presented our new reconfiguration-aware list scheduling algorithm, REFT, along with a variant that supports

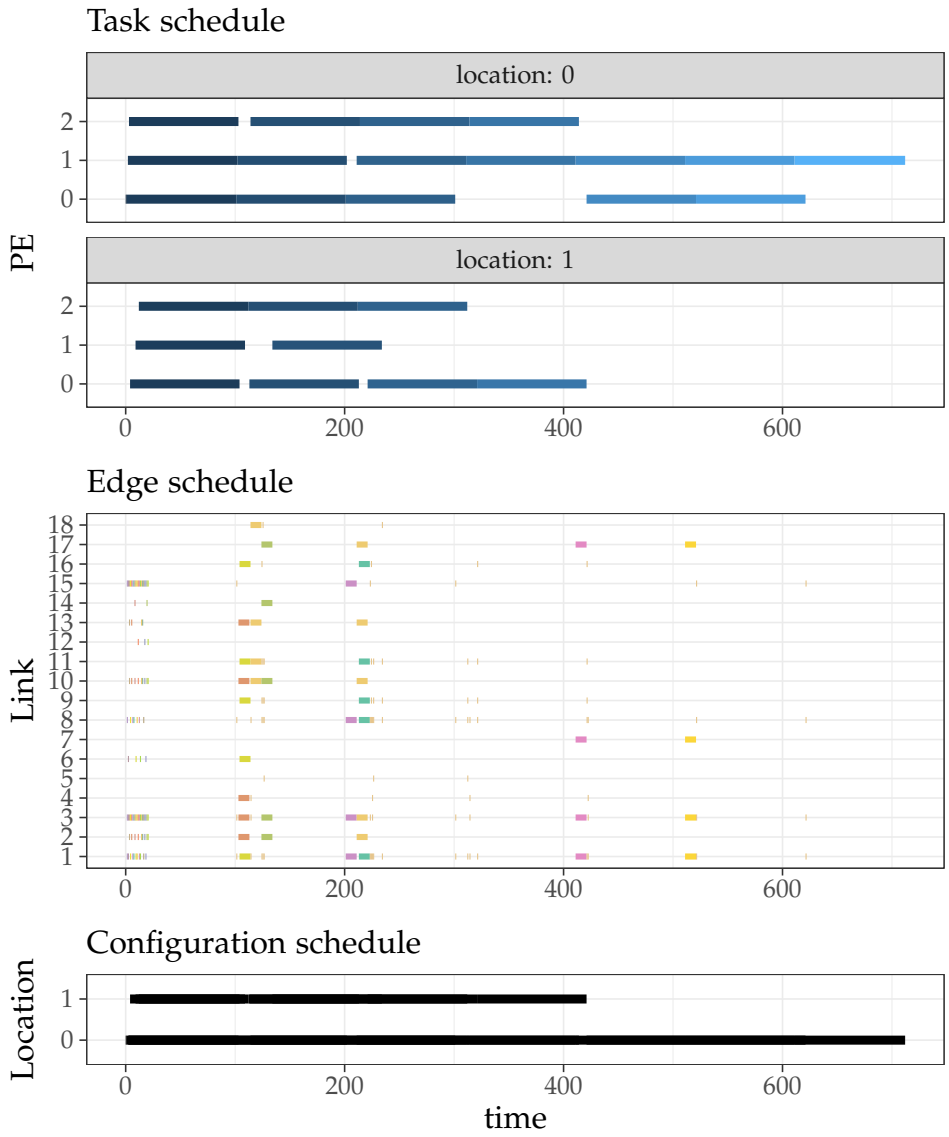



Figure 4.1: A task schedule and edge schedule for a randomly generated graph with 25 tasks and 15 edges

communication congestion. Our motivation and the extensions to HEFT as well as the asymptotic complexity are described in detail.

Chapter 5 is concerned with the optimization of PEs and their organization into configurations. Given a task graph and a machine model, how can PEs be arranged into configurations? Where should computational resources be allocated to optimize the schedules? To answer these questions, REFT is used as a part of an optimization procedure to quickly generate a large range of schedules.

# Chapter 5

# Design Space Exploration

Chapters 3 and 4 are concerned with finding the optimal schedule for a given machine model and a given task graph. In conventional computing scenarios, the potential for optimization is depleted after an optimal schedule is found. The hardware is static and already optimized for some use case and the task graph can only be changed by a different algorithm – something that possibly requires manual intervention.

## 5.1 Optimizing Reconfigurable Computing

In reconfigurable computing there exists another dimension that can be optimized: the equivalent of the hardware itself in the form of configurations. Although the input HLS code describes the semantic of the program, compilers are free to generate any configuration that represents the program's semantic. We model the hardware as a machine model (see Chapter 3) and apply three strategies for the optimization:

1. IntraCO: changes contained within a configuration to affect the behavior of the circuits
2. InterCO: migrations between configurations
3. CO: the combination of both, IntraCO and InterCO

IntraCO will reserve more or different resources for some circuit in order to speed up a computation, make it more energy efficient or allow for different schedules. Listing 5.1 shows an OpenCL code that trivially uses such optimizations. The code shows a vector addition of two double precision floating point numbers. It can be translated to a configuration for an FPGA-based accelerator. Line 2 sets the *unroll factor* `N`. In conventional computing following the Von Neumann model, unrolling a loop will reduce the overall overhead of the loop condition by executing more instructions per iteration. On an FPGA, unrolling a loop will typically allow for a deeper pipeline, i.e. to process more elements of the loop per step, and could yield better performance. However, it will also require more resources, could limit the maximum frequency of the chip and even have no benefit at all, because of stalled loop iterations or memory bandwidth limits.

```
1 __kernel void add(__global double *a, __global double *b, __global
    ↪ double *result, int n) {
2     #pragma unroll(N)
3     for (int i = 0; i < n; i++)
4         result[i] = a[i] + b[i];
5 }
```

Listing 5.1: OpenCL code to add two vectors of size n. The loop is unrolled by a factor of N.

InterCOs describe the migration of PEs to other configurations. As established in Chapter 3, scheduling approaches for reconfigurable hardware assume a one-to-one mapping of PEs to configurations. While this is not unreasonable, it suffers from the described drawbacks of large development overhead and limited portability.

Recent high-level parallel programming environments like Intel OneAPI [126] or implementations of the OpenMP target directive [137] support offloading to FPGA-based accelerators. This continued shift towards higher level of abstractions is convenient for users, but it also transfers some responsibilities from the user to the runtime and compiler(s). Among others, the tools must organize the accelerated functions as PEs within configurations. For simple cases it is possible to combine all PEs of a program into one configuration to avoid reconfiguration. In Section 5.2.1 we show that in highly optimized codes, such as HPC programs, the organization of PEs can have an effect on the program's performance, due to reconfiguration overhead and congestion. We see potential to aid the compiler in two categories:

- *IntraCOs* may balance the resources of PEs inside a configuration according to some cost function.
- *InterCOs* may automatically identify beneficial combinations of PEs that perform better inside a single configuration.

Grouping and weighing the resources of PEs to achieve a better execution is the goal, but the search space is large. Even just considering grouping the implementation of $n$ functions into an arbitrary number of configurations $c \leq n$ results in $c^n$ combinations[1] of arrangements. Identifying the right combinations and – if possible – including some InterCOs in the search space is our goal.

To answer RQ 3 we are using the notion of machine models and definitions from Chapter 3. Once the underlying hardware is described in terms of a machine model and one or more task graphs are available, can we recommend inter- and intra- configuration optimization to aid authors of software and automatic tools alike? In essence, the goal is to find a modified machine model, given an existing one for a set of task graphs.

The remainder of this chapter describes our process to achieve that goal in five subsections:

- The process to *modify machine models* in general and to evaluate the effects on schedules is discussed in Section 5.2. Accurately modeling the behavior of a machine model after it has been changed, for example by adding PEs, can be challenging. The section highlights the problems with resource usage modeling and our approach to it.
- A GA automates the *optimization of machine models* and explores the search space of possible machine models in Section 5.3. The GA's purpose is to automatically generate new – potentially better – machine models using the knowledge from Section 5.2 and then evaluate their performance compared to other machine models. The structure and functionality of GAs was introduced in Section 2.4, here we describe the details of our adaptation of the GA to machine model optimization.
- Our *three optimizations* InterCO, IntraCO and CO are presented in Sections 5.4, 5.5 and 5.6, respectively.

[1] One can think of the problem as coloring $n$ shapes in at most $c$ colors.

## 5.2 Applying Optimizations by Modifying the Machine Model

The purpose of optimizations such as IntraCO is to optimize the scheduling and therefore to lead to a better application runtime. However, IntraCOs have trade-offs: they introduce changes to the hardware model of which the effects are difficult to predict accurately. The reasons are the following:

- Increasing the resource usage of a PE does not have to result in better performance.
- Changing the resources or increasing the resources for one function in a configuration may influence other functions in the same configuration negatively.
- If a task only has a short run time or is not in the critical path, any optimizations of the PE will not lead to a faster execution.

The process of predicting the performance of changes in the hardware model is also called performance modeling. Suppose we know how a program behaves on a fixed hardware, either directly through tracing or indirectly through a task graph. If this hardware is changed, the effects must be modeled or evaluated to assess whether the modified version performs better, e.g. generating shorter makespans. An evaluation of the hardware for every change is time-consuming for FPGAs, since the compilation can take hours per configuration. Hence, we resort to (performance) modeling.

### 5.2.1 Performance Modeling

There have been attempts to project the performance of IntraCO in a HLS environment, especially with the goal of DSE. Since the synthesis process to the bitstream is extremely time-consuming, especially for a large search space, the performance of an applied optimization must be estimated and typically cannot be measured effectively. Additional to the uncertainties of predicting the performance of executing tasks on a given hardware, FPGAs suffer from the effects of congestion. As more resources are allocated to optimize the cost of executing some tasks, the performance may degrade. This congestion can be divided into two categories, static and dynamic:

- Dynamic congestion affects the memory bandwidth and other resources at runtime.
- Static congestion affects the chip resources like LUTs, LBs, DSPs and the interconnect.

For an accurate prediction of the performance, both categories must be considered. Together, they result in non-linear behavior when optimizations are applied.

#### 5.2.1.1 Static and Dynamic Congestion

The content of this section is based on our paper: P. Jungblut and D. Kranzlmüller, "Dynamic spatial multiplexing on FPGAs with OpenCL," in *International Symposium on Applied Reconfigurable Computing*, Rennes, France: Springer, 2021, pp. 265–274.

Dynamic congestion is handled by the scheduling algorithm, since the congestion appears at runtime. An example of an approach to handling dynamic congestion is described in Section 3.4.5.3, specifically the derivation of scheduling constraints by applying edge scheduling.

Static congestion, however, is determined by the resource usage and it can be observed even when only one PE is executing. The mere presence of other PEs and their resource usage have a negative effect on the executing PE.

Figure 5.1 shows the effect of resource utilization on an Intel Stratix 10 DX accelerator card. It shows the performance in Floating Point Operations per Second (FLOPS) for two PEs combined in one configuration on the y-axis for a matrix of given size on the x-axis. Both PEs implementing the same OpenCL kernel (a Double Precision General Matrix Multiplication (DGEMM)). PE 1 is given more chip resources using the Single Instruction, Multiple Data (SIMD) width attribute (`((num_simd_work_items(n))`). The plot marked 1× does not use SIMD, while the SIMD width of PE 1 is 2 in the middle and 4 on the right. Notably, the plot also shows that the *other* PE 2 is affected by that allocation, although the input code was not changed at all. Where PE 2 can generate 38 GFLOPS in the left plot, its maximum performance drops to 25 GFLOPS in the right plot. The performance of these PEs was measured sequentially, i.e. the effect can be purely explained by the static and not by dynamic congestion.

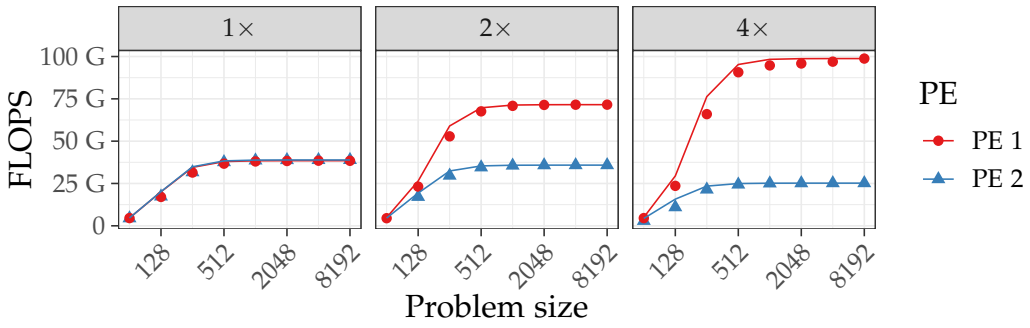


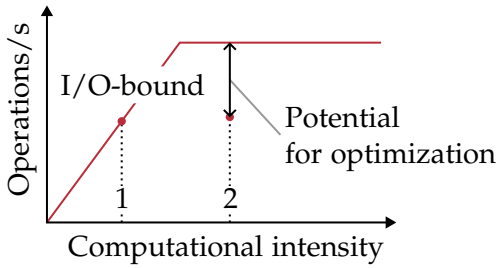

Figure 5.1: Effect of resource usage on the performance of PEs

The roofline model and roofline plots [138] are a straightforward way to assess the potential performance of a given code on a given processor or accelerator. The Computational Intensity (CI) of a code is the number of operations performed per loaded byte. In the roofline model, the maximum operations per second for a given CI is compared against the actual achieved operations per second of a code. The x-axis of a roofline plot as depicted in Figure 5.2 (a) shows the CI and the y-axis shows operations per second. A line depicting the hardware's maximum operations per second increases from 0 to a maximum that is reached when the maximum number of operations per second equals the processors peak operations per second and it is no longer bound be the memory bandwidth. Increasing the CI further will not allow a better maximum performance. Such a maximum can be seen in Figure 5.2 (a) as a red line. The performance of two codes 1 and 2 is depicted as red dots. Code 1 on the left is at the maximum operations per second. To allow more performance, the CI of the code would have to be increased, for example by changing the algorithm or by changing to data types requiring less data. Code 2 on the other hand has the same number of operations per second but has a higher CI. The difference between the "roofline" and the code's data point can be seen as potential for optimization, since it has not reached the theoretical limit of the hardware.

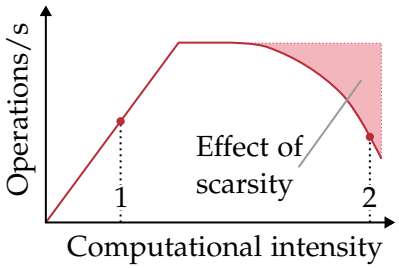
Da Silva et al. [139] present an extension to the roofline model for FPGAs. Increasing the CI is part of the manual or automatic optimization that happens in the development process or during the compilation phase, respectively. The extended roofline for HLS takes into account the scaling factor (SC). Due to the factor, it is not possible to add arbitrarily many PEs or optimizations to the design. Figure 5.2 shows a comparison between the conventional roofline model in Figure 5.2 (a) on the left and one that has been adjusted for FPGAs in Figure 5.2 (b) on the right. Both graphics show two codes 1 and 2 with different CI. Code/PE 1 (left) with a lower CI is I/O-bound and a more optimized version (of the PE) with a higher CI was introduced as

code/PE 2. In Figure 5.2 (a), code 2 on the right is below the roofline and therefore has potential left for optimization, since it is not bound by the limits of the hardware. A software engineer can be tasked with detailed analysis to determine why the code does not reach the maximum performance.

In Figure 5.2 (b) PE 2 has fewer operations per second, although it has a higher CI. This is due to the effect of the resources that are consumed by the PE (and other PEs on the same configuration), to increase the CI.




(a) Traditional roofline model [138]




(b) Da Silva roofline model [139]

Figure 5.2: Comparison of the traditional roofline model and the model proposed by Da Silva with resource congestion

The experimental evaluation of Da Silva et al. [139] is not conclusive, since it shows the effect of congestion only for some cases (including no counterexample). However, the underlying principle of congestion can easily be observed, as we did in our paper [34]. The evaluation by Nguyen et al. shows a similar picture (Fig. 24) where the saturation can be seen for more optimized kernels and higher CI [77].

Static congestion is one reason why predicting the outcome of an optimization of a machine model is particularly difficult. Even without considering the congestion, predicting the performance of optimizations in PEs is not an easy task.

#### 5.2.1.2 Performance of Optimized PEs

Figure 5.3 shows a series of measurements performed by Wang et al. by applying a range of optimizations on a PE [140]. The optimizations are:

- CU$x$: cloning a PE $x$ times
- UL$x$_UL$y$: loop unrolling with a factor of $x$ for the inner and $y$ for the outer loop, respectively.
- SM: local memory optimization (caching)
- MC: memory address optimization

Combinations of these are denoted with "A_B_...". A few observations can be made by comparing the predictions of the authors' model with the measurements:

1. Not every optimization scales linearly. For example, CU16 does not give a 16-fold speedup over Baseline, although the PE was cloned 16 times. Similarly, UL8_UL4 and UL8_UL8 perform approximately equal, although the author's estimation predicts a doubling in performance, as can be seen by the dashed line.

2. Combinations of optimizations will not always scale multiplicatively. For example, the combination SM_UL8_UL8_MC provides a 1,7-fold speedup over SM_UL8_UL4_MC, although UL8_UL4 and UL8_UL8 perform approximately equal.
3. Even with this limited set of variants (we call them "parameters" below), the search space is too large to compile. Even if compiling every variant takes only 30 minutes, just the shown 18 combinations would result in a 9 hour wait. Evaluating all variants and combinations would be $1 + 4 \cdot 5 \cdot 2 \cdot 2) \cdot 5\,\text{h} = 40$ hours.

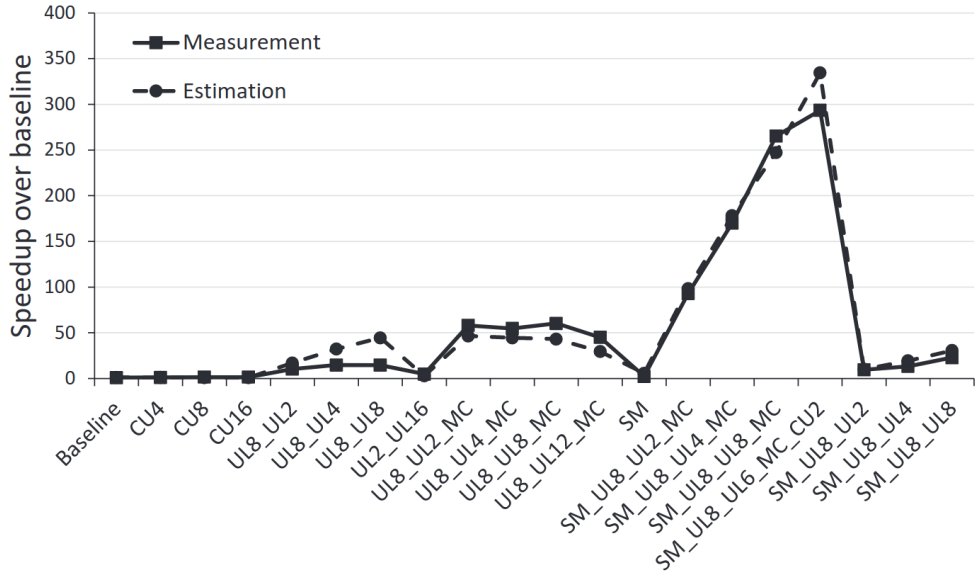



Figure 5.3: A range of optimizations applied to a single matrix-matrix multiplication kernel [141]

This behavior is exemplary for many optimizations and their combinations on FPGAs. Hence, a strategy to incorporate the effects of congestion and performance increases must be included in the DSE process. There are some methods to achieve that for HLS code.

Other approaches to performance modeling require runtime data but provide more accurate results (rather than an upper bound). An analytical approach is taken by Wang et al. [140], [141], where an IR of the compilation step is analyzed automatically to predict the outcome of optimizations. Both approaches use runtime profiling to gain information about the memory access patterns. Especially the FlexCL model achieves good accuracy at an 8.7% and 9.5% error rate for the Rodinia [142] and Polybench [143] benchmark suites, respectively. FlexCL is compared to the prediction accuracy of Xilinx' analysis which only achieves an error rate between 30.4% and 84.9% for the same benchmarks. This suggests that even vendor tools can only provide rough estimates of the effects of optimizations on the performance.

The implementations of Wang et al. [140], [141] are unfortunately not available, thus we resort to vendor tools for the performance estimation in the process. However, the method of performance modeling is exchangable in our process.

### 5.2.2 Updating a Task Graph for an Optimized Machine Model

To explore a potential optimized machine model $\mathcal{M}'$ for a given machine model $\mathcal{M}$, the input task graph $G = (V, E, w, c, t)$ is modified to $G' = (V, E, w', c', t)$, where the updated computation cost $w'$ and communication cost $c'$, respectively. The task graph $G'$ represents the task graph as

executed by the optimized machine model $\mathcal{M}'$. Both computation cost $w'$ and communication cost $c'$ are created by using the estimated performance and communication cost and applying them as a factor to $w$ and $c$ on a per-node and per-edge basis, respectively.

**Input:** Task graph $G = (V, C, w, c, t)$, machine model $\mathcal{M} = (\mathcal{A}, \mathcal{T}, \mathcal{P})$, machine model $\mathcal{M}' = (\mathcal{A}', \mathcal{T}', \mathcal{P}')$
**Result:** Task graph $G' = (V, C, w', c', t)$
**foreach** $v \in V, p \in P, p' \in P' \in C'$ **do** 1
  $w'(v, p') \leftarrow \mathsf{estimate}(v, p')$ 2
**end** 3
**foreach** $e \in E, l_a, l_b \in L, l'_a, l'_b \in L'$ **do** 4
  $c'(e, l_a, l_b) \leftarrow c(e, l'_a, l'_b) \cdot \frac{\min_{l \in l_a, l_b} \mathcal{P}_b^L(l)}{\min_{l' \in l'_a, l'_b} \mathcal{P}'^L_b(l')}$ 5
**end** 6
**return** $G'$ = (V,E,w',c',t) 7

Algorithm 5.1: Algorithm to create a new task graph from an original and an optimized machine model

Algorithm 5.1 shows the process of creating a derived task graph $G'$ form an input task graph $G$ for a given machine model $\mathcal{M}$ and another machine model $\mathcal{M}'$ with some optimizations applied. In lines 1–3 the algorithm iterates over all combinations of nodes $v$ from task graph $G$, all existing/unoptimized The function $\mathsf{estimate}(v, p)$ takes a node $v$ of a task graph and a PE $p$ and generates an estimate for the cost of the task $v$ on $p$. This is a placeholder for suitable estimation methods like the ones introduced by Wang et al. [140] or vendor-supplied software. The computation cost $w$ is scaled to match the estimator's cost. The communication cost $c$ is scaled according to the bandwidth property $\mathcal{P}_b^L$ in line 5, so that a lower or higher bandwidth is reflected accordingly in the task graph $G'$.

With the process of introducing potential changes to a machine model and creating adjusted task graphs stated, we can now continue with the DSE below, i.e. with automatic evaluation and discovery of optimized machine models and thus schedules.

## 5.3 Optimization of Machine Models with GA

GAs in general were introduced in Section 2.4 and in this section we describe our adoption to meet our needs. There exist some approaches that directly optimize schedules (e.g. [144]–[146]), but we aim to optimize *the underlying machine model* with GA. The section describes our application of GA to the problem at hand. It discusses choices and trade-offs for the algorithm.

The search space for optimal machine models grows quickly compared to the size of the input. Consider a machine model with 4 distinct PEs. Just the combinatorical possibilities of assigning the PEs to 1 or 2 configurations results in $2^4 = 16$ combinations. If we also take IntraCO into account and assume just one parameter that can have a range of 4 different values per PE, then $4^4 \cdot 16 = 4096$ combinations exist, far too many to synthesize. Thus, we want to quickly analyze the available solutions and reduce the number of them to consider. In general this process is called Design Space Exploration (DSE). One of the more commonly used algorithms for DSE is the GA. However, many techniques exist to perform DSE in the context of FPGAs and HLS [147]. Since the compilation times are so long, it is especially rewarding to prune the search space with DSE. In fact, the goals of the approach to optimize HLS programs by Liu et al. [147] are very

similar to our approach as described below. However, our approach differs from existing ones in two ways:

1. We do not require a costly synthesis and learning phase – possibly to the disadvantage of the result and in favor of speed and applicability.
2. We view the HLS program (abstracted as machine models) in the context of task graphs.

In the following the peculiarities of applying the GA to the problem of optimizing machine models are highlighted. We determine the following parameters of the GA below:

- Two options for an encoding
  - Simple encoding
  - Duplicate encoding
- A fitness function based on scheduling metrics
- Operators for the mutation phase

### 5.3.1 Encoding of Machine Models as Chromosomes

Since the goal of the optimization is to find a near optimal machine model, these models must be encoded into the chromosome. The following assumptions can be made in this approach:

1. The number of locations is known and static.
2. A PE can be moved between configurations (inter configuration optimization), possibly with influence on the cost of executing tasks.
3. A machine model can be fully described by $\mathcal{M} = (C, L, \mathcal{P})$.
4. One or more task graphs are known as an input.

Since the machine model can be described fully by the triple $\mathcal{M} = (C, L, \mathcal{P})$, it is sufficient to encode only this information in the chromosome. Since the locations $L$ are static and known, there is no need to encode them explicitly. Further, if it is assumed that the properties $\mathcal{P}$ do not change (apart from renaming/reassigning), it does also not have to be encoded.

#### 5.3.1.1 Simple Encoding

The remaining information $C$ can be encoded easily. Recall that $C$ is a set of configurations $c_0, c_1, \ldots, c_n$ each consisting of a disjunctive set of PEs $p_{0,0}, p_{0,1}, \ldots, p_{m,n}$ where $p_{i,j}$ is PE $i$ in configuration $j$. This leads to a simple value representation. Each chromosome has $|P|$ genes. The values of all genes are integers in the range $[0, |C|)$. If the value of gene $x$ is set to $y$, then PE $p_{x,y} \in c_y$.

Figure 5.4 shows a chromosome of a machine model using this encoding. It consists of two configurations and a total of five PEs on the left. The chromosome is shown on the right with each gene representing a PE. The first gene from the left represents PE 0 and so on. The value of the gene (0 or 1) denotes in which configuration the gene is contained. For example, the third gene has the value 1, so the PE 2 is contained in configuration 1.

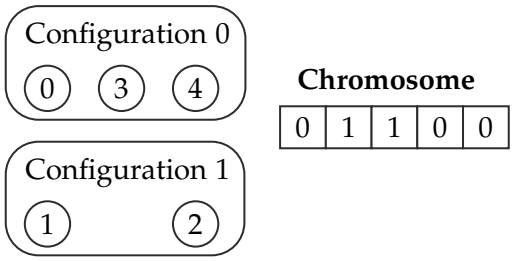



Figure 5.4: Simple encoding with two configurations and five PEs

#### 5.3.1.2 Duplication Encoding

The simple encoding has the drawback that it only allows the assignment of PEs to configurations. It may also be beneficial to create clones of PEs for both IntraCO and InterCO. If schedules are limited by the amount of achievable parallelism, increasing it by cloning PEs should have a positive influence on the schedule. Also, task types that are common but do not dominate the runtime can be cloned to avoid reconfigurations.

The duplication encoding allows for a PE to be cloned either into the same configuration as the original or into a different one. The chromosome consists of $k \cdot |P|$ genes, where $k$ is the *cloning factor*. This allows each PE to have at most $k - 1$ clones. The genes can have an integer value in the range $[-1, |C| - 1)$, where a value of $-1$ encodes the absence of a clone.

The disadvantage of this approach is that the search space is $|C|^{k*|P|}$ instead of $|C|^{|P|}$.

Figure 5.5 shows the duplication encoding with $k = 2$ and two configurations. The genes are arranged in the chromosome such that gene $x$ (gray background) represents the original gene and $x + 1$ represents the clone. PE 2 is cloned in configuration 1 so that it exists twice, while PE 4 is present both in configuration 0 and 1. All other PEs are not cloned, since the values at index $x + 1$ are $-1$.

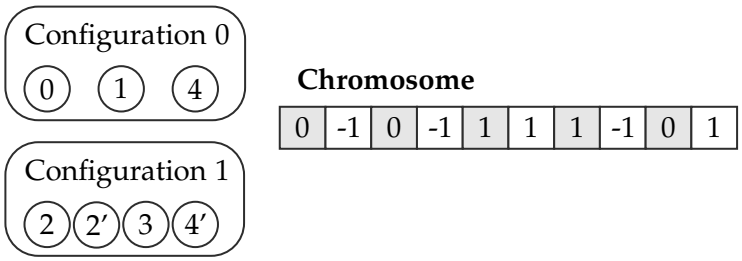



Figure 5.5: Example of duplication encoding with two configurations, five PEs and two clones

### 5.3.2 Fitness Functions

The encoding allows us to formulate a solution – in our case a machine model – in a simple numeric fashion, enabling the GA to manipulate it with simple operations. The other parameter the GA requires to operate is an objective function or fitness function in the context of GAs. The fitness function takes a solution – a machine model – as input and maps it to a real number reflecting the solution's optimality (or lack thereof). We propose to use metrics of schedules as the core of the fitness function.

The machine model by itself does not provide much information about the fitness of the resulting schedules. Therefore, we resort to deriving metrics from the schedules generated by

the machine model, since we are ultimately interested in the optimization of schedules. As established in Chapter 3, the problem of scheduling on heterogeneous hardware is NP-hard and generating an optimal schedule as part of the fitness function is not feasible.

In contrast, the complexity of generating a schedule with a heuristic like REFT from Chapter 4 is polynomial. This makes it a good algorithm to generate a schedule as part of the fitness function. Metrics of the schedule can then be used to compute a single number representing the fitness. We established in Section 2.2 that the schedule length sl is the primary optimization goal in this work. Hence, we use it as part of the fitness function. Further metrics are presented in Chapter 6.

For a given schedule metric $m(\mathcal{S})$, for example the schedule length $\mathsf{sl}(\mathcal{S})$, a schedule $\mathcal{S}_c$ for the machine model represented by the chromosome $c$, and $m_{\max}$ the maximum $m(\mathcal{S})$ of the whole population, the fitness function $f$ is:

$$f(c) = m_{\max} - m(\mathcal{S}_c). \tag{5.1}$$

The fitness function has a direct influence on the probability for the selection of a chromosome for reproduction. A chromosome $c$ with a high value of $f(c)$ (and thus a lower $m(\mathcal{S})$) will be reproduced more often.

### 5.3.3 Operators of GA

To create a new population (of solutions) from an existing one, three steps happen in each generation (iteration) in the GA:

1. *Selection*: which chromosomes should be considered for the next generation?
2. *Crossover*: how are existing parent chromosomes combined into new ones, the offspring?
3. *Mutation*: what changes are applied to genes?

#### 5.3.3.1 Selection

The selection of chromosomes for reproduction is dependent of its fitness. For our work we use the *rank selection* technique. All chromosomes of a population of $n$ are ranked by their fitness from 1 (worst) to $n$ (best). The random process to select parents for reproduction then selects by rank, i.e. rank $k$ has a chance of $\frac{2 \cdot k}{n \cdot (n+1)}$ to get selected.

The *roulette wheel* selection method was dismissed, because it removed low fitness chromosomes at a too high rate. In this method the probability of being selected is *directly* proportional to the normalized fitness.

#### 5.3.3.2 Crossover

For both the simple and the duplication encoding a *single point crossover* function is used: a random point in the chromosome is selected and all genes with an index higher than that will be swapped between the two chromosomes taking part in the crossover.

#### 5.3.3.3 Mutation

The mutation step changes the value of each gene with the probability $p_m$ to a random value in its possible value range. Note that $p_m$ is typically chosen small, e.g. $p_m = 1$ is what we used for all experiments.

| Chromosome | Configurations | Reconfigurations |
|---|---|---|
| [3,2,3,2,3] | 2 | 1 |
| [0,1,2,3,4] | 5 | 4 |
| [2,2,2,2,2] | 1 | 0 |

Table 5.1: Minimal number of reconfigurations for two genomes in the simple encoding

With the encoding, a fitness function and operations defined, we continue to describe InterCO, IntraCO and CO, where these are applied.

## 5.4 Inter-Configuration Optimization (InterCO)

Our methods of optimization use the GA as described above. While all three approaches (InterCO, IntraCO and CO) are just applications of the GA, each of them has specifics that are detailed below.

The machine model $\mathcal{M} = (\mathcal{A}, \mathcal{T}, \mathcal{P})$ consists of an accelerator model $\mathcal{A} = (C, L)$, a topology $\mathcal{T} = (N, O, D, b)$ and properties $\mathcal{P}$. We assume that the locations $L$, the topology $\mathcal{T}$ and properties $\mathcal{P}$ are static and given. The GA only optimizes the combination of PEs into configurations that make up $C$. This poses the following challenges which we will discuss further below:

1. How to prevent the GA to converge due to wrong incentives?
2. How can the resources of a machine model be described?
3. How can the resource limits be enforced?

### 5.4.1 Convergence

When the target optimization metric is the schedule length $\mathsf{sl}(\mathcal{S})$ for schedule $\mathcal{S}$, then there are several parts of the machine model that affect the schedule length. Among others:

- *Cost of execution* of each task
- *Order of execution* of the tasks
- *Communication cost* between locations
- *Reconfiguration overhead*

Since InterCO only optimizes the assignment of PEs to configurations, its outcome is mainly affected by the reconfiguration overhead if the properties of the system are not sufficiently defined. Table 5.1 shows the minimal number of reconfigurations for three chromosomes. Recall that each value in the chromosome assigns a PE to a configuration. The table shows the minimal number of reconfigurations under the assumption that the number of locations $|L| = 1$ and each PE must execute at least one task.

If we use the GA directly as described above to perform InterCO, the trivial solution to any input would be to combine as many PEs into one configuration as possible. Even without the reconfiguration overhead, combining all PEs into one configuration can yield shorter schedules, since all PEs are instantiated at all times. Thus, without a more precise definition of the machine model, the GA will converge towards a minimum number of configurations, one. We demonstrated in Section 5.2.1 that the utilization of resources comes at a cost of PE-performance, which is not reflected if we apply the GA naively.

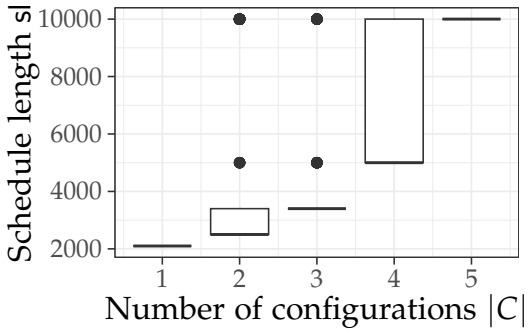



Figure 5.6: The schedule length sl in relation to the number of configurations $|C|$ over 100 generations of GA

An example of this difficulty can be seen in Figure 5.6. It shows the schedule length for all chromosomes of 100 generations during a run of the GA as described above. On the x-axis it shows the number of configurations used by the machine model and on the y-axis the schedule length sl. During the convergence of the GA it tends to set all genes to the same value, because the trivial solution to combine all PEs in one configuration has the optimal cost. This solution makes all PEs available without reconfiguration *and* allows parallel execution of $|P|$ tasks. For example, the chromosome $[0, 0, 0, 0, 0]$ has a minimal schedule length, since all five PEs are contained in configuration 0.

An FPGA, however, consists only of a limited amount of resources. If one tries to synthesize a configuration with PEs that require more resources than the chip can provide, the process fails. Even if the number of resources is sufficient, it is not necessarily possible to synthesize a group of PEs. The placement of resources is also dependent on the routing between LBs, LUTs, DSPs and so on. Additionally, the maximum achievable frequency for a configuration might go down as the resource usage approaches 100%.

In our work on dynamic task scheduling on FPGAs [34] we showed that the optimization of one PE can have negative effects on other PEs in the same configuration – even if they are not executing in parallel. This can be an effect of the aforementioned routing and frequency issues. Additionally, access to memory buses must be shared among PEs. This adds complexity in the design but can also introduce congestion during the execution.

### 5.4.2 Resource Properties

The machine model must be able to reflect the scarcity of resources when applying InterCO as well as IntraCO. Otherwise, the trivially optimal solution is to combine all PEs into one configuration and apply all optimization technique at the expense of chip resources, but it does not reflect the behavior of the real system.

To model these limitations more accurately during the optimization process, additional properties can be used to describe:

1. The *available resources* per location
2. The *required resources* per PE

As with the properties introduced in Chapter 3, the granularity of describing the system may be arbitrarily fine. For practicality the resources are described directly in terms of FPGA components: the number of LUTs, LBs and DSPs are used. The information about the required resources

| Entity | Property | Identifier | Domain | Default | Example |
|---|---|---|---|---|---|
| Location | LUTs | $\mathcal{P}^{L}_{\text{LUT}}$ | $\mathbb{N}$ | $\infty$ | 512000 |
| | LBs | $\mathcal{P}^{L}_{\text{LB}}$ | $\mathbb{N}$ | $\infty$ | 12400 |
| | DSPs | $\mathcal{P}^{L}_{\text{DSP}}$ | $\mathbb{N}$ | $\infty$ | 10800 |
| PE | LUTs | $\mathcal{P}^{P}_{\text{LUT}}$ | $\mathbb{N}$ | 0 | 14144 |
| | LBs | $\mathcal{P}^{P}_{\text{LB}}$ | $\mathbb{N}$ | 0 | 8392 |
| | DSPs | $\mathcal{P}^{P}_{\text{DSP}}$ | $\mathbb{N}$ | 0 | 32 |
| Configuration | LUTs | $\mathcal{P}^{C}_{\text{LUT}}$ | $\mathbb{N}$ | $\sum_{p \in c} \mathcal{P}^{P}_{\text{LUT}}(p)$ | $\mathcal{P}^{P}_{\text{LUT}}(p_0) + \mathcal{P}^{P}_{\text{LUT}}(p_1)$ |
| | LBs | $\mathcal{P}^{C}_{\text{LB}}$ | $\mathbb{N}$ | $\sum_{p \in c} \mathcal{P}^{P}_{\text{LB}}(p)$ | $\mathcal{P}^{P}_{\text{LB}}(p_0)$ |
| | DSPs | $\mathcal{P}^{C}_{\text{DSP}}$ | $\mathbb{N}$ | $\sum_{p \in c} \mathcal{P}^{P}_{\text{DSP}}(p)$ | $\mathcal{P}^{P}_{\text{DSP}}(p_4) + \mathcal{P}^{P}_{\text{DSP}}(p_3)$ |

Table 5.2: Resource properties to describe resource limits of locations and resource usage of a configuration $c$ and PEs $p_0, \ldots$

can be obtained very early in the synthesis process and the available resources are part of the specification of the FPGA.

Table 5.2 introduces the properties describing the *available* resources for a location and the *required* resource for PEs and configurations. For each entity in the first column, the property describing a certain resource is broken down in the second column. The property uses the same convention for identifiers as introduced in Section 3.3. The domain column shows the underlying image of values which are the natural numbers in all presented cases. The default value will be used when no other value is set. The last column shows an example of a value for each property. For example, the number of DSPs used per PE is identified with the property $\mathcal{P}^{P}_{\text{DSP}}$. If the property is not specified, 0 is assumed. Notable are the default values for the resource usage of the configuration: they are the sum over the same property of all PEs inside a configuration. The list of properties can be extended, if more information about the system is available.

### 5.4.3 Enforcing Resource Properties

In contrast to the constraint for PEs in Table 3.4, the properties above do not need to be translated into constraints on the schedule. The resource properties limit the design space of the machine model. The effect of the resource limits on the performance is encoded in the cost function $w$ of the task graph $G = (V, E, w, c, t)$. Hence, the resource properties have an effect on the fitness of a model in the context of a GA. We propose two methods to enforce these effects and limits described by the resource properties:

1. *Exact cutoff*: enforces that only as many resources as locations provide are used.
2. *Linear cutoff*: takes into account the performance effects of a highly occupied FPGA.

#### 5.4.3.1 Exact Cutoff

The exact cutoff method ensures that the amount of necessary resources within a configuration do not exceed the available resources of a location. It is clear that a valid machine model must fulfill the following three constraints:

$$\forall c \in C, \forall l \in L\colon \mathcal{P}^L_{\text{LUT}}(l) \leq \mathcal{P}^C_{\text{LUT}}(c), \tag{5.2}$$
$$\forall c \in C, \forall l \in L\colon \mathcal{P}^L_{\text{LB}}(l) \leq \mathcal{P}^C_{\text{LB}}(c) \text{ and} \tag{5.3}$$
$$\forall c \in C, \forall l \in L\colon \mathcal{P}^L_{\text{DSP}}(l) \leq \mathcal{P}^C_{\text{DSP}}(c). \tag{5.4}$$

Providing the GA with appropriate values for the resource properties removes the trivial solution to combine all PEs into one configuration from the search space. This strategy instead gives the GA an incentive to search for solutions that can be synthesized but still perform well.

The GA does not have a mechanism to invalidate solutions (e.g. not fulfilling the above constraints) per se. Instead, the integration of the resource properties into the GA is done via a penalty in the fitness function. If the initial population or any of the offspring after a crossover or mutation phase emit an invalid machine model, a penalty is subtracted from the fitness of that chromosome. The adapted fitness function becomes:

$$f_{\text{cutoff}}(c) = \begin{cases} m_{\max} - m(\mathcal{S}_c) & \text{if } \mathcal{M} \text{ is valid} \\ 0 & \text{otherwise.} \end{cases} \tag{5.5}$$

The fitness function will behave as before for all valid machine models but will assume the worst case for all invalid machine models. This may lead to a convergence towards valid machine models in the GA, since all chromosomes representing valid machine models are guaranteed to have a higher fitness. An alternative fitness function for InterCO is presented below.

#### 5.4.3.2 Linear Cutoff

The exact cutoff has the disadvantage that it incentivizes machine models that combine as many PEs into each configuration as possible, *just* undercutting the location's resources. These do not account for the increasing computational cost for PEs in configurations with a high resource usage (cf. Section 5.2.1).

Our second approach is to introduce a function that penalizes machine models that have a (too) high resource usage. We call it the resource congestion function. For some location $l$, let $\mathcal{P}^L_R(l)$ be the resource property for resource $R$ and let $\mathcal{P}^P_R(p)$ be the resource property for some PE $p$. The resource congestion is defined by

**Definition 31** (Resource Congestion Function) *The **resource congestion** is a function* $\text{cong}_R(c, l) = h\left(\frac{\mathcal{P}^C_x(c)}{\mathcal{P}^L_x(l)}\right)$ *where $R$ is a resource, $c$ is a configuration, $l$ is a location and $h$ is a function $h\colon [0, \infty) \mapsto [0, 1]$:*

$$h(x) = \begin{cases} 0 & \text{if } x \leq a, \\ \frac{x-a}{b-a} & \text{if } a < x < b, \\ 1 & \text{otherwise.} \end{cases} \tag{5.6}$$

The resource congestion function is used as an indicator for the effect of the resource utilization on the performance of the PEs. The function $h$ is used to map the ratio of available and used resources to a value between 0 and 1 inclusive. Using the ratio as an indicator directly would not accurately reflect the performance for lower utilization.

Besides the fraction of allocated resources $x$, the function $h$ has two parameters $a, b \in [0,1]$, which specify the range $(a,b)$ over which the congestion function increases from 0 to 1. These parameters are dependent on the hardware and software being modeled.

The parameters $a$ and $b$ can be obtained by measuring the relative performance of a benchmarking PE (or a set thereof). Suppose PE $p$ has performance 1 if no other PE is contained in the same configuration. After introducing a second (third, fourth, ...) PE and measuring the performance of PE $p$ for these scenarios as data points $(x_i, y_i)$ with $i = 1, \ldots ...n$, where $x_i$ is the resource usage of the $i$-th measurement and $y_i \leq 1$ the relative performance. The parameters for a function $h$ can be obtained for example by a numerical least squares method, minimizing the sum:

$$\sum_{i=1}^{n} \left(y_i - h(x_i, a, b)\right)^2 . \tag{5.7}$$

We found $a = 0.5$ and $b = 0.9$ to yield results reflecting the behavior of an Intel Stratix 10 DX accelerator. However, we want to stress that this approach – and even more so using the exact cutoff – is not an accurate approximation for reasons described in Section 5.2.1.

Putting everything together, the fitness function $f$ becomes:

$$f_{\text{linear}}(c) = m_{\max} - \left(1 - \max_{l \in L, r \in R} \text{cong}_r(c,l)\right) \cdot m(\mathcal{S}_c) \tag{5.8}$$

where $R$ is the set of resources to consider, e.g. LUTs and DSPs. The linear fitness function reduces the fitness of a solution if the congestion is above $a$ and considers it the worst case solution if the congestion is $b$ or above.

With these fitness functions, InterCO can be performed while respecting the resource limits of the hardware. The second approach, IntraCO is not concerned with the migration of PEs between configurations, but with the relative weighting of PEs inside the same configuration.

## 5.5 Intra-Configuration Optimization (IntraCO)

In this section we highlight IntraCO, an approach to automatically balance the resources used by PEs to optimize the scheduling. There are several ways to affect the allocation of resources to PEs within a configuration. Similar to conventional software, users of an HLS environment have an influence on the bitstream that is generated by the compiler.

1. *Explicit* changes to the structure of the algorithm will be reflected in the bitstream.
2. *Semi-explicit* annotations are hints to the compiler how the code should be transformed into a bitstream.
3. *Implicit* factors influence the bitstream generation as well.

Users of FPGAs for program acceleration must have good knowledge of the hardware to write well performing code. Similar to how the understanding of compilers and the structure of CPUs and their interaction with memory enables programmers to write performant code for the specific architecture, HLS programs can be considerably optimized by restructuring the code.

There are many *explicit* techniques to change the code of a functional PE to a better performing, equivalent one. For example loop fission, loop unrolling, explicit pipelining and even some bit width optimizations [148, pp. 39–40]. Additionally, one could change the algorithm to better

reflect the target architecture, for example replacing a quicksort implementation by a sorting network.

Since changes are error-prone and the optimization techniques for FPGAs are similar across devices, some HLS compilers support annotations for simpler application of the approaches. These *semi-explicit* changes can be parameterized. For example, for a loop unroll annotation, the unroll factor can be a parameter.

Compiler and driver versions, thermal conditions and shared load on the system are *implicit* influences, that are often more difficult to change systematically. For example, it is often not predictable how a version change of the compiler will influence the output bitstream.

We focus on semi-explicit parameters to change the weighting of PEs inside configurations to provide more resources to PEs that have a positive influence on the schedule. This section describes how semi-explicit parameters in conjunction with the GA are used to perform IntraCO.

### 5.5.1 Semi-Explicit Parameters

Semi-explicit parameters have the advantage that they typically do not require large changes to the code, but instead support a limited set of options:

- A *toggle* enables or disables a feature or a block of code, hints or forces the compiler to apply an optimization or configures the compiler to make assumptions about the HLS code.
- A *number range* that configures parameters like buffer- or block size, work group sizes or unrolling factors.

Annotations and their parameters are easy to insert into the code, but anticipating the resulting interactions with the application can be challenging. Similar to inter-configuration optimization, congestion can have an effect on the configurations and the performance of the PEs. We propose a similar approach to performing InterCO for IntraCO: use the GA as described above. The procedure differs for IntraCOs in two aspects:

- *Search space*: instead of optimizing the allocation of PEs into multiple configurations, IntraCOs only consider one configuration.
- *Encoding*: the representation of the machine models must be adapted to the problem. Before, a chromosome encoded a mapping of PEs to configurations. With intra-configuration optimization we take a different approach and let the GA change the (semi-explicit) parameters of the PEs.

### 5.5.2 Definition of the Search Space

The search space is spanned by the parameters of the semi-explicit optimizations. For example, if a PE contains the annotation `#pragma unroll(n)`, one dimension represents values for `n` from 1 to some upper limit. Unrolling a loop further could decrease the runtime of tasks for that PE, but in turn will consume more resources. Additionally, if executions on this PE are not (or only a small) part of the critical path of the application, the effect of the schedule will be small, non-existent or even negative in turn.

The parameters can be identified automatically by lexical or semantic analysis of the input program or specified explicitly, for example by using annotations or macros. The optimizer has to have some knowledge about the semantic of the macros for two reasons:

1. It needs to set the *range of valid values*. If the parameter is a toggle, it only has two valid values – on or off. In the case of a number range, it must know a valid and preferably sensible set of numbers. For example, a SIMD-width is often specified as a low power-of-two. If such parameter is set to 4 in the program code, it is unreasonable to expect a value of 4223 to perform well.
2. The *scope* of the parameters is important. A loop unroll factor only has an effect on the annotated loop, whereas a different loop might only be affected indirectly. If a parameter `BLOCK_SIZE` is set for a file and it is used across all PEs (functions in the code), then the optimizer must assume that the value must be consistent across all PEs.

Parameters can be classified into three categories: PE-level, configuration-level and machine-level.

If a parameter is used in the scope of a function (compiled to a PE), the optimizer can classify it as a PE-level parameter. If all PEs inside a configuration use the parameter, it must be at least classified as configuration-level. If all PEs make use of a parameter, it is a machine-level parameter. The level of the parameter has direct influence on how it is represented in the chromosome.

A machine-level parameter only needs to be encoded once on each chromosome. If it was encoded for each PE, inconsistent values could lead to invalid code. For example, if the block size of a sending PE is different from the block size of the receiving one, the program might not terminate. A configuration-level parameter needs to be encoded once for each possible configuration (which was set to the number of PEs in Section 5.4). Conversely, PE-level parameters must be encoded for each PE separately. An automated analysis must be conservative: changing a parameter on finer granularity can compromise the program's correctness.

Again, we resort to properties to describe additional information about the accelerator model, such as the parameters. Suppose the parameter $n$ of PE $p$ is to be optimized with IntraCO. We set $\mathcal{P}_n^P(p) = k$ to its input value $k$. The key difference between InterCO and IntraCO is that for machine model $\mathcal{M} = (\mathcal{A}, \mathcal{T}, \mathcal{P})$, accelerator model $\mathcal{A}$ and topology $\mathcal{T}$ are fixed, whereas the properties $\mathcal{P}$ is optimized.

### 5.5.3 Encoding for IntraCO

We propose the following encoding for IntraCO:

- For each configuration-level parameter $\mathcal{P}_a^C(c)$ of configuration $c$, introduce a gene $g_a^c$. If the parameter is a toggle, then it is binary encoded. If it is a number range, it is encoded by value.
- For each PE-level parameter $\mathcal{P}_a^P(p)$ of PE $p$, introduce a gene $g_a^p$.

The order of the genes inside the chromosome is not important. It needs to be consistent across chromosomes and must be able to construct a valid configuration/machine model from a chromosome, i.e. given a gene and its value, the parameter must be set correctly.

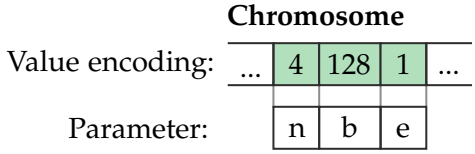
Figure 5.7 shows an example for the value and binary encodings for an extract of an OpenCL HLS code. The HLS code shown on top in Figure 5.7 (a) depicts an extract from an OpenCL file containing a kernel function `do_process` with three parameters:

- `b`: a block size in line 1
- `e`: a toggle to disable the expression balance in line 2
- `n`: an unroll factor in line 5

The value encoding on the bottom in Figure 5.7 (b) shows each gene directly corresponding to a value for each parameter. For example, `n` has the value 4 and the block size `b` is set to 128 in this chromosome. In contrast, each possible value in the binary encoding is either 0 or 1. If more than one gene per parameter is set to 1, the PE would be synthesized more than once.

```
1 #define BLOCK_SIZE b
2 #pragma HLS expression_balance e
3
4 __kernel void do_process() {
5 #pragma unroll(n)
6     for(size_t i = 0; i < BLOCK_SIZE; i++) {
7         // ...
8     }
9 }
```

(a) OpenCL HLS code




(b) Parameters and encodings

Figure 5.7: An excerpt of OpenCL code and value and binary encodings for IntraCO

A chromosome can encode a machine models incapable of executing the given task graph(s) for several reasons:

- *High resource usage*: similar to InterCO, it might be impossible to synthesize a configuration due to limited resources.
- *Constraints*: if the properties of a PE prohibit combinations with other PEs, this must be reflected in valid machine models.
- *Types*: the machine model must contain at least one PE for each type of task in the task graph.

An invalid machine model is penalized by the fitness function as defined in Section 5.4.3, so that the GA will converge to a valid one if it exists.

## 5.6 Configuration Optimization (CO)

To find a schedule as near to the global optimum as possible, both InterCO and IntraCO are combined. This means that both, the assignment of PEs to configurations and their parameters are to be optimized. We propose a combination of the value encoding. For all levels of parameters.

A chromosome consists of two parts:

1. A set of genes for the assignment of PEs to configurations representing InterCO
2. A set of genes for machine-, configuration- and PE-level parameters representing IntraCO

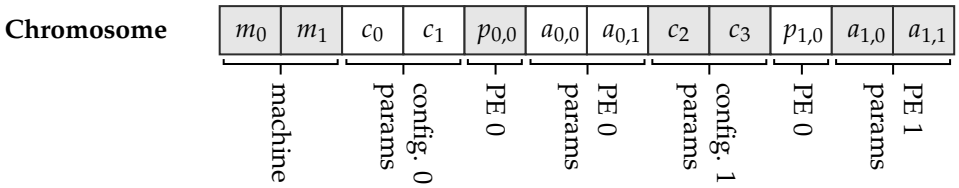



Figure 5.8: A chromosome encoded for CO with two machine parameters, two configurations and one PE

Figure 5.8 shows the encoding a machine with two machine parameters ($m_0$ and $m_1$), two configurations with two parameters each ($c_0, \ldots, c_3$) and one PE ($p_0$). The PE can be cloned into each of the configurations and within each configuration it may have a unique set of parameters ($a_{0,1}, \ldots, a_{1,1}$). The semantic is as follows:

- The machine-, configuration and PE-level parameters use the same encoding as IntraCO as described above in Section 5.5.1.
- The genes $p_{x,y}$ either have the value 0 when the PE is not included in the configuration (and the parameters $a_{y,z}$ have no effect) or for values $k \geq 1$ a number of $k$ clones of PE $y$ is inserted into configuration $x$.

This encoding allows independent configuration of all parameters except for the clones of PEs in the same configuration: these will share the same set of values for all parameters.

All three approaches are evaluated in Chapter 6.

This chapter introduced three methods of optimizing existing machine models with the help of GA. Inter-Configuration Optimization is concerned with the distribution of PEs into configurations, Intra-Configuration Optimization is concerned with the distribution of resources within a configuration to different PEs and Configuration Optimization (CO) combines both applications. Special attention was paid to the problem of static congestion and its effects on the schedule.

Chapter 6 is concerned with the evaluation of the previous three chapters, Chapters 3, 4 and 5. We describe our methodology for the evaluation, the software that was used and developed, present our results and discuss them briefly.

# Chapter 6

# Evaluation

In this chapter we evaluate our work presented in Chapters 3, 4 and 5. Evaluating scheduling algorithms and by extension machine models is not an easy task, since it heavily depends on the input data. Our strategy to perform an accurate and reproducible evaluation is the following:

1. Establish that we can accurately simulate task graphs for a machine model. This is done by executing task graphs on existing hardware and comparing the resulting schedules to our simulations.
2. Compare our polynomial time scheduling algorithm to optimal schedules to assess REFT's results and the effects of input sizes.
3. Show the effect of machine models optimized through DSE by comparing scheduling and machine model metrics prior to and after the optimization.

This strategy is manifested in the structure of this chapter as outlined below. Before the evaluation and its results are reported, the software framework we used to perform it is described and metrics are introduced. The chapter highlights several results of the evaluation, aggregated results are prevented in Appendix B and raw measurements can be found in the corresponding repository. The rest of this chapter is outlined as follows:

1. The *description of RESCH* outlines our software framework that was used for this evaluation.
2. The *definition of metrics* introduces more metrics that can be used to assess the quality of machine models and schedules.
3. The *evaluation of the machine models* is considered with the capability of the theoretical framework to model the underlying HLS workflow and the hardware as presented in Chapter 3.
4. The *evaluation of the polynomial time scheduling algorithm*, REFT, is directly considered with the quality of the emitted schedules as presented in Chapter 4.
5. The *evaluation of the DSE process* is considered with the output machine models created by the GA as presented in Chapter 5.

## 6.1 The RESCH Framework

The REconfigurable SCHeduling (RESCH) framework is a collection of software modules to generate, execute and analyze schedules on reconfigurable hardware such as FPGA-based accelerators. It implements the concepts and algorithms presented in Chapters 3, 4 and 5 and extends them for the evaluation. RESCH consists of the following modules, each of which is described more thoroughly below:

1. The *graph generator module* is used to create random or predefined task graphs.

2. The *trace importer module* can be used to automatically extract task graphs from existing applications without interference.
3. The *hardware description module* can be used to define machine models as described in Chapter 3.
4. The *simulator module* consists of two submodules:
   a) The *optimal solver* generates optimal schedules using a constraint programming technique.
   b) The *heuristic scheduler* produces schedules for a given machine model and task graph in polynomial time by using the REFT algorithm.
5. The *OpenCL executor module* derives an OpenCL program from a machine model and executes an arbitrary task graph on it.
6. The *DSE module* applies the GA to generate recommendations for HLS code.
7. The *analysis module* receives a set of schedules or machine models and performs statistical analysis for the evaluation.

The software workflow reproduces the methodology as shown in Figure 3.1 on page 31. The inputs consist of task graphs, either generated from the graph generator or observed by the trace importer, and the machine model as described by the Python hardware description. The inputs are used to generate schedules, either via simulation or via execution on an OpenCL platform. The generated schedules can either be used for analysis using the metrics module or fed into the optimizer module for optimization of the input machine model.

### 6.1.1 Graph Generator Module

The graph generator module is one of the two possible ways to inputs graphs, the other being task graphs based on OpenCL traces. The module is extensible and parameterizable for several key variables. It supports structured and random generation of task graphs:

1. Structured task graphs emulate existing programs with well known execution patterns like the Cholesky decomposition [149] and the LU factorization [150].
2. Random task graphs are generated according to some schema with their structure and parameters based on a stochastic process. We implement three schemes:
   a) layer-by-layer technique [151]
   b) Erdős–Rényi technique [152]
   c) uniformly random

The graph generator module is implemented in Python and uses the Boost Graph Library [153] to interact with graph data structures.

#### 6.1.1.1 Structured Graphs

Structured graphs emulate well-known algorithms. The RESCH framework supports the creation of graphs emulating the LU factorization and the Cholesky decomposition [154, pp. 915–920].

The LU factorization decomposes a matrix $A$ into a lower triangular $L$ and an upper triangular matrix $U$ such that $A = LU$. Task-based implementations work recursively blockwise on matrices [154, pp. 819–822] with four different kernel functions. A task graph $G = (V, E, w, c, t)$ representing the LU factorization uses four different types as described by function $t$. We assume

a square $n \times n$ matrix as an input that is split up into blocks of size $b \times b$, thus the first two parameters for the graph generator are $n$ and $b$. An execution time $w(v)$ for all tasks must be specified as well. Due to the blocked construction, it is sufficient to observe the execution time once for each task type for a given block size ($b$). For example, we observed a runtime of 16 ms for all tasks of a block size of 50 on an Intel Stratix 10 DX accelerator card. The third parameter is the execution cost $w$ in ms. The communication cost $c$ can be calculated using the memory bandwidth $B$ as $c = \frac{s \cdot b^2}{B}$, where $s$ is the size of the underlying data type in bytes (e.g. $s = 8$ for `double`).

From a task scheduling perspective, the Cholesky decomposition is similar: it also works blockwise on a (now Hermitian positive-definite) matrix $A$, but produces a lower triangular matrix $L$ such that $A = LL^{\mathsf{T}}$. A task-based formulation of the algorithm also contains 4 types of tasks, similar to the LU factorization. Thus, the parameters for the generator are identical, but it produces a task graph with a different structure.

#### 6.1.1.2 Random Graph

Task scheduling algorithms are often evaluated by executing randomly generated graphs. This prevents bias towards a certain set of problems and may produce numerous graphs with predefined properties. The RESCH framework implements three methods to generate task graphs randomly:

1. The *layer-by-layer technique* partitions tasks into a well-defined number of layers.
2. The *Erdős–Rényi technique* connects vertex $i$ and $j$ with a probability $p$.
3. The *enumeration method* to generate uniformly random graphs by Kuipers et al. [155].

All algorithms have as the first parameter the number $n$ of vertices (tasks) to generate.

The layer-by-layer technique requires two additional parameters, $l$ and $p$ and generates graphs in two steps:

1. It generates $n$ vertices and partitions these into $l \leq n$ layers, i.e. every vertex $v_k$ belongs to a layer $0 < k < l$.
2. It creates a directed edge $(v_k, v_{k+1}) \in E$ with probability $p$.

This technique does not guarantee that the resulting graph consists of $l$ layers, since for each layer $k$ an edge to layer $k+1$ is not created with probability of $(1-p)^{n_k \cdot n_{k+1}}$, where $n_k$ is the number of nodes in the $k$-th partition.

The Erdős-Rényi technique creates an $n \times n$ adjacency matrix $A$. The matrix represents the graph with $n$ vertices. An edge exists from vertex $v_f$ to $v_t$ if the element $A_{f,t}$ in the matrix is 1 – it is 0 otherwise. The entry $A_{i,j}$ with $i < j$ is set to 1 with probability $p$. Thus, an edge exists from $v_i$ to $v_j$ with probability $p$. Acyclicity is guaranteed because $A$ is a lower triangular matrix, since only elements with $i < j$ can be set to 1.

The enumeration method by Kuipers et al. [155] is capable of generating uniformly random graphs. It recursively constructs a DAG by drawing random numbers and reconstructing the associated graph of all enumerated DAGs. While the Erdős–Rényi method produces random tasks as well, they are not uniformly random, i.e. not every DAG has an equal probability of being generated for a given set of parameters.

### 6.1.2 Trace Importer Module

A challenge in the quantitative analysis of scheduling algorithms is the availability of suitable task graphs. As detailed above, the graph generator module is able to generate structured and random graphs for further examination. However, especially the structured graphs must resemble the workload accurately. Obtaining accurate task graphs is not trivial. Some data sets exist, but none capture the unique characteristics of FPGAs [151]. Even if such data sets existed, the heterogeneity of the field of FPGA makes it difficult to use task graphs obtained from one device and draw conclusions for the properties of others as we observed in Section 5.2.1.

Therefore, the RESCH framework includes a C++ trace importer module. It is based on the Intel OpenCL Intercept Layer[1] (ICL) and extends ICL to emit machine-readable logs that contain information to create task graphs using Algorithm 3.2 on page 57. To obtain a trace from an OpenCL program (where the kernels may execute on an FPGA-based accelerator), the program must be called as an argument to the `resch_trace` binary. For example, calling the Hello World example code of the Xilinx Vitis Accel examples[2], one must call `resch_trace ↪ ./hello_world configuration.xclbin`. The trace importer will intercept the OpenCL calls and log relevant information to a file.

The module gathers dependencies of tasks by observing OpenCL events (`cl_event`) instances. With each task a user can specify an *event wait list* of events that must be completed before its execution may start. Also, each task creates an event that other tasks may use in their respective wait list and so on. The exact timings of the execution are obtained by the profiling feature of OpenCL. Intercepting the creation of OpenCL command queues (`cl_command_queue`) lets us pass the parameter `CL_QUEUE_PROFILING_ENABLE`. After the program finishes its execution, the trace importer can read the timings using the `clGetEventProfilingInfo` function per task invocation. Data movement is tracked by intercepting (and logging) calls that allocate and/or transfer memory objects to and from the device (the FPGA-based accelerator), like `clEnqueueWriteBuffer`. These allow to track the data size and dependencies with event waiting list, similar to the task exeuction tracing.

A caveat of this method is that some OpenCL programs are not formulated according to the underlying task dependencies, but use `clFinish` to wait for all enqueued tasks and data movements to succeed before a new task is enqueued and executed. In this case the RESCH framework cannot reconstruct a task graph, but only a series of task enqueues and `clFinish` calls. Similarly, it is possible to read arguments of the kernels to assess which memory objects (buffers) they have read or write access to, but it is not possible to determine the actual amount of data read or written.

In the evaluation the module was used to obtain accurate parameters for the structured graph generator algorithms by executing OpenCL implementations on Xilinx Alveo U280 accelerators and tracing it with `resch_trace`.

### 6.1.3 Hardware Description Module

Machine models are a central part of this work and it must be possible to formulate them with the RESCH framework. That is the purpose of the hardware description module. It contains definitions and convenience functions for the constructs introduced in Chapter 3 for machine models, accelerator models, topologies as well as task graphs and schedules.

[1] https://github.com/intel/opencl-intercept-layer
[2] https://github.com/Xilinx/Vitis_Accel_Examples

```
1 locations = [Location(0), Location(1)]
2 config = Configuration(0, locations)
3 PEs = [PE(i, config) for i in range(num_PEs)]
4 acc = Accelerator(PEs)
5 topo = Topology.default_from_accelerator(acc)
6 machine_model = Machine(acc, topo))
```

Listing 6.1: Definition of a machine model using the hardware description module

Listing 6.1 shows how a machine model $\mathcal{M} = (\mathcal{A}, \mathcal{T}, \mathcal{P})$ (`machine_model`) is constructed in a bottom-up fashion. In line 1, the locations $L$ (`locations`) are declared. A configuration is created and passed the list of locations, which also sets $\mathcal{P}_l^C(\texttt{config})$. Then a list of `num_PEs` PEs is created, each of which is assigned to the configuration `config`. The PEs are passed to the constructor of the accelerator model. The default topology (see Section 3.2.2) is created from the accelerator model and passed to the machine model, together with the accelerator model. Not used here is a third, optional parameter `properties`: an dictionary of properties, that maps PEs, locations, configurations, and vertices and edges of the topology graph to properties. For example, the code `properties[pe]["s"]` returns the property $\mathcal{P}_s^P(\texttt{pe})$. The first parameter for `Location`, `Configuration` and `PE` is its index – it must be unique per class and machine model.

### 6.1.4 Simulator Module

The simulator module is used to for the scheduling of task graphs on arbitrary machine models. Its input are either task graphs of traced applications or task graphs from the graph generator module. It consists of two submodules:

1. The *optimal solver* to generate the best solution
2. The *heuristic scheduler* relying on heuristics

#### 6.1.4.1 Optimal Solver

The implementation makes use of constraint programming by formulating the constraints from Section 3.4 as inputs for the OR-Tools constraint programming solver [129]. It implements the constraint derivation rules to translate properties of the machine model to constraints for the solver. Input to the solver is a set of variables, in our case the values for the start time $t_s$, the finish time $t_f$ and the resource mapping proc for all tasks. It attempts to find values for these variables. Of course, the goal is not to find arbitrary values, but such values that represent a valid schedule for the given machine model. Therefore, the set of constraints is extended depending on the communication setting (see Table 3.7), the machine model and the task graph.

The output of the simulator module is a list of valid schedules including the optimal one (for a specific hardware model). The variable to be optimized is the maximum finish time, i.e. the makespan of the schedule.

One drawback of the constraint solver is its runtime. It can generate a valid schedule quickly, but the search for the optimal schedule can be prohibitively long: while an optimal schedule can be found within 30 ms for a task graph with 10 tasks, it takes over 180 s for a task graph with 30 tasks, everything else being equal. Thus, the implementation should not be used in practice to find optimal schedules, but it serves as a provider for a *ground truth* for optimal schedules, especially as a baseline for REFT.

#### 6.1.4.2 Heuristic Scheduler

The simulator contains an implementation of REFT. It supports all task graphs and machine models as the solver. It also supports edge scheduling with the congestion-aware communication setting.

The implementation is not particularly optimized for runtime, but performs significantly better than the solver. Especially larger task graphs can be computed in time frames that make the implementation usable in practice. For example, it generates a schedule for the same random graph with 30 tasks in 70 ms, where the constraint solver module requires 180 s.

### 6.1.5 OpenCL Executor Module

The OpenCL executor module's purpose assures accuracy of the simulated schedules. One major challenge of the simulation of schedules is to ensure that the simulated schedules reflect the behavior of the modeled hardware. In our case, the hardware is modeled with machine models. We do not want to rely on low-level, vendor-specific features and therefore resort to the OpenCL programming model to meet this challenge.

#### 6.1.5.1 Inputs

The OpenCL executor module is implemented in C++. To interact with the rest of the Python-based components, it takes four input files which are described further below:

1. A *machine model description* in JavaScript Object Notation (JSON) [156] format
2. A *schedule description* in JSON
3. A *bitstream* for each configuration defined in the machine model
4. A *task graph* to execute in GraphML format [157]

**Machine Model Description** The machine model definition only needs to contain structural information, such as the configurations and their PEs. However, it does not require any properties, since these are inherent to the hardware. Listing 6.2 shows an example of such a definition: a configuration with `id` 0 contains two PEs with ids 0 and 1. The implementation will look for a configuration file with the filename `basic` and – depending on the underlying hardware – will append an appropriate file extension like `.xclbin` for Xilinx- or `.aocx` for Intel-based FPGAs. Since all PEs must be implemented by an OpenCL `kernel` function, instead of specifying a PE's type, its corresponding kernel function name is used in the `function_name` key. When a task is scheduled on the PE, the function with this name is called.

```
1 {
2    "configurations": [ "id": 0, "file_name": "basic", "PEs": [
3       { "id": 0, "function_name": "fmmult" },
4       { "id": 1, "function_name": "check" }
5    ]
6 }
```

Listing 6.2: An example description of a machine model with one configuration and two PEs

A corresponding JSON Schema [158] definition of the file format can be found in Listing A.1.

**Schedule Description** The schedule $\mathcal{S}$ that is to be executed by the OpenCL executor is similarly described in the JSON format. It consists of an entry for each of the tasks $v \in V$ of the task graph $G = (V, E, w, c, t)$ and holds a unique identifier `id`, the allocation to a PE `PE` ($\text{alloc}(v)$) and the value for `t_s(v)`: `t_s`. An example of a schedule with two tasks is depicted in Listing 6.3.

```
1 {
2   "schedule": [
3     { "id": 0, "PE": 0, "t_s": 0 },
4     { "id": 1, "PE": 1, "t_s": 100 }
5   ]
6 }
```

Listing 6.3: An example schedule with two tasks on two PEs

A JSON Schema definition of the file format can be found in Listing A.2.

**Bitstream** Configurations must be suited to execute the input task graphs. We provide a universal OpenCL implementation that supports up to nine PEs per configuration. These standard PEs are described below in the methodology in Listing 6.4. An OpenCL file containing these PEs is compiled using the HLS compiler from the vendor to create the bitstream(s).

**Task Graph** Task graphs are imported using the standard GraphML [157] file format. The files are read and processed using the Boost Graph Library [153]. Attributes like communication and computation cost can be saved as attributes in the GraphML file. The easiest way to obtain a GraphML task graph is through the task graph generator module.

#### 6.1.5.2 Methodology

To perform schedules accurately, the OpenCL executor must be able to create tasks with the cost denoted in the task graph description. Suppose $G = (V, E, w, c, t)$ is a task graph and $\mathcal{M} = (\mathcal{A}, \mathcal{T}, \mathcal{P})$ a machine model. The GraphML description contains computation cost information $w(v)$ for each task $v \in V$ and communication cost information $c(e)$ for each edge $e \in E$. How the abstract cost value of a task graph is applied depends on the use case, e.g. a cost of $w(v) = 10$ could represent the energy cost in Joule. We use the cost to represent execution (and communication) time in this evaluation for reasons presented in Section 2.2. Therefore, our method to construct tasks with a defined execution (and communication) time is:

1. For each configuration $C$, create an OpenCL file with a matching number of *standard PEs*.
2. Enqueue a standard PE task with work size 1, increase the work size and repeat the process $m$ times.
3. Enqueue a standard PE task with data size 1, increase the data size and repeat the process $n$ times.
4. Perform a linear regression on the observed execution times.

Using this method of standard PE allows us to finely control the execution and communication times. Listing 6.4 shows the (simplified) OpenCL code used to implement it. The complete code can be found in Listing A.3 on page 131. Lines 4–5 load `data_size` integers from the global memory and store them temporarily in a local variable. Lines 5–6 perform an arithmetic operation `cost` times on an arbitrary integer value `val`. In the end – to prevent optimization – a

result value is written to a buffer in global memory. The `data` (input) buffer must be sufficiently large to hold `data_size` elements, while the output `out` buffer must only hold one element.

```
1 kernel void standard_pe(const int cost, global const int *restrict
    ↪ data,
2                 const int data_size, global int *restrict out) {
3   float val = 23;
4   for(int i = 0; i < data_size; i++) // Data loading
5     val = data[i];
6   for (int i = 0; i < cost; i++) // Processing
7     val += i;
8   *out = val;
9 }
```

Listing 6.4: The standard PE in OpenCL

After $m + n$ tasks have been executed, profiling data is used to determine the exact execution times for each given input size. Let $x_i$ be the work size of the $i$-th task with $1 \leq i \leq m$, $\overline{x}$ the arithmetic mean of all $x_i$, $y_i$ the measured execution time for $x_i$ and $\overline{y}$ the arithmetic mean of all $y_i$. Compute the simple linear regression with

$$\beta_1 = \frac{\sum_{i=1}^{m}(x_i - \overline{x})(y_i - \overline{y})}{\sum_{x=1}^{m}(x_i - \overline{x})^2} \tag{6.1}$$

and

$$\beta_0 = \overline{y} - \beta_1 \overline{x}. \tag{6.2}$$

Since $\hat{y}_i = \beta_0 + \beta_1 x_i$ gives the estimate $\hat{y}_i$ for an input size of $x_i$, it can also be formulated as:

$$\hat{y}_i = \beta_0 + \beta_1 x_i \tag{6.3}$$

$$\Rightarrow \hat{x}_i = \frac{y_i - \beta_0}{\beta_1}. \tag{6.4}$$

To create a kernel that executes for a duration of $y_i$, the input parameter must have the value $x_i$. With this method, it is possible to precisely control the execution time of tasks on standard PEs. The procedure is equivalent with data sizes (communication times).

### 6.1.6 DSE Module

The purpose of the DSE module is to apply the CO techniques presented in Chapter 5 to optimize machine models. The module uses the GA to perform the optimization. The input to the GA is a machine model defined using the hardware description module and a task graph. The output is an optimized machine model and a valid schedule for the task graph.

The module supports InterCO, IntraCO and CO and applies modifications to the task graph as described in Section 5.2. The encoding of machine models is performed as detailed in the respective sections in Chapter 5.

### 6.1.7 Analysis Module

The purpose of the analysis module is to perform automatic evaluation of machine models and schedules. It consists of two parts:

1. The benchmark submodule performs automated test series for the results presented in Sections 6.3, 6.4 and 6.5.
2. The metrics submodule calculates metrics for machine models and schedules.

The benchmark submodule uses the infrastructure provided by the RESCH framework and performs automated test series for the evaluation of machine models, REFT and DSE. The metrics module implements all metrics as defined below, so machine models and schedules can be quantitatively evaluated.

## 6.2 Metrics

The metrics used to assess the quality of both schedules and machine models are presented below. So far, we have focused solely on the schedule length as a primary metric for reasons described in Section 2.2. However, there are several more ways to evaluate the quality of a schedule or machine model. These metrics are mainly shown in the results in Appendix B, but some are also highlighted in the evaluation below. The metrics can be split into two categories presented below:

1. Metrics for *schedules*
    a. Schedule length/makespan
    b. Schedule length ratio
    c. Speedup
    d. Slack
2. Metrics for *machine models*
    a. Stability
    b. Mean schedule metric
    c. Efficiency

### 6.2.1 Metrics for Schedules

Schedule metrics assign a single real number to a schedule. A metric $m$ has the form $m\colon \mathcal{S} \mapsto \mathbb{R}$. In this section $\mathcal{S} = (t_s, \mathsf{proc})$ denotes a schedule and $G = (V, E, w, c, t)$ is a task graph.

#### 6.2.1.1 Schedule Length/Makespan

Often used in the literature is the schedule length $\mathsf{sl}(\mathcal{S})$, which is the difference between the earliest node start time and the latest node finish time:

$$\mathsf{sl}(\mathcal{S}) = \max_{v \in V} t_f(v) - \min_{v \in V} t_s(v). \tag{6.5}$$

Since we defined $\min_{v \in V} t_s(v) := 0$, this can be simplified to

$$\mathsf{sl}(\mathcal{S}) = \max_{v \in V} t_f(v). \tag{6.6}$$

Generally low values for the makespan are preferable, especially when energy efficiency is of concern as we stated in Section 2.2.

#### 6.2.1.2 Schedule Length Ratio

The schedule length ratio describes the ratio between the makespan and the sum of the smallest execution cost of the critical path CP.

$$\mathsf{slr}(\mathcal{S}) = \frac{\mathsf{sl}(\mathcal{S})}{\sum_{v \in CP} \min_{p \in P} w(v,p)} \tag{6.7}$$

Because the denominator is a lower bound on the schedule length, lower values are better and a value of $\mathsf{slr}(\mathcal{S}) = 1$ is a lower bound [135].

#### 6.2.1.3 Speedup

The speedup in our case is defined as the ratio between the sequential execution time and the makespan. The sequential execution time is the lowest execution time on any PE without any communication cost. The main difference to the inverse of the schedule length ratio is that the sequential execution time includes all tasks rather than the critical path [135].

$$\mathsf{speedup}(\mathcal{S}) = \frac{\sum_{v \in V} \min_{p \in P} w(v,p)}{\mathsf{sl}(\mathcal{S})} \tag{6.8}$$

#### 6.2.1.4 Slack

The slack is a metric for the variance in the schedule lengths generated by a scheduling algorithm. It describes the robustness of the results of the algorithm given uncertainty in the task processing time [135].

$$\mathsf{slack}(\mathcal{S}) = \frac{\sum_{v \in V} \left(\mathsf{sl}(\mathcal{S}) - \mathsf{rank}_u(v) - \mathsf{rank}_d(v)\right)}{|V|} \tag{6.9}$$

The lower the slack, the nearer the schedule operates on the global optimum for an infinite number of PEs. Intuitively, a machine with no limit on the number of PEs can schedule each task at its data ready time and the numerator approaches 0. Note that minimizing the makespan is not automatically minimizing the slack, since tasks not on the critical path may contribute to higher slack without affecting the makespan.

### 6.2.2 Metrics for Machine Models and Scheduling Algorithms

To assess the fitness of a machine model and a scheduling algorithm – as opposed to a schedule generated for a machine model – we introduce additional metrics. In this section $G^n$ is a set of $n$ task graphs $G = (V, E, w, c, t)$ and $\mathcal{M} = (C, L, \mathcal{P})$ a machine model.

#### 6.2.2.1 Stability

In Section 3.7.4.1, we introduced noise into the cost functions of a task graph and showed its effect on the resulting schedule for two machine models. The stability reduces the effect to a single number. Let $G^p$ be a set of $o \cdot n$ task graphs $G' = (V, E, w', c', t)$ where $w'$ and $c'$ are constructed by adding Gaussian noise with a mean $\mu = 0$ to the computational cost $w(v, p)$ and communication cost $c(e, l_0, l_1)$, respectively. The stability for machine model $\mathcal{M}$ and task graph $G$ is given by:

$$\mathsf{stability}(\mathcal{M}, G^m) = \frac{\sigma}{s_m}, \tag{6.10}$$

where $\sigma$ is the standard deviation of the Gaussian noise and $s_m$ is the sample standard deviation of the metric $m$ for all schedules generated for the task graphs $G^m$ on $\mathcal{M}$. We use the schedule length for $m$.

#### 6.2.2.2 Mean Schedule Metric

The arithmetic mean over all generated schedules for a metric $m$ is:

$$\overline{m}(\mathcal{M}, G^n) = \frac{1}{n} \sum_{G \in G^n} m(\mathcal{S}_G), \tag{6.11}$$

where $\mathcal{S}_G$ is the schedule for $G$. This may be of interest when optimizing machine models for a specific schedule metric. For example, instead of working with a single task graph during DSE, one could work on a set of task graphs and use this metric instead of the underlying schedule metric instead.

#### 6.2.2.3 Efficiency

The efficiency is often described as the ratio of speedup to number of processors [27], [135, p. 222]. However, that does not provide a meaningful metric in reconfigurable computing, since a processor (i.e. a PE) can consume an arbitrary amount of resources. We propose a more specialized notion based on the resources being used. The resource-specific efficiency $\mathsf{efficiency}_r$ for a configuration $c \in C$ is defined as follows:

$$\mathsf{efficiency}_r(\mathcal{S}, c) = \frac{\mathsf{speedup}(\mathcal{S})}{\frac{\sum_{p \in c} \mathcal{P}_r^P(p)}{\mathcal{P}_r^C(c)}} \tag{6.12}$$

The overall efficiency is defined as the mean minimal efficiency over all resources $R$:

$$\mathsf{efficiency}(\mathcal{S}) = \frac{1}{|C|} \sum_{c \in C} \min_{r \in R} \mathsf{efficiency}_r(\mathcal{S}, c) \tag{6.13}$$

With these schedule and machine model metrics defined, we can proceed to the evaluation of machine models, REFT and the DSE.

## 6.3 Evaluation of Machine Models

The machine models as defined in Chapter 3 represent one of our novel contributions to describe reconfigurable hardware for program acceleration. A challenge is to evaluate the accuracy of

the model for the (described) hardware. How well do machine models map to FPGA-based accelerators and their behavior in task scheduling? The RESCH framework, our software implementation, supports the execution of task graphs on FPGA-based accelerators as OpenCL tasks. To assess the accuracy of the models, the same task graphs are scheduled using REFT on the machine model. The schedules from both the execution on hardware and the simulation are then compared. This section consists of three parts:

1. A *classification* of machine models for a quick description of the models under consideration in the evaluation.
2. A systematic *determination of parameters* for standard PEs.
3. An investigation of the effects of *reconfiguration overhead* of a machine model.

### 6.3.1 A Classification for Machine Models

The evaluation makes extensive use of different machine models with a wide range of properties. We introduce a classification to describe the fundamentals of each model with a small set of variables. Especially in the statistical analysis it is neither feasible nor useful to describe each machine model exactly. However, classifying machine models makes it easy to compare results without overwhelming readers with data. The relevant data and logs are still available in the RESCH framework's repository.

Each machine model $\mathcal{M} = (\mathcal{A}, \mathcal{T}, \mathcal{P})$ is classified with the following features[3]:

- The number of locations $|L|$
- The number of configurations $|C|$
- The mean number of PEs per configuration $\overline{|C|} = \frac{\sum_{c \in C} |c|}{|C|}$
- The *heterogeneity* $h(\mathcal{M})$, where

$$h(\mathcal{M}) = \frac{\sum_{c_a, c_b \in C} J_t(c_a, c_b)}{|C|}, \text{ where} \tag{6.14}$$

$$J_t(A, B) = \frac{A_t \cap B_t}{A_t \cup B_t} \text{ is the Jaccard index,} \tag{6.15}$$

$$A_t = \{\mathcal{P}_t^P(p) \mid p \in A\} \text{ and} \tag{6.16}$$

$$B_t = \{\mathcal{P}_t^P(p) \mid p \in B\}. \tag{6.17}$$

The heterogeneity function $h = 0$ if all the PEs have the same type and $h = 1$ if there is no overlap in the types, i.e. each configuration contains no PE with an equal type of any other PE.

Table 6.1 shows three classes that depend on the features. A static machine model makes no use of reconfiguration, i.e. only one configuration is provided. A partially reconfigurable model has one PE per configuration, but contains multiple locations and configurations. Every other constellation is called (dynamically) reconfigurable[4] The heterogeneity is included in this classification, because it provides context on the amount of reconfiguration that must be performed.

[3] We use *features* instead of *properties* to avoid confusion with the machine model's properties $\mathcal{P}$.

[4] We prefer the term *reconfigurable* but sometimes use the terms *dynamically reconfigurable* to distinguish from partially reconfigurable.

| Feature | | | |
|---|---|---|---|
| $\lvert L\rvert$ | $\lvert C\rvert$ | $\overline{\lvert C\rvert}$ | **Class** |
| $\geq 1$ | 1 | - | static |
| 1 | $> 1$ | 1 | reconfigurable |
| $> 1$ | $> 1$ | 1 | partially reconfigurable |
| $> 1$ | $> 1$ | $> 1$ | reconfigurable |

Table 6.1: A set of machine model classes based on their features

We write that a machine model $\mathcal{M} = (\mathcal{A}, \mathcal{T}, \mathcal{P})$ (or a set thereof) is an $(|L|, |C|, \overline{|C|}, h(\mathcal{M}))$-model to give the reader a quick and precise description of its characteristics, e.g. a $(3, 5, 1, 1)$-model is a partially reconfigurable model with 3 locations, 5 configurations and each with 1 PE per configuration, i.e. 5 different in total that can be instantiated at all 3 locations.

### 6.3.2 Determination of Parameters for Standard PEs

To accurately simulate given schedules on real hardware, we must be able to execute tasks and data transfers with precise parameters. As described in Section 6.1.5.2, this is done with a linear regression. To obtain the parameters for the evaluation, the linear regression was performed on two high-end FPGA accelerators.

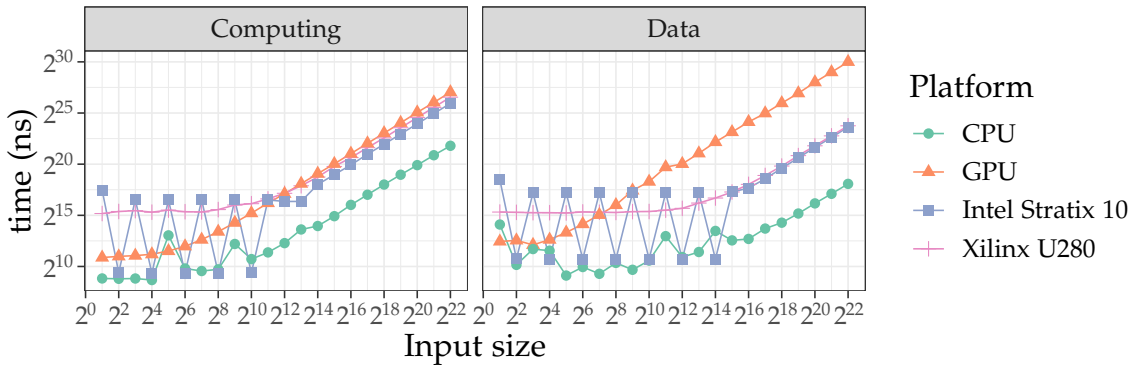



Figure 6.1: Execution times for the standard kernel for a range of input sizes at 500 MHz

Figure 6.1 shows the determined values for a range of input sizes on the Xilinx Alveo U280 and Intel Stratix 10 accelerator cards as a log-log plot. The x-axis shows the input size, in this case measured in 32bit integers, and the y-axis the execution time in ns at 500 MHz. The values for an Intel Core i7-1185G7 CPU and Intel Iris Xe Graphics G7 GPU are shown for comparison. For the Xilinx Alveo U280 accelerator, it is apparent that a standard task cannot be executed shorter than circa 33 µs. Once the input size is large enough (circa $2^{10} = 1024$), the execution time increases linearly with the input. This property can be used to produce tasks of arbitrary execution time over the 33 µs limit.

After the parameters are determined, the tasks are executed. An Out-Of-Order (OOO) command queue `cl::CommandQueue` is created using the property `CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE`. A single OOO queue can provide parallel execution of the tasks. Afterward, for each task the corresponding kernel function is selected and

executed. For each task the corresponding kernel function (PE) is selected and set up. Listing 6.5 shows the procedure that is repeated for all scheduled tasks, ordered by the start time $t_s$. It shows the procedure described above, where lines 1–2 compute the input sizes, lines 4–8 set up the arguments and finally lines 9–10 enqueue the kernel for execution. The enqueuing in lines 9–10 is delayed using a timer in the second strategy, so that the task $v$ can only start executing after the time $t_s(v)$ passes.

```
1  comp_size = comp_beta_1 + data_beta_0 * task.comp_duration
2  data_size = data_beta_1 + data_beta_0 * task.data_curation
3  // Enqueue the task
4  cl::NDRange gsize(1), offset(0), lsize(1);
5  kernel.setArg(0, comp_size);
6  kernel.setArg(1, in_buf);
7  kernel.setArg(2, data_size);
8  kernel.setArg(3, out_buf);
9  queue.enqueueNDRangeKernel(pe.kernel, offset, gsize, lsize,
       ↪ &dependent_events, &task_event);
10 queue.flush()
```

Listing 6.5: Building and enqueing a kernel object for a task using the allocation-only strategy

### 6.3.3 Effects of Reconfiguration Delay

Since the overhead of reconfiguration can be an important variable for the metrics of the schedule, its influence on the outcome must be evaluated. An advantage of our approach to the machine models is that we can vary the reconfiguration overhead and observe its effect on any schedule. This can deliver important insights for developers of FPGA HLS code and vendors alike. Developers are interested in whether it is worth the effort to implement complex task scheduling logic. Vendors can weigh the effort of providing and optimizing, for example, PR capabilities with the benefits this could bring to users.

For his section two features of our approach are particularly important: the type function $t(v)$ and the cost function $c(v, p)$. The former returns a type for each node $v$ in the task graph and the latter the execution cost *per PE* $p$ for each node $v$. Previously, with machine models $\mathcal{M}_{\text{PL}}$ and $\mathcal{M}_{\text{PP}}$ all PEs may execute all tasks. In this section we investigate the influence of reconfiguration overhead. Thus, the task graph generators are extended to include types (setting $t(v)$ for each $v \in V$) and $\mathcal{P}_t^P(p)$ is assigned uniformly random for each $p \in P$. This models a workload where reconfigurations are necessary, because the PEs of a configuration do not have a compatible type for all tasks. To ensure that a schedule can be built, at least one PE with $\mathcal{P}_t^P(p) = t_v$ is generated for each type $t_v$. If the number of PEs is greater than the number of types, each additional PE is assigned a type uniformly at random.

There are several variables of interest:

1. What effect does the reconfiguration overhead have on a schedule?
2. What effect does the reconfiguration overhead have on machines supporting PR compared to machines without?
3. How is the impact of reconfiguration overhead on REFT compared to the output of the optimal solver?
4. Does sorting the task graph into a priority queue using $\mathsf{rank}_{u,p}$ from Section 4.3.2 have a benefit compared to using the conventional $\mathsf{rank}_u$ (as in HEFT)?

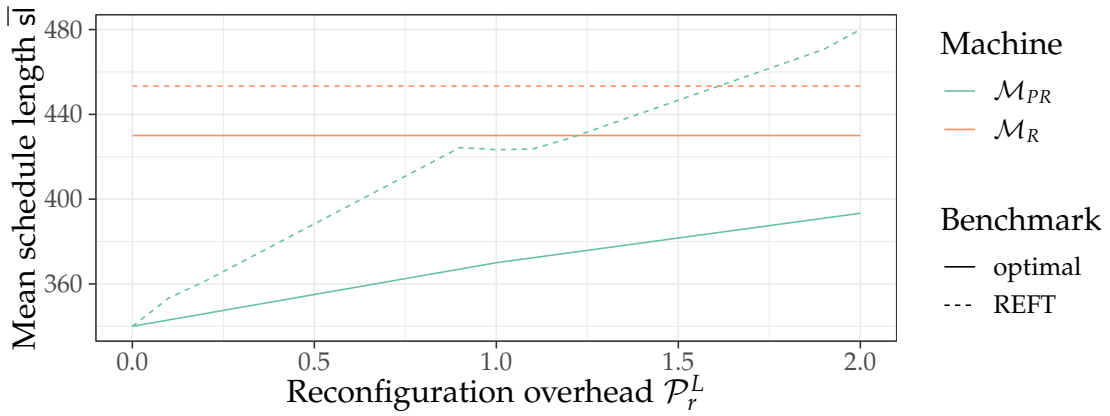



Figure 6.2: Effects of reconfiguration overhead on the schedule length $\overline{\mathsf{sl}}$ in the range 340 – 480 relative to the reconfiguration overhead $\mathcal{P}_r^L$ for two machine models and an optimal and REFT-generated schedule

5. What effect does the reconfiguration overhead have for a range of task graphs with different parameters?

The effects of reconfiguration overhead $\mathcal{P}_r^L$ on the schedule can be directly measured, both for an optimal schedule and for REFT generated ones. Since the cost associated with tasks does not have a unit attached, the reconfiguration overhead is formulated relative to the mean task cost. For a task graph $G = (V, E, w, c, t)$, let

$$\overline{w} = \frac{\sum_{p \in P} \sum_{v \in V} w(v, p)}{|V| \cdot |P|} \tag{6.18}$$

be the mean computation cost over all tasks. In the following the reconfiguration overhead for a location is denoted as multiple (or fractions) of $\overline{w}$.

Figure 6.2 shows the effects of the reconfiguration overhead relative to the average task size for 300 random task graphs. The x-axis shows the reconfiguration overhead/delay $\mathcal{P}_r^L$ where 0 is no overhead and 2 is double the average task cost, i.e. 100 in this case. The y-axis shows the mean schedule length $\overline{\mathsf{sl}}$ for all task graphs. The plot contains lines for the optimal solver output ("optimal") and REFT and differentiates between the two machine models $\mathcal{M}_{\mathrm{R}}$ and $\mathcal{M}_{\mathrm{PR}}$. Machine model $\mathcal{M}_{\mathrm{R}}$ is a $(1, 3, 3, 0)$-model that does not have to perform any reconfiguration to execute any of the task graphs in $G_n$. Machine model $\mathcal{M}_{\mathrm{PR}}$ in contrast is a $(3, 3, 1, 1)$-model that represents a conventional PR-based FPGA system. First, the graphic shows that for machine model $\mathcal{M}_{\mathrm{PR}}$, REFT performs optimal if the reconfiguration overhead is 0 but performs increasingly worse as the overhead grows. Second, the optimal schedules for machine model $\mathcal{M}_{\mathrm{R}}$ perform more similar to machine model $\mathcal{M}_{\mathrm{PR}}$ as the overhead grows. Third, the length of the optimal schedules does not scale linearly with the configuration overhead. At $\mathcal{P}_r^L \leq 1$ the slope decreases for $\mathcal{M}_{\mathrm{PR}}$, indicating that it can perform tasks parallel to the reconfiguration and thus negating some of the cost. The performance of REFT using $\mathsf{rank}_{u,p}$ does not differ from using $\mathsf{rank}_u$ for all values of $\mathcal{P}_r^L$, showing that the well-known upper rank function $\mathsf{rank}_u$ is sufficient even in a reconfigurable scenario. From here on we use $\mathsf{rank}_u$ in REFT.

## 6.4 Evaluation of the REFT Algorithm

Our REFT algorithm introduced in Chapter 4 provides polynomial time task scheduling on reconfigurable hardware. In this section we provide an evaluation of its performance and behavior. The evaluation consists of three parts:

1. The methodology of evaluating this unique approach is explained.
2. REFT is compared to optimal schedules.
3. Effects of properties of the machine model are evaluated.

### 6.4.1 Methodology

The challenge of evaluating REFT lies partly in its novelty: other scheduling algorithms do not support our machine model directly. This can be overcome by adapting either a machine model accordingly or the scheduling algorithms themselves. However, the lack of support of the scheduling of HLS configurations without task-level reconfiguration cannot be overcome this way. We identify two approaches to evaluate REFT:

1. Port other scheduling algorithm so it uses the machine model: This strategy has the problem that we would evaluate the *port* of the scheduling approach that is used as a baseline or comparison. Also, none of the presented approaches from Chapter 2 supports a wide range of hardware that a machine model can describe.
2. Use optimal schedules as emitted by the simulator module of the RESCH framework as a baseline. This has the benefit that the lower bound is independent of the implementation and can be generated for all machine models. The drawback is the comparably long runtime of the solver.

After careful consideration, we chose the option 2. over picking a set of comparison algorithms. Due to the large runtime, the number of tasks, locations and PEs for that is feasible to find a solution is rather small, but still sufficient to make meaningful statements about the quality of REFT. Especially for smaller input sizes, we expect REFT to perform near the optimum and any large deviations must lead to careful investigation.

Thus, the evaluation starts with simple constellations and gets progressively more complex. We evaluate a series of $(\{1,\ldots,l\},\{1,\ldots,m\},\{k \mid k \geq 1\},[0,1])$-machine models on random and structured task graphs.

### 6.4.2 REFT versus Optimum

The purpose of this subsection is to establish a baseline with the simulator module output and to compare schedules generated with REFT with it. For each machine model and each set of parameters, a set of random task graphs $G_n = \{G_0, G_1, \ldots, G_{n-1}\}$ are generated and simulated with the solver module as well as the REFT Python implementation. Unless specified otherwise the number of graphs per set of parameters $n = 30$: each of the three graph generator algorithms is used to generate ten graphs. Note that although the graphs are random, the *same set* $G_n$ is used for the simulation in the solver as well as with REFT.

#### 6.4.2.1 Parallel Locations

In this simple scenario the $(\{1,2,3\},1,1,0)$-machine model $\mathcal{M}_{\text{PL}}$ allows partial reconfiguration and is restricted as little as possible:

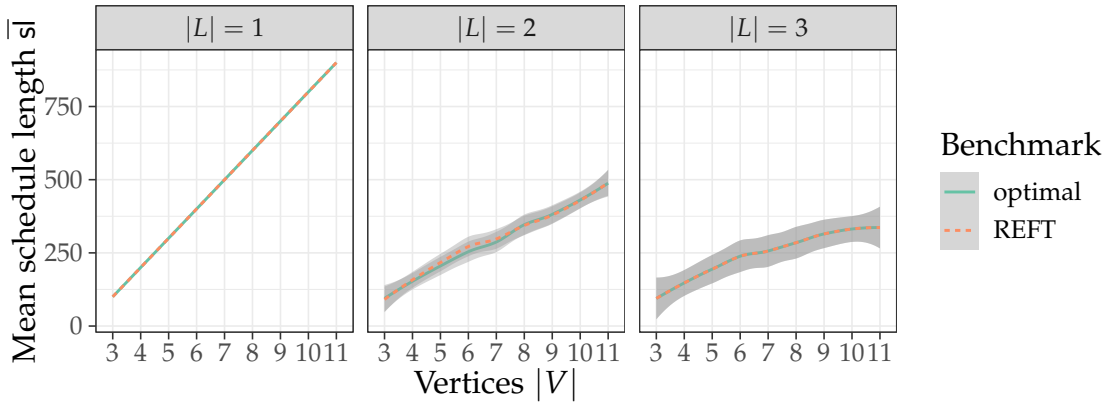



Figure 6.3: The mean schedule length $\overline{\text{sl}}$ for a partially reconfigurable $(\{1,2,3\},1,1,0)$-machine model for random task graphs with 3 to 11 vertices $|V|$

- No properties are set, i.e. $\mathcal{P}$ is empty.
- Task types ($t$) are not set for any task, i.e. any PE can execute any task. This is equivalent to all tasks having the same type.
- Communication cost is not set.
- Computation cost is uniformly 100.

Figure 6.3 shows the makespan for the number of locations $|L| = \{1,2,3\}$. The x-axis shows the number of tasks in the uniformly random task graph and the y-axis shows the corresponding makespan. The gray area denotes the standard deviation, although it is barely visible for such a simple scenario. As expected, the makespan grows linearly when only one location is available for execution. Due to the lack of properties, the reconfiguration overhead is 0 and the tasks are executed in topological order.

REFT manages to find that solution in all cases. For two locations, there is potential for parallelism, although the amount of possible parallelism is dependent on the task graph. Having more than one location in a PR-enabled machine model allows for parallelism, and it can be seen that the makespan is reduced for $|L| = 2$ and further reduced for $|L| = 3$. It is also visible that REFT performs as well as the optimal solver for these scenarios, since both lines are congruent.

#### 6.4.2.2 Parallel PEs

In this parallel scenario the $(1,1,\{2,\dots,\},0)$-machine model $\mathcal{M}_{\text{PP}}$ is static, i.e. *the* configuration can contain more than one PE, but it can only be configured to one location. Trivially, this should yield comparable results to the simple scenario, especially since communication is not considered. Since there is only one configuration, none of the schedules does involve any reconfiguration, but without the reconfiguration overhead $\mathcal{P}_r^L$ set, REFT should perform similar. Indeed, the makespan and other metrics do not show much difference between both scenarios as can be seen in Table 6.2. The table shows the mean values for both the simple $\mathcal{M}_{\text{PL}}$ model above and $\mathcal{M}_{\text{PP}}$ from this section. The values are so similar that we do not include the makespan plot for this machine model. The table shows that, without any communication or reconfiguration overheads, the schedule has the same quality for both scenarios. This is unsurprising, since schedules for $\mathcal{M}_{\text{PP}}$ can execute a task on a different PE within the same configuration, whereas schedules for $\mathcal{M}_{\text{PL}}$ must configure one of the available locations with a PE (without any overheads).

| Machine | | | | | | |
|---|---|---|---|---|---|---|
| **Model** | $\|P\|$ | $\|L\|$ | **Makespan** | **Speedup** | **SLR** | **Slack** |
| $\mathcal{M}_{\text{PP}}$ | 1 | 1 | 500.00 | 1.00 | 5.00 | 296.44 |
| | 2 | | 295.41 | 1.63 | 2.95 | 91.84 |
| | 3 | | 245.96 | 1.99 | 2.46 | 42.40 |
| $\mathcal{M}_{\text{PL}}$ | 1 | 1 | 500.00 | 1.00 | 5.00 | 296.44 |
| | | 2 | 295.26 | 1.63 | 2.95 | 91.70 |
| | | 3 | 245.59 | 2.00 | 2.46 | 42.03 |

Table 6.2: Comparison of the mean of a set of metrics for both $\mathcal{M}_{\text{PL}}$ and $\mathcal{M}_{\text{PP}}$. Both machines executed the same set of task graphs.

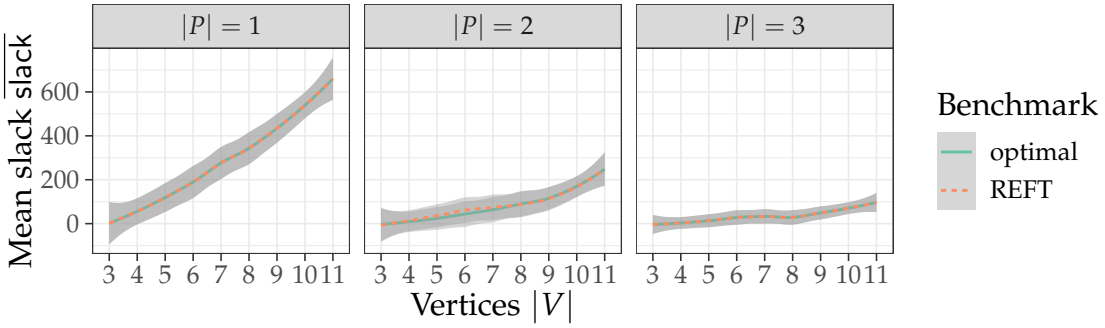



Figure 6.4: The mean slack $\overline{\text{slack}}$ for a partially reconfigurable $(\{1,2,3\},1,1,0)$-machine model for random task graphs with 3 to 11 vertices

Figure 6.4 shows the slack for both the solver output and for REFT. The x-axis shows the number of vertices $|V|$ and the y-axis shows the slack per task graph size. The figure also shows the slack for three different numbers of PEs $|P|$. For $|P| = 1$ the slack grows at a higher rate with the task graph size compared to $|P| = 2$ or $|P| = 3$. This is to be expected: with only one PE (and one location), the speedup is 1 and the tasks are executed in topological order one after another. Slack exists, because for two tasks in the same equivalence class under $\prec$, then their start times can be interchanged without violating constraint 3. For $|P| > 1$ the tasks can be scheduled closer to the optimal start time (w.r.t an infinite number of PEs) and thus lowering the slack. The graphs in Figure 6.4 indicate that the machine model with $|P| = 3$ is already near the optimum and introducing more PEs will not yield a benefit, especially for smaller task graphs.

#### 6.4.2.3 Task Graphs

Besides the machine model the task graphs have an impact on the schedule. In this section we evaluate how REFT handles different task graph structures with varying parameters. The construction of the random task graphs is described in Section 6.1.1. The first parameter we can adjust is the task graph size ($n$) as in the previous sections. The second parameter is $p$, the probability that an edge between two nodes exist. The layer-by-layer techniques offers a third parameter, specifically the number of layers. A total of 84,000 task graphs were simulated for this section. The average runtime of the prototypical REFT implementation in Python, that is the time it takes REFT to generate the schedule (not the makespan) is 10.5 ms, compared to 300 ms that

the solver takes. For larger graphs the runtimes diverge further as shown below.

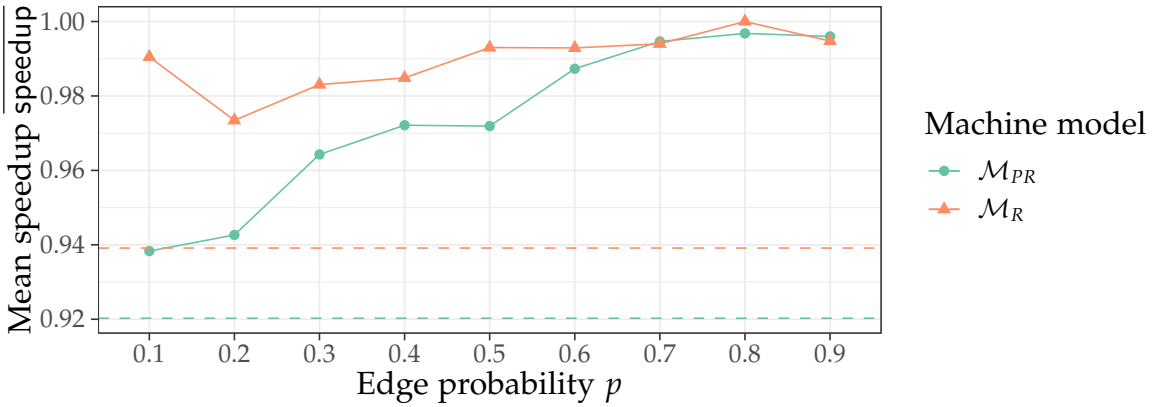



Figure 6.5: The mean speedup of REFT speedup in the range 0.92 – 1.0 over the optimal solution for uniformly random graphs (dashed lines) and graphs from the Erdős–Rényi method over a range of edge probabilities $p$; a total of 84,000 schedules are summarized

Figure 6.5 shows the relative speedup of REFT compared to the optimal solution of a total of 84,000 task graphs with size 10 constructed with the random and the Erdős–Rényi models. The dashed line shows the speedup for uniformly random task graphs, whereas the solid line shows values for the Erdős–Rényi graphs. For the latter the x-axis shows an increasing value of $p$, i.e. the probability of which an entry is 1 in the upper triangle adjacency matrix. Error bars are omitted as the standard deviation is close to 0. The graph shows that REFT performs worse for low $p$ and better for high values of $p$. It also shows that REFT performs better – even optimal – for reconfigurable machine models and worse for partially reconfigurable ones. The tendency to perform better for higher values of $p$ is likely due to the ranking function outputting more suitable orders for the REFT algorithm with many edges. Further analysis of single schedules showed that REFT tends to introduce more fragmentation regarding the instances on partially reconfigurable machines. Once a configuration is instantiated at a location, all PEs from other configurations are unavailable for that time. A problem that is not as apparent in dynamically reconfigurable machines, since the PEs from the same configuration do not suffer from fragmentation.

Figure 6.6 shows the relative speedup of REFT over the optimal solution for 75,600 task graphs with size 10 constructed with the layer-by-layer technique. The x-axis shows the probability $p$ with which an edge from two vertices is created and the y-axis shows the number of levels from 1 (only 1 layer) to 10 (each task on its own layer). Again, the figure distinguishes between two machine models: one with PR support and one without. In general the performance of REFT is worse for the PR-based machine model with a mean speedup $\overline{\text{speedup}}$ of 0.922 compared to 0.978 for the machine model without PR. A penalty of – on average – 7.8% and 2.2% is relatively low, considering that the prototypical Python implementation of REFT uses a fraction of the time of the solver. Further, REFT produces particularly good results for independent tasks (either low $p$ or with one layer).

#### 6.4.2.4 Contention-Aware Schedules

Instead of using the fixed communication cost, REFT also supports edge scheduling to model communication congestion, i.e. to generate contention-aware schedules according to Definition 27.

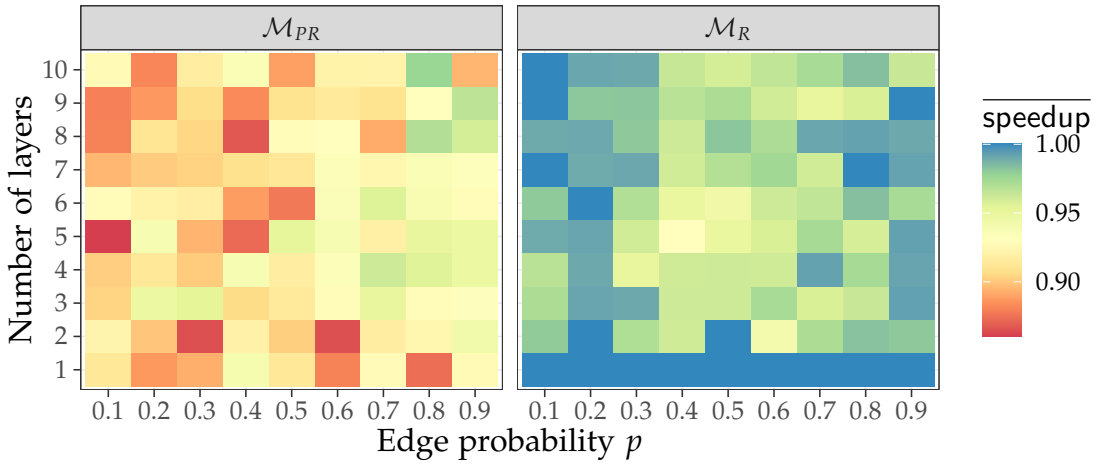



Figure 6.6: The mean speedup of REFT $\overline{\text{speedup}}$ in the range 0.85 – 1.0 over the optimal solution for the number of layers, the edge probability $p$ and two machine models; a total of 75,600 schedules are summarized

This is further described in Section 3.4.5.3 and the REFT variant using edge scheduling is described in Section 4.3.5, in particular by Algorithm 4.3 on page 74.

The purpose of this subsection is to investigate the effect of edge scheduling on schedules and the output of REFT in particular. We repeat the experiments from the previous section but include edge scheduling. For all machine models the default topology (see Section 3.2.2.1) is used. Then we vary the communication cost per edge from 0 to double the cost of the average task. Sinnen describes this ratio for a task graph $G = (V, E, w, c, t)$ as the Computation-Communication-Ratio (CCR) $\text{CCR} = \frac{\sum_{e \in E} c(e)}{\sum_{v \in V} w(v)}$ [125]. Varying this parameter gives insight into how a scheduling algorithm (and in our case also a machine model) responds more communication or more computation-bound workloads.

Figure 6.7 compares the mean schedule length $\overline{\text{sl}}$ for 840 randomly generated task graphs for the solver and REFT each with and without edge scheduling. This means in both cases the communication cost are included, but only with edge scheduling the congestion is also accounted for. It also shows the mean runtime on a logarithmic scale using the same breakdown of solver and REFT output and edge scheduling. Both plots are divided up for a partially reconfigurable and a dynamically reconfigurable machine model. The figure gives a few insights:

1. The effect of the edge scheduling is clearly visible. Over all task graphs, the mean optimal schedule length is 2,486 with congestion compared to 932 without.
2. REFT performs well within 3% of the optimal schedule length except for the partially reconfigurable machine with edge scheduling.
3. This combination also took the solver significantly more time to find the optimal solution, i.e. more than 149 s compared to 0.38 s on average.
4. In general – and somewhat unsurprisingly – scheduling edges additional to tasks increases the search space and therefore the runtime for the solver. The effect is also visible for REFT but orders of magnitude smaller.

This further shows the problem of using the solver for anything other than the verification of other algorithms like REFT: even for small task graphs the time to find a solution is much longer

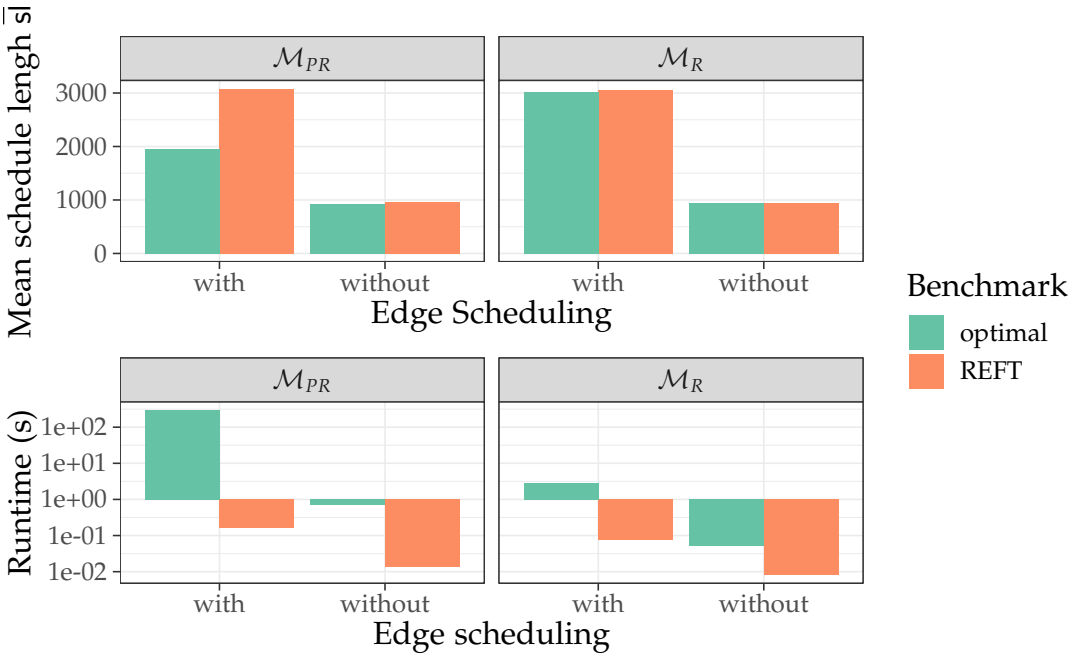



Figure 6.7: The logarithmic runtime of REFT and the mean schedule length $\overline{sl}$ with and without congestion for two machine models

than a typical application executes, let alone making up for the optimization potential over even a naive schedule.

Figure 6.8 shows the effect of the CCR on REFT and the optimal schedule. The x-axis shows the CCR and the y-axis shows the mean schedule length $\overline{sl}$ in the upper part and the runtime in seconds in the lower part. Again, the negative impact of REFT with edge scheduling for the partially reconfigurable machine is clearly visible. It is also apparent that the difference between the optimum and REFT's output increases with the CCR. At the same time, REFT's output is again within 3% of the optimum for all other cases.

Upon further inspection of the schedules it becomes apparent that this effect seems to have its origin in the much simpler allocation process of communication edges that REFT uses. Whereas the solver can find – for each link – the optimal point in time, REFT uses the union of all communication free time slots along the path. This means that REFT must find an interval where all links along the path do not communicate at the same time and allocate an edge during this interval. Even more so, REFT must take the slowest link's communication bandwidth as bandwidth for all links across the path. In contrast, the solver is much more flexible, since it must only fulfill constraints 5 and 6 (Definition 27).

### 6.4.3 Other Effects on REFT

While the comparison with optimal schedules gives a clear frame of reference, it is not feasible to use the solver module for significantly larger task graphs, since it is too slow. However, the effects the machine model, its properties and other parameters are of interest in more complex scenarios.

We generated a set of 2,100,000 task graphs with a size of 100 vertices on average with varying factors like the CCR and the imbalance (the range of task costs) for both partially reconfigurable and dynamically reconfigurable machine models. These were scheduled by REFT, both in the

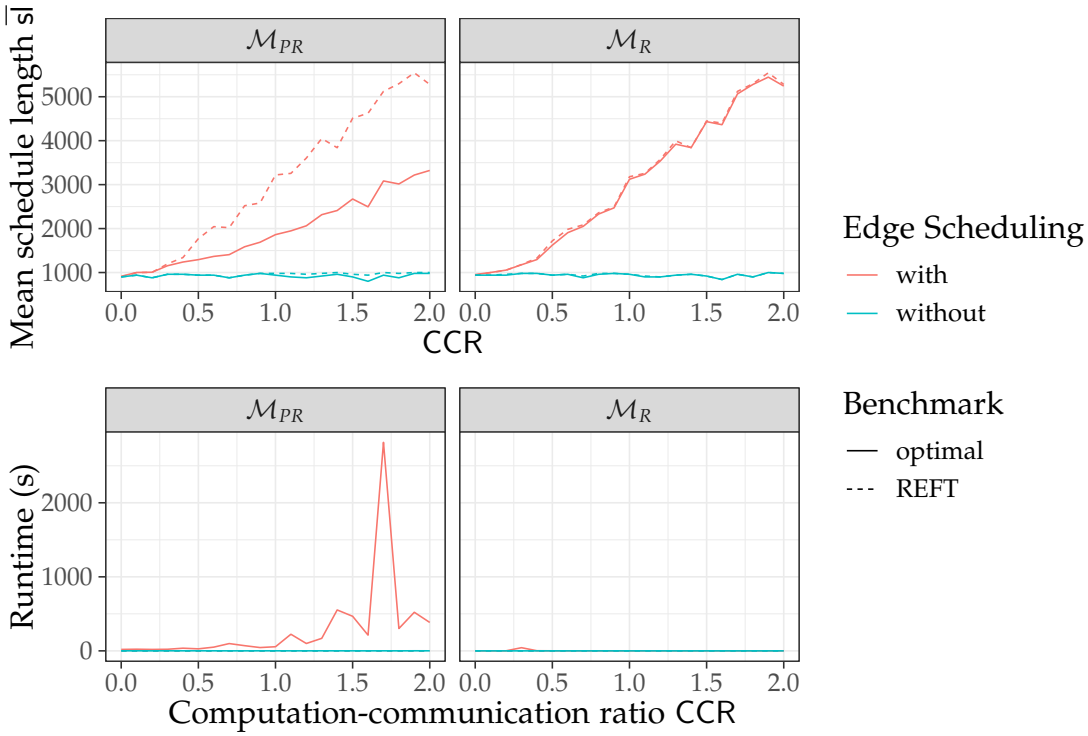



Figure 6.8: The runtime of REFT and the mean schedule length $\overline{sl}$ over the computation-communication ratio CCR

more simple static communication cost variant as well with the edge scheduling variant.

Figure 6.9 shows the impact of the CCR and the imbalance on the schedule length. The x-axis shows the CCR and the y-axis the mean schedule length $\overline{sl}$ over all executed schedules. A few observations can be made from the figure:

- Clearly, the CCR has an effect on the schedule. This can also be observed in Figure 6.8 and is confirmed here.
- The dynamically reconfigurable machine model performs better than the partially reconfigurable one. Further inspection indicates that REFT can make better use of backfilling in the former case which leads to shorter schedules. This effect is negligible if the configuration overhead is near zero (not shown in the figure).
- The imbalance has an effect on the schedule. Large imbalances lead to longer schedules, although the mean total execution cost is equal for all imbalance values. This effect can be attributed to fragmentation and higher mean data ready times. Any long task will postpone *all* successors, while the successors of a short(er) task can only start execution once all other dependencies are met.
- The effect the CCR on schedules with congestion is large. Note that the scales on the top plot differs an order of a magnitude compared to the bottom one.
- The edge scheduling works significantly better with no imbalance. Both partially reconfigurable and dynamically reconfigurable machine models perform better without any imbalance. This can be attributed to a similar effect as the third point.
- Although it may be difficult to see in Figure 6.9, using edge scheduling gives shorter schedules for a $ccr = 0$ for larger imbalance compared to not using it. This may be

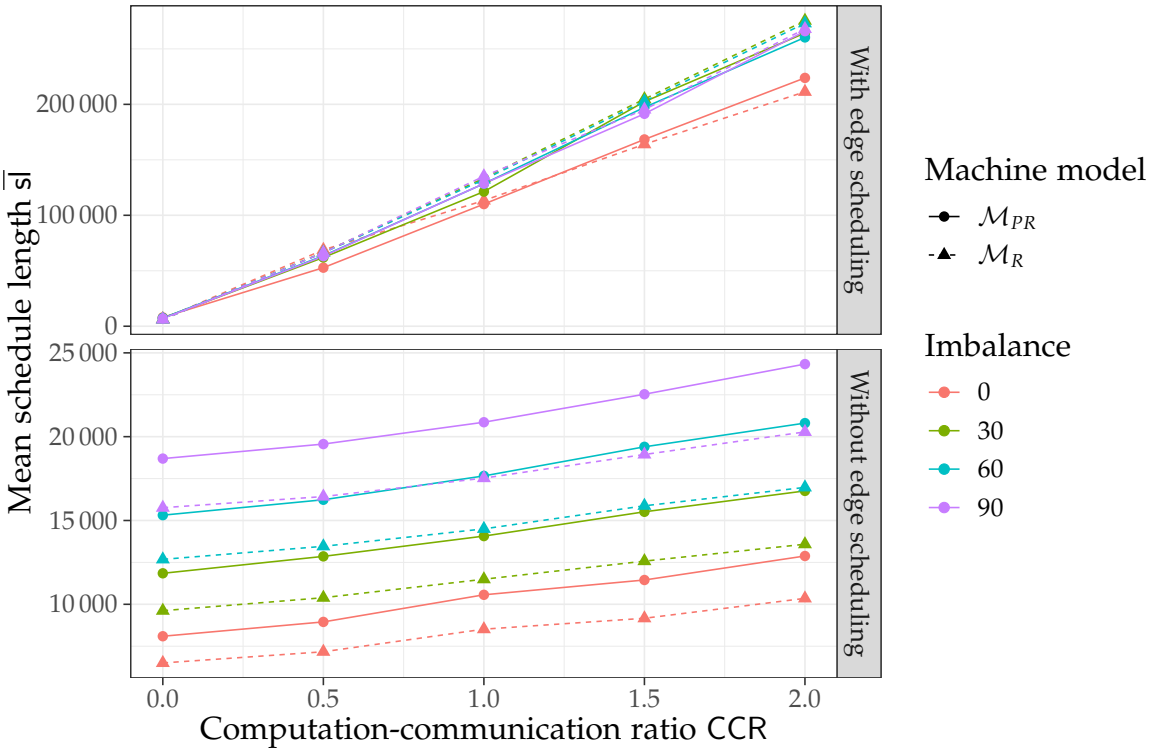



Figure 6.9: The mean schedule length $\overline{\text{sl}}$ of REFT over the computation-communication ratio CCR with and without edge scheduling

counterintuitive, but fact that every edge is explicitly scheduled allows for better backfilling, i.e. finding lower EFTs for the tasks.

We also tested much larger graphs with up to 10,000 nodes: the general trends persisted. Performing edge scheduling does come at a price, though. Already for task graphs with 500 nodes we observed runtimes of REFT of over 200 seconds. Upon further investigation it is clear that the Python package `portion` used for interval operation becomes a bottleneck. It is not well optimized for the operations done in REFT. However, we want to emphasize that REFT *is* a prototype implementation for experimental purposes. One could optimize the interval implementation, employ caching or explicitly construct the intervals during the allocation, removing the need for costly operations altogether.

## 6.5 Evaluation of the DSE

The DSE process aims to optimize a machine model for a given task graph using the GA. In this section we assess how well the optimization process works. In particular, we are interested in the difference in schedule length of a naive input program and an optimized program. This scenario represents the usage of DSE as a compilation step. In this scenario, a programmer writes an HLS program and the DSE identifies a good allocation of PEs and configurations.

The resource usage plays an important role in DSE. If the resource usage is not considered, the DSE will opt to create as many clones as possible, disregarding all negative effects of static and dynamic congestion. As detailed in Section 5.4.3, our DSE approach uses the available chip

resources to model static congestion and to avoid overfilling configurations, so that they can be synthesized.

| Resource | Quantity |
|---|---|
| LUT | 1,866,240 |
| FF | 3,732,480 |
| Memory Blocks | 11,721 |
| DSP Units | 5,760 |

Table 6.3: Resources of Intel FPGA PAC D5005 with Intel Stratix 10 GX FPGA [12]

For this evaluation we use the resource properties of an Intel FPGA PAC D5005 accelerator card which contains an Intel Stratix 10 GX FPGA. Table 6.3 shows the resources that are available on the FPGA. The values already have the resources used by the *OpenCL shell* subtracted, meaning that all resources are available for PEs. For the PEs the used resources are estimated by the `aoc` compiler offered by Intel. A call to `aoc -report -profile -g -rtl -report pes.cl` will generate a report file with accurate numbers for the kernel functions (PEs) included in `pes.cl` that will be included in one configuration. The `-rtl` flag stops the compilation at the RTL-level, avoiding the long-lasting place-and-route step of the compilation. The numbers `aoc` generates generally do not change when invoked without the `-rtl` flag, which results in the configuration file, i.e. the bitstream.

The structure of this evaluation follows the structure of Chapter 5:

1. An evaluation of InterCO
2. An evaluation of IntraCO
3. An evaluation of CO

### 6.5.1 Evaluation of InterCO

Recall that the purpose of InterCO is to solve the combinatorial problem of arranging PEs in configurations for a given task graph. In this section we evaluate the effect of the GA and its performance.

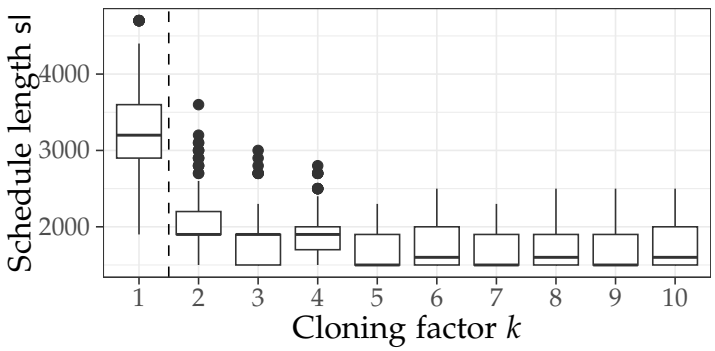



Figure 6.10: InterCO: schedule length sl over the cloning factor $k$

Figure 6.10 shows a box-plot for schedule lengths for all machine models generated – over all generations – during an execution of the DSE for an implementation of the LU-decomposition.

The y-axis shows the makespan sl and the x-axis shows the cloning factor $k$. A cloning factor of 1 is equivalent of the simple encoding as described in Section 5.3.1. The vertical lines on the boxes show the minimum/maximum, the box itself show the 25% quantile and the bold line represents the median. The dots show outliers. The GA generated 50 generations with 5 solutions each per $k$. The upper bound for the number of possible configurations is 4, since there are 4 different kernel types. It is clear that without any cloning, the schedule length is not optimal, i.e. the best optimal schedule generated by the GA is 1,900, while the best overall schedule length is 1,500. However, the best schedule length is found already with $k = 2$ and does not improve for higher $k$.

While this represents the GA's output for one particular task graph and set of resource properties, the overall procedure for optimization can be applied for all task graphs and hardware.

Furthermore, for a single user (e.g. in an HPC use case), the goal might be to reduce the makespan. In multi-tenant system, the fitness function could target the resource usage or a weighted combination of both. This is especially true if the task graphs of all tenants cannot be considered in combination, but in isolation. In the former case one could still optimize the combined set of tasks graphs, in the latter one cannot assume knowledge about the behavior of other users.

### 6.5.2 Evaluation of IntraCO

The purpose of IntraCO is the automatic optimization of a given HLS program and one (or more) task graphs using one configuration. For the evaluation, we used the LU-decomposition of matrices split up into $10 \times 10$ blocks. The task graph was built from a trace of an OpenCL program on an Intel Stratix 10 GX accelerator card as described in Chapter 3.

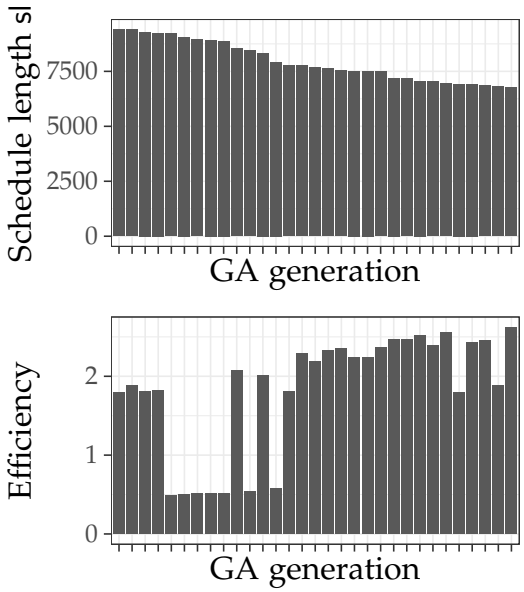



Figure 6.11: IntraCO: schedule length sl and efficiency over the GA generation

We input this data into the GA using the encoding presented in Section 5.5.3 and generate 50 generations with a population of five chromosomes each to optimize the machine model (i.e. the HLS program). Figure 6.11 shows two bar charts: on the top, the best (minimal) schedule length per GA generation on the x-axis is shown. On the bottom, the efficiency for the same generations of the GA is plotted. The GA is able to reduce the schedule length from 10,578 down to 6,773, a

reduction of over 35 %. This solution is also the one with the highest efficiency with 2.6 compared to a naive solution with an efficiency of 0.5.

To show the generality of IntraCO, we perform a set of optimizations on 100 $(1, 1, 9, 1)$-machine models that are randomly generated using the three techniques from Section 6.1.1. Each machine model starts an equal allocation of the system's resources. For example, if the location provides $n$ LUT blocks and 5 PEs are required, then the machine model is generated so that each PE requires $\frac{n}{5}$ LUT blocks.

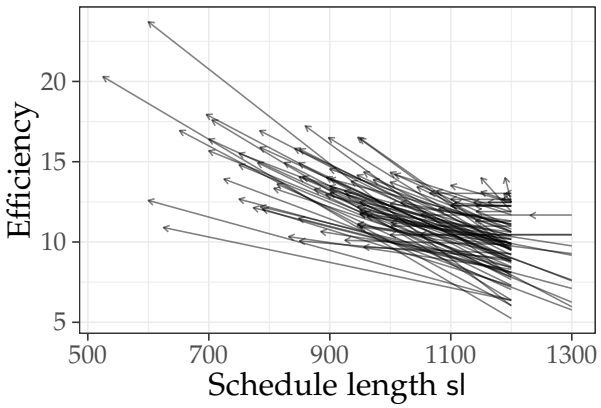



Figure 6.12: IntraCO: efficiency over the schedule length sl

Figure 6.12 shows the schedule length and efficiency for both the initial and optimized machine model. The x-axis shows the schedule length sl and the y-axis the efficiency. Each machine model is depicted as an arrow that points from the model's initial schedule length and efficiency to its optimized values. For example, if arrow points from the $(1300, 10)$ coordinate to $(900, 15)$, then it represents a machine model that was optimized so that the input task graph is executed with a schedule length of 900 instead of 1,300, while the efficiency improved from 10 to 15. Clearly visible is the tendency for arrow pointing top-left, i.e. the GA manages to increase the efficiency and reduce the schedule length.

This impression is backed by the underlying data. Considering the relative speedup as $\frac{\mathsf{sl}_{\text{in}}}{\mathsf{sl}_{\text{out}}}$ where $\mathsf{sl}_{\text{in}}$ and $\mathsf{sl}_{\text{opt}}$ are the input and optimized schedule lengths, respectively. The relative speedup for the optimized machine models is 1.34 with a standard deviation of 0.26. Conversely, the relative gain in efficiency is 0.48 over the input machine models with a standard deviation of 0.52. This shows that IntraCO is able to automatically identify optimized machine models for specific work loads. In particular, it is possible to decrease the schedule length using only one configuration and a limited set of resources.

### 6.5.3 Evaluation of CO

As a combination of InterCO and IntraCO, CO can be evaluated as such. To assess the benefits of the combination of both strategies, we generate 100 $(1, \{1, 2, 3\}, 9, 1)$-machine models and observe their optimization potential. In addition to IntraCO, the machine models are now also optimized with InterCO to examine if this further reduces the target metric, the schedule length.

Figure 6.13 shows the results of optimizing the 100 generated task graphs with CO. It depicts the schedule length on the x-axis and the efficiency on the y-axis and arrows denoting the development form input machine model to optimized machine model, similar to Figure 6.12. Note that the $(1, \mathbf{1}, 9, 1)$-machine model – represented by the figures in the left column with

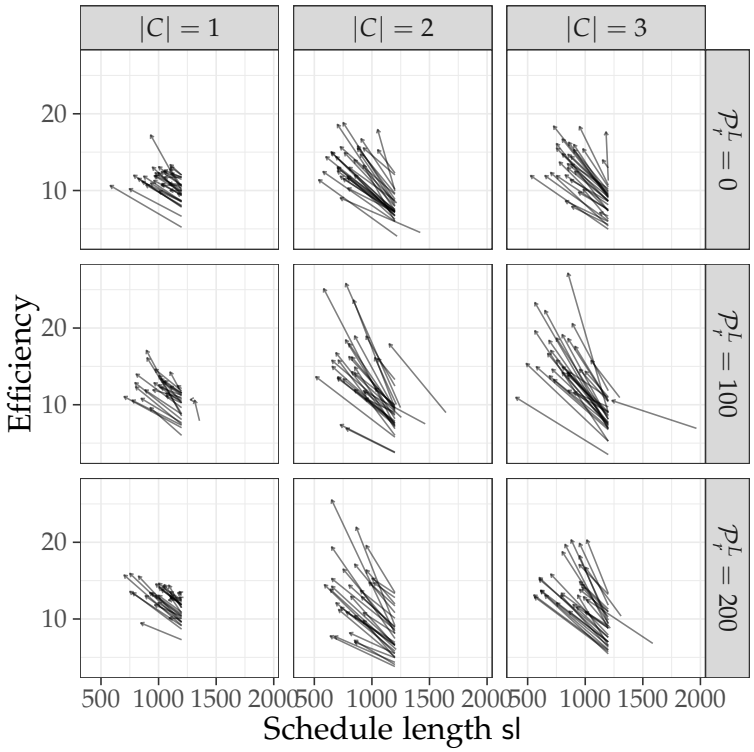



Figure 6.13: CO: efficiency over the schedule length sl split by the number of configurations $|C|$ and configuration delay $\mathcal{P}_r^L$

$|C| = 1$ is equivalent to only performing IntraCO, since there exist only one configuration and consequently the optimizations may only be performed within that configuration. In the middle and right column two and, respectively, three configurations are available for the GA. The top row shows machine models without any reconfiguration delay $\mathcal{P}_r^L$. The middle row shows a reconfiguration delay equal to the mean task length and the bottom row a reconfiguration delay equal to twice the mean task length. We make the following observations:

- As expected, CO is able to optimize the machine models over all. All arrows in the plot point to the left direction, i.e. the CO identified a machine model that generates a faster schedule.
- The introduction of a second configuration is beneficial for the schedule length as well as for the efficiency. The second (third) configuration make use of the existing resources (like LUTs and DSPs) a second (third) time, increasing the efficiency.
- The increase in efficiency is not affected by a lower configuration delay, i.e. in the top row. Remember that REFT performs the scheduling during the optimization. This indicates that REFT is able to alleviate the cost of reconfiguration.
- The introduction of more configurations is not always beneficial.

The last point is evident by looking at the accumulated data. Where the mean schedule length $\overline{\text{sl}} = 977$ (135 standard deviation) when the number of configurations $|C| = 1$, it is considerably lower for $|C| = 2$ with a mean schedule length $\overline{\text{sl}} = 798$ (131) but it goes up to $\overline{\text{sl}} = 824$ (158) for $|C| = 3$.

This concludes the evaluation of CO. Accumulated date for all the measurements performed in this chapter can be found in Appendix B.

## 6.6 Summary

This chapter presents the evaluation of our work in Chapters 3, 4 and 5. Our software framework RESCH to describe machine models, task graphs and REFT is introduced. Using the RESCH framework, we evaluated:

1. The definition of a machine model and how it can describe FPGA-based accelerators for task scheduling.
2. The polynomial time scheduling task algorithm REFT.
3. The DSE to optimize machine models representing HLS programs for a given workload.

We found that machine models are able to accurately describe FPGA-based accelerators for the purpose of analyzing task scheduling algorithm. The challenge of reliably re-producing existing task graphs is met by automatically tuning OpenCL kernels to the hardware and using the RESCH framework as a scheduling runtime.

The polynomial task scheduling algorithm REFT is compared against optimal schedules generated from a constraint programming framework. We found that REFT can perform at near-optimal schedule lengths for a wide range of settings, while having a significantly shorter runtime. It also handles limited reconfigurability well, with the results being nearer to the optimum than using conventional PR-based schedules.

The DSE can be used to optimize machine models for a given task graph, descreasing the schedule length and increasing the efficiency for a wide range of workloads. Both InterCO and IntraCO and their combination CO have a positive impact on the resulting schedules. The process is executed in time frames that allow an integration into the HLS compilation process to automatically optimize code without the interaction of a user.

The next chapter provides a conclusive view on our work and wages an outlook for further research based in our work.

# Chapter 7

# Conclusion

In this chapter we summarize our work and venture an outlook to future work as well as potential applications of the approach.

## 7.1 Conclusions

FPGAs have a lot of potential for computation offloading, which is currently not widely utilized. The PR feature of FPGAs, commonly leveraged in research for task scheduling, faces limited practical application due to various impediments. Consequently, we pursue an alternative approach, devising and evaluating task scheduling methodologies on FPGAs without relying on task-level reconfiguration.

We present a novel framework to describe FPGA-based accelerators at differnt levels of precision as the basis for analysis. The evaluation shows that it can not only be applied to a range of accelerators but also be used to compare purely Partial Reconfiguration (PR)-based scheduling strategies to alternative ones. Additionally, comparing the framework against existing FPGA-based hardware demonstrates that it can accurately model the scheduling behavior as implemented in our RESCH framework evaluation. Using this model, the differences between task-level reconfigurations and more ergonomic reconfigurations can be quantified. This opens a path to potentially vastly simplified software development models with comparable efficiency and performance.

Two approaches for task scheduling, built on the framework, have been introduced. Both are also agnostic to the granularity of possible reconfigurations. The static scheduling approach is employed to generate optimal schedules for a particular hardware model. Due to the nature of the scheduling problem, it is only feasible for problems of smaller size or for exact comparisons. A polynomial-time algorithm is more suited for implementations in live systems. In the evaluation, we demonstrate that it performs near optimally in many cases, but does so in intervals feasible for use in production environments. It must be mentioned that both approaches – in contrast to published research – work transparently with currently available, standardized technologies like OpenCL.

Another central and novel contribution is the optimization of the underlying hardware model for a given task graph or set of task graphs. Since FPGAs are reconfigurable, this GA-based algorithm determines the best organization of logic into configurations. The evaluation shows that the approach is able to increase the efficiency of the usage of FPGA resources while optimizing task schedules. A new generation of software environments can integrate this algorithm, automatically optimizing applications for a given FPGA-based accelerator.

## 7.2 Future Work and Applications

Our work lays the groundwork for further endeavors in task scheduling on FPGA-based accelerators, especially considering limited support for PR to build better abstractions.

Our approach can be applied as part of a HLS workflow: for example, an FPGA vendor with precise knowledge about its FPGA-based accelerator platform may describe the accelerator as a machine model. During the HLS workflow, the machine model may be used to find possible schedules and perform DSE to either automatically implement optimizations or present the user of the workflow with feedback and hints about their code. Also, REFT can be implemented as a part of a HLS runtime environment, steering the execution of tasks in a reconfiguration-aware manner without reliance on task-level reconfiguration. FPGA software engineers can use our approach to assess the benefit of implementing a complex PR-based scheduling approach.

We see the following concrete issues that need further research:

1. The potential impact of hardware support for the proposed approach is significant, especially considering the current motivation stems from the absence of portable PR features. If vendors wish to adopt this approach and invest in explicit support for shell-based setups, it could lead to notable changes in scheduling strategies such as REFT.
2. The granularity of the machine model plays a crucial role in the accuracy of schedules. Certain characteristics of the model may significantly influence the precision of scheduling outcomes. It is essential to develop a method to measure the significance of each property in the overall model.
3. The applicability of the proposed approach extends beyond FPGA-based hardware. It is conceivable that Coarse-Grained Reconfigurable Architectures (CGRAs) could be modeled and utilized within the REFT framework, suggesting a broader potential for the approach in various reconfigurable hardware contexts.

It may be also be interesting to investigate whether the application of our approach leads to broader acceptance by users and vendors, thereby bringing wider adoption to FPGAs. Further, it needs to be evaluated whether a PR-less scheduling strategy could enable task-based approaches in areas that currently refrain from using a tasking programming model.

# Appendix A

## Listings

Listing A.1 shows the JSON Schema description for machine model definitions. This schema defines required fields and the structure of valid definitions, so that the RESCH framework can process it and use it to generate an internal representation of the machine model. An example of a complying file can be found in the RESCH framework's `sample` directory.

```
{"$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "configurations": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "id": { "type": "integer" },
          "file_name": { "type": "string" },
          "PEs": {
            "type": "array",
            "items": {
              "type": "object",
              "properties": {
                "id": { "type": "integer" },
                "function_name": { "type": "string" }
              },
              "additionalProperties": true,
              "required": [ "id", "function_name" ]
            },
            "additionalItems": true
          }
        },
        "additionalProperties": true,
        "required": [ "id", "file_name", "PEs" ]
      },
      "additionalItems": true
    }
  },
  "additionalProperties": true, "required": [ "configurations" ]}
```

Listing A.1: JSON Schema for machine model definitions.

Listing A.2 shows the JSON Schema description for a schedule so that is can be processes by the RESCH framework. For each scheduled task it consists of a PE identifier, a task identifier and the start time of the task execution.

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "properties": {
    "schedule": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "id": { "type": "integer" },
          "PE": { "type": "integer" },
          "t_s": { "type": "integer" }
        },
        "additionalProperties": false,
        "required": [ "id", "PE", "t_s" ]
      },
      "additionalItems": false
    }
  },
  "additionalProperties": false,
  "required": [ "schedule" ]
}
```

Listing A.2: JSON Schema for schedules.

Listing A.3 shows the implementation of standard PEs in OpenCL that are used in the evaluation. It uses the definition in the `CLONE_PES` macro to implement a standard PE and a series of macros in lines 10–19 to generate clones of the function. Compiling the file with `NUM_PES` set to an integer between 1 and 9 can be used to set the number of standard PEs. The size of the buffer (to influence the resource consumption per PE) can be specified with `BUFSIZE`.

```
1  #ifndef NUM_PES
2  #define NUM_PES 1
3  #endif
4
5  #define BUFSIZE 16
6
7  #define PASTER(x, y) x##_##y
8  #define EVALUATOR(x, y) PASTER(x, y)
9
10 #define PE_CLONE_1 CLONE_PES(1)
11 #define PE_CLONE_2 PE_CLONE_1 CLONE_PES(2)
12 #define PE_CLONE_3 PE_CLONE_2 CLONE_PES(3)
13 #define PE_CLONE_4 PE_CLONE_3 CLONE_PES(4)
14 #define PE_CLONE_5 PE_CLONE_4 CLONE_PES(5)
15 #define PE_CLONE_6 PE_CLONE_5 CLONE_PES(6)
16 #define PE_CLONE_7 PE_CLONE_6 CLONE_PES(7)
17 #define PE_CLONE_8 PE_CLONE_7 CLONE_PES(8)
18 #define PE_CLONE_9 PE_CLONE_8 CLONE_PES(9)
19 #define GEN_PES() EVALUATOR(PE_CLONE, NUM_PES)
20
21 #define CLONE_PES(INDEX)                                          \
22   kernel void pe_##INDEX(                                         \
23       const int cost, global const int *restrict data,            \
24       const int data_size, global int *restrict out) {            \
25     float val = 23;                                               \
26     float local_buf[BUFSIZE];                                     \
27     for (int i = 0; i < data_size; i++) {                         \
28       local_buf[i % BUFSIZE] += data[i];                          \
29     }                                                             \
30     val = local_buf[cost % BUFSIZE];                              \
31     for (int i = 0; i < cost; i++) {                              \
32       val += i;                                                   \
33     }                                                             \
34     *out = val;                                                   \
35   }
36
37 GEN_PES()
```

Listing A.3: The implementation of the standard PE in OpenCL

# Appendix B

## Aggregated Evaluation Results

This section shows some of the aggregated evaluation results. The evaluation data is condensed where needed. For the raw data see the `logs` directory in the RESCH framework's repository.

Table B.1 shows the underlying data for the machine model execution for selected $\log_2$ values of the input size and the mean execution time of the standard PEs in nanoseconds.

Table B.1: Standard PE timings

| Parameter | Platform | Data | time (ns) |
|---|---|---|---|
| Computing | Intel Stratix 10 | 2 | 679 |
| | | 8 | 657 |
| | | 16 | 993944 |
| | | 22 | 64527981 |
| | Xilinx U280 | 2 | 42461 |
| | | 8 | 48271 |
| | | 16 | 1637494 |
| | | 22 | 98014286 |
| Data | Intel Stratix 10 | 2 | 1704 |
| | | 8 | 1679 |
| | | 16 | 207410 |
| | | 22 | 12929069 |
| | Xilinx U280 | 2 | 40095 |
| | | 8 | 39715 |
| | | 16 | 266894 |
| | | 22 | 14119722 |

Table B.2 shows the aggregated results of the REFT evaluation, comparing the optimal solver solution. The data is grouped by the machine model, the benchmark, the algorithm for task graph generation, number of PEs $|P|$ and number of locations $|L|$. It shows the mean execution time of the scheduling algorithm in seconds, the mean makespan $\overline{\mathsf{sl}}$, speedup $\overline{\mathsf{speedup}}$, and slack $\overline{\mathsf{slack}}$.

Table B.2: REFT metrics

| Model | Benchmark | Algorithm | $\lvert P\rvert$ | $\lvert L\rvert$ | avg. time (s) | $\overline{\mathsf{sl}}$ | $\overline{\mathsf{speedup}}$ | $\overline{\mathsf{slr}}$ | $\overline{\mathsf{slack}}$ |
|---|---|---|---|---|---|---|---|---|---|
| $\mathcal{M}_R$ | optimal | Erdős–Rényi | 1 | 1 | 0.010 | 500 | 1.0 | 5.0 | 291.5 |
| | | | 2 | | 0.023 | 278 | 1.7 | 2.8 | 69.3 |
| | | | 3 | | 0.030 | 233 | 2.1 | 2.3 | 24.8 |
| | | layer-by-layer | 1 | | 0.011 | 500 | 1.0 | 5.0 | 364.8 |
| | | | 2 | | 0.022 | 278 | 1.7 | 2.8 | 142.6 |
| | | | 3 | | 0.029 | 200 | 2.4 | 2.0 | 64.8 |
| | | enumeration | 1 | | 0.010 | 500 | 1.0 | 5.0 | 216.1 |
| | | | 2 | | 0.022 | 322 | 1.4 | 3.2 | 38.3 |
| | | | 3 | | 0.029 | 300 | 1.6 | 3.0 | 16.1 |
| | REFT | Erdős–Rényi | 1 | | 0.004 | 500 | 1.0 | 5.0 | 291.5 |
| | | | 2 | | 0.006 | 289 | 1.7 | 2.9 | 80.4 |
| | | | 3 | | 0.007 | 233 | 2.1 | 2.3 | 24.8 |
| | | layer-by-layer | 1 | | 0.004 | 500 | 1.0 | 5.0 | 364.8 |
| | | | 2 | | 0.005 | 278 | 1.7 | 2.8 | 142.6 |
| | | | 3 | | 0.007 | 200 | 2.4 | 2.0 | 64.8 |
| | | enumeration | 1 | | 0.004 | 500 | 1.0 | 5.0 | 216.1 |
| | | | 2 | | 0.006 | 322 | 1.4 | 3.2 | 38.3 |
| | | | 3 | | 0.007 | 300 | 1.6 | 3.0 | 16.1 |
| $\mathcal{M}_{PR}$ | optimal | Erdős–Rényi | 1 | | 0.011 | 500 | 1.0 | 5.0 | 291.5 |
| | | | | 2 | 0.023 | 278 | 1.7 | 2.8 | 69.3 |
| | | | | 3 | 0.029 | 233 | 2.1 | 2.3 | 24.8 |
| | | layer-by-layer | | 1 | 0.010 | 500 | 1.0 | 5.0 | 364.8 |
| | | | | 2 | 0.025 | 278 | 1.7 | 2.8 | 142.6 |
| | | | | 3 | 0.028 | 200 | 2.4 | 2.0 | 64.8 |
| | | enumeration | | 1 | 0.011 | 500 | 1.0 | 5.0 | 216.1 |
| | | | | 2 | 0.022 | 322 | 1.4 | 3.2 | 38.3 |
| | | | | 3 | 0.029 | 300 | 1.6 | 3.0 | 16.1 |
| | REFT | Erdős–Rényi | | 1 | 0.004 | 500 | 1.0 | 5.0 | 291.5 |
| | | | | 2 | 0.006 | 289 | 1.7 | 2.9 | 80.4 |
| | | | | 3 | 0.007 | 233 | 2.1 | 2.3 | 24.8 |
| | | layer-by-layer | | 1 | 0.004 | 500 | 1.0 | 5.0 | 364.8 |
| | | | | 2 | 0.006 | 278 | 1.7 | 2.8 | 142.6 |
| | | | | 3 | 0.006 | 200 | 2.4 | 2.0 | 64.8 |
| | | enumeration | | 1 | 0.004 | 500 | 1.0 | 5.0 | 216.1 |
| | | | | 2 | 0.006 | 322 | 1.4 | 3.2 | 38.3 |
| | | | | 3 | 0.007 | 300 | 1.6 | 3.0 | 16.1 |

Table B.3 shows the aggregated results of the REFT evaluation using the layer-by-layer technique. The data is grouped by the machine model, the benchmark, and the number of locations. It shows the mean execution time of the scheduling algorithm in seconds, the mean makespan $\overline{\mathsf{sl}}$, speedup $\overline{\mathsf{speedup}}$, and slack $\overline{\mathsf{slack}}$.

Table B.3: REFT metrics

| Model | Benchmark | Overhead | $\lvert L \rvert$ | time (s) | $\overline{\mathsf{sl}}$ | $\overline{\mathsf{speedup}}$ | $\overline{\mathsf{slr}}$ | $\overline{\mathsf{slack}}$ |
|---|---|---|---|---|---|---|---|---|
| $\mathcal{M}_{PR}$ | optimal | 0 - 40 | 3 | 0.582 | 417 | 2.4 | 4.2 | 187.0 |
| | | 40 - 80 | | 0.589 | 423 | 2.4 | 4.2 | 193.3 |
| | | 80 - 120 | | 0.594 | 432 | 2.3 | 4.3 | 202.3 |
| | | 120 - 160 | | 0.604 | 448 | 2.3 | 4.5 | 217.4 |
| | | 160 - 200 | | 0.603 | 463 | 2.2 | 4.6 | 232.9 |
| | REFT | 0 - 40 | | 0.013 | 444 | 2.3 | 4.4 | 213.7 |
| | | 40 - 80 | | 0.013 | 485 | 2.1 | 4.8 | 254.8 |
| | | 80 - 120 | | 0.013 | 518 | 2.0 | 5.2 | 287.6 |
| | | 120 - 160 | | 0.013 | 553 | 1.9 | 5.5 | 323.1 |
| | | 160 - 200 | | 0.013 | 594 | 1.7 | 5.9 | 363.5 |
| $\mathcal{M}_{R}$ | optimal | 0 - 40 | 1 | 0.036 | 514 | 2.0 | 5.1 | 284.0 |
| | | 40 - 80 | | 0.036 | 514 | 2.0 | 5.1 | 284.0 |
| | | 80 - 120 | | 0.036 | 514 | 2.0 | 5.1 | 284.0 |
| | | 120 - 160 | | 0.036 | 514 | 2.0 | 5.1 | 284.0 |
| | | 160 - 200 | | 0.036 | 514 | 2.0 | 5.1 | 284.0 |
| | REFT | 0 - 40 | | 0.008 | 538 | 1.9 | 5.4 | 308.1 |
| | | 40 - 80 | | 0.008 | 538 | 1.9 | 5.4 | 308.1 |
| | | 80 - 120 | | 0.008 | 538 | 1.9 | 5.4 | 308.1 |
| | | 120 - 160 | | 0.008 | 538 | 1.9 | 5.4 | 308.1 |
| | | 160 - 200 | | 0.008 | 538 | 1.9 | 5.4 | 308.1 |

Table B.4 shows the aggregated results of the REFT evaluation using the layer-by-layer technique. The data is grouped by the machine model, the benchmark, and the number layers. It shows the mean execution time of the scheduling algorithm in seconds, the mean makespan $\overline{\mathsf{sl}}$, speedup $\overline{\mathsf{speedup}}$, and slack $\overline{\mathsf{slack}}$.

Table B.4: REFT metrics by layer

| Model | Benchmark | Layers | time (s) | $\overline{\text{sl}}$ | $\overline{\text{speedup}}$ | $\overline{\text{slr}}$ | $\overline{\text{slack}}$ |
|---|---|---|---|---|---|---|---|
| $\mathcal{M}_{PR}$ | optimal | 1 | 0.562 | 411 | 2.4 | 4.1 | 311.1 |
| | | 2 | 0.610 | 418 | 2.4 | 4.2 | 228.0 |
| | | 3 | 0.611 | 439 | 2.3 | 4.4 | 186.7 |
| | | 4 | 0.602 | 445 | 2.3 | 4.4 | 174.6 |
| | | 5 | 0.601 | 453 | 2.3 | 4.5 | 172.8 |
| | | 6 | 0.594 | 444 | 2.3 | 4.4 | 178.2 |
| | | 7 | 0.592 | 445 | 2.3 | 4.4 | 197.3 |
| | | 8 | 0.582 | 435 | 2.3 | 4.4 | 201.8 |
| | | 9 | 0.581 | 433 | 2.4 | 4.3 | 199.1 |
| | | 10 | 0.579 | 432 | 2.3 | 4.3 | 205.2 |
| | REFT | 1 | 0.012 | 497 | 2.1 | 5.0 | 396.8 |
| | | 2 | 0.013 | 505 | 2.0 | 5.1 | 314.9 |
| | | 3 | 0.013 | 506 | 2.0 | 5.1 | 253.4 |
| | | 4 | 0.013 | 510 | 2.0 | 5.1 | 240.3 |
| | | 5 | 0.013 | 532 | 1.9 | 5.3 | 251.5 |
| | | 6 | 0.013 | 519 | 2.0 | 5.2 | 254.1 |
| | | 7 | 0.013 | 525 | 2.0 | 5.3 | 277.8 |
| | | 8 | 0.013 | 514 | 2.0 | 5.1 | 281.0 |
| | | 9 | 0.013 | 518 | 2.0 | 5.2 | 283.7 |
| | | 10 | 0.013 | 510 | 2.0 | 5.1 | 282.4 |
| $\mathcal{M}_{R}$ | optimal | 1 | 0.032 | 481 | 2.1 | 4.8 | 381.1 |
| | | 2 | 0.038 | 507 | 2.0 | 5.1 | 316.3 |
| | | 3 | 0.039 | 531 | 1.9 | 5.3 | 278.5 |
| | | 4 | 0.037 | 512 | 2.0 | 5.1 | 242.1 |
| | | 5 | 0.037 | 534 | 1.9 | 5.3 | 254.0 |
| | | 6 | 0.036 | 513 | 2.0 | 5.1 | 248.0 |
| | | 7 | 0.036 | 526 | 2.0 | 5.3 | 278.3 |
| | | 8 | 0.036 | 513 | 2.0 | 5.1 | 279.9 |
| | | 9 | 0.035 | 511 | 2.0 | 5.1 | 277.2 |
| | | 10 | 0.035 | 512 | 2.0 | 5.1 | 284.9 |
| | REFT | 1 | 0.007 | 481 | 2.1 | 4.8 | 381.1 |
| | | 2 | 0.008 | 531 | 1.9 | 5.3 | 340.7 |
| | | 3 | 0.008 | 560 | 1.8 | 5.6 | 307.4 |
| | | 4 | 0.008 | 541 | 1.9 | 5.4 | 271.0 |
| | | 5 | 0.008 | 571 | 1.8 | 5.7 | 290.7 |
| | | 6 | 0.008 | 546 | 1.9 | 5.5 | 280.2 |
| | | 7 | 0.008 | 546 | 1.9 | 5.5 | 298.3 |
| | | 8 | 0.008 | 531 | 1.9 | 5.3 | 297.6 |
| | | 9 | 0.008 | 538 | 1.9 | 5.4 | 303.9 |
| | | 10 | 0.008 | 538 | 1.9 | 5.4 | 310.5 |

Table B.5 shows the aggregated results of the REFT with CCR influence. The data is grouped by the machine model, the benchmark, and the number layers. It shows the mean execution time of the scheduling algorithm in seconds, the mean makespan $\overline{\text{sl}}$, speedup $\overline{\text{speedup}}$, and slack $\overline{\text{slack}}$.

Table B.5: REFT metrics

| Model | CCR | Imbalance | time (s) | $\overline{\text{sl}}$ | $\overline{\text{speedup}}$ | $\overline{\text{slr}}$ | $\overline{\text{slack}}$ |
|---|---|---|---|---|---|---|---|
| $\mathcal{M}_{PR}$ | 0 | 0 | 0.435 | 8095 | 1.7 | 81.0 | 2761.4 |
| | | 30 | 1.066 | 11849 | 1.8 | 165.9 | 3653.7 |
| | | 60 | 1.833 | 15318 | 1.9 | 359.0 | 4233.5 |
| | | 90 | 2.778 | 18693 | 1.9 | 1344.4 | 4627.4 |
| | 50 | 0 | 0.520 | 8950 | 1.7 | 89.5 | 1570.2 |
| | | 30 | 1.140 | 12859 | 1.8 | 180.3 | 1995.7 |
| | | 60 | 1.934 | 16237 | 1.8 | 385.7 | 1900.2 |
| | | 90 | 2.934 | 19558 | 1.9 | 1385.6 | 1477.1 |
| | 100 | 0 | 0.601 | 10567 | 1.5 | 105.7 | 907.2 |
| | | 30 | 1.263 | 14070 | 1.7 | 196.4 | 151.2 |
| | | 60 | 2.060 | 17660 | 1.8 | 410.7 | 0.0 |
| | | 90 | 3.096 | 20862 | 1.8 | 1612.4 | 0.0 |
| | 150 | 0 | 0.689 | 11450 | 1.5 | 114.5 | 0.0 |
| | | 30 | 1.391 | 15519 | 1.6 | 218.5 | 0.0 |
| | | 60 | 2.243 | 19396 | 1.6 | 459.5 | 0.0 |
| | | 90 | 3.318 | 22531 | 1.7 | 1740.8 | 0.0 |
| | 200 | 0 | 0.788 | 12881 | 1.4 | 128.8 | 0.0 |
| | | 30 | 1.527 | 16764 | 1.5 | 234.7 | 0.0 |
| | | 60 | 2.397 | 20809 | 1.6 | 489.5 | 0.0 |
| | | 90 | 3.567 | 24331 | 1.6 | 1880.0 | 0.0 |
| $\mathcal{M}_{R}$ | 0 | 0 | 0.275 | 6501 | 2.3 | 65.0 | 1167.4 |
| | | 30 | 0.586 | 9618 | 2.5 | 134.6 | 1423.1 |
| | | 60 | 1.018 | 12681 | 2.5 | 297.6 | 1596.6 |
| | | 90 | 1.598 | 15763 | 2.5 | 1111.5 | 1697.1 |
| | 50 | 0 | 0.326 | 7169 | 2.4 | 71.7 | 0.0 |
| | | 30 | 0.650 | 10394 | 2.4 | 145.7 | 0.0 |
| | | 60 | 1.108 | 13459 | 2.4 | 319.7 | 0.0 |
| | | 90 | 1.713 | 16427 | 2.5 | 1170.6 | 0.0 |
| | 100 | 0 | 0.378 | 8516 | 2.1 | 85.2 | 0.0 |
| | | 30 | 0.726 | 11496 | 2.3 | 160.4 | 0.0 |
| | | 60 | 1.211 | 14501 | 2.4 | 336.9 | 0.0 |
| | | 90 | 1.838 | 17527 | 2.4 | 1360.7 | 0.0 |
| | 150 | 0 | 0.438 | 9167 | 2.2 | 91.7 | 0.0 |
| | | 30 | 0.810 | 12576 | 2.2 | 177.0 | 0.0 |
| | | 60 | 1.322 | 15877 | 2.3 | 376.7 | 0.0 |
| | | 90 | 1.975 | 18937 | 2.4 | 1463.1 | 0.0 |
| | 200 | 0 | 0.501 | 10354 | 2.1 | 103.5 | 0.0 |
| | | 30 | 0.905 | 13584 | 2.1 | 190.1 | 0.0 |
| | | 60 | 1.444 | 16975 | 2.2 | 398.6 | 0.0 |
| | | 90 | 2.133 | 20284 | 2.3 | 1553.8 | 0.0 |

Table B.6 shows the aggregated results of the REFT with communication influence. The data is grouped by the machine model, the benchmark, and by whether edge scheduling is enabled. It shows the mean execution time of the scheduling algorithm in seconds, the mean makespan $\overline{\mathsf{sl}}$, speedup $\overline{\mathsf{speedup}}$, and slack $\overline{\mathsf{slack}}$.

Table B.6: REFT communication metrics

| Model | Benchmark | Edge Scheduling | time (s) | $\overline{\mathsf{sl}}$ | $\overline{\mathsf{speedup}}$ | $\overline{\mathsf{slr}}$ | $\overline{\mathsf{slack}}$ |
|---|---|---|---|---|---|---|---|
| $\mathcal{M}_{PR}$ | optimal | without | 0.721 | 924 | 1.1 | 9.2 | 0.0 |
| | | with | 295.906 | 1956 | 0.6 | 19.6 | 622.3 |
| | REFT | without | 0.014 | 957 | 1.1 | 9.6 | 0.0 |
| | | with | 0.163 | 3082 | 0.5 | 30.8 | 1748.7 |
| $\mathcal{M}_{R}$ | optimal | without | 0.050 | 941 | 1.1 | 9.4 | 0.0 |
| | | with | 2.812 | 3017 | 0.5 | 30.2 | 1683.5 |
| | REFT | without | 0.008 | 946 | 1.1 | 9.5 | 0.0 |
| | | with | 0.078 | 3052 | 0.5 | 30.5 | 1718.3 |

Table B.7 shows the accumulated results of the InterCO evaluation. The data is grouped by the number of possible configurations $k$, the number of generations in steps of 25. It shows the mean makespan $\overline{\text{sl}}$, speedup $\overline{\text{speedup}}$, Schedule Length Ratio (SLR) $\overline{\text{slr}}$ and slack $\overline{\text{slack}}$.

Table B.7: InterCO metrics

| $k$ | Generation | $\overline{\text{sl}}$ | $\overline{\text{speedup}}$ | $\overline{\text{slr}}$ | $\overline{\text{slack}}$ | n |
|---|---|---|---|---|---|---|
| 2 | 0 - 25 | 3346 | 1.7 | 5.6 | 1591.3 | 24 |
| | 25 - 50 | 3443 | 1.7 | 5.7 | 1688.3 | 14 |
| | 50 - 75 | 3650 | 1.6 | 6.1 | 1895.5 | 2 |
| | 75 - 100 | 3600 | 1.6 | 6.0 | 1845.5 | 3 |
| 3 | 0 - 25 | 2254 | 2.5 | 3.8 | 499.8 | 46 |
| | 25 - 50 | 1890 | 3.0 | 3.2 | 135.9 | 42 |
| | 50 - 75 | 1858 | 3.0 | 3.1 | 103.3 | 38 |
| | 75 - 100 | 2052 | 2.8 | 3.4 | 297.8 | 42 |
| 4 | 0 - 25 | 2016 | 2.8 | 3.4 | 261.8 | 49 |
| | 25 - 50 | 1718 | 3.3 | 2.9 | 0.0 | 49 |
| | 50 - 75 | 1787 | 3.1 | 3.0 | 32.1 | 45 |
| | 75 - 100 | 1695 | 3.3 | 2.8 | 0.0 | 44 |
| 5 | 0 - 25 | 2008 | 2.8 | 3.3 | 253.6 | 49 |
| | 25 - 50 | 1975 | 2.8 | 3.3 | 220.5 | 48 |
| | 50 - 75 | 1815 | 3.1 | 3.0 | 60.0 | 48 |
| | 75 - 100 | 1746 | 3.2 | 2.9 | 0.0 | 50 |
| 10 | 0 - 25 | 1734 | 3.2 | 2.9 | 0.0 | 50 |
| | 25 - 50 | 1662 | 3.4 | 2.8 | 0.0 | 50 |
| | 50 - 75 | 1740 | 3.2 | 2.9 | 0.0 | 50 |
| | 75 - 100 | 1720 | 3.3 | 2.9 | 0.0 | 50 |
| | 0 - 25 | 1888 | 3.0 | 3.1 | 133.5 | 50 |
| | 25 - 50 | 1706 | 3.3 | 2.8 | 0.0 | 50 |
| | 50 - 75 | 1624 | 3.4 | 2.7 | 0.0 | 50 |
| | 75 - 100 | 1720 | 3.3 | 2.9 | 0.0 | 50 |

Table B.8 shows the accumulated results of the IntraCO evaluation based on the structured LU decomposition graph set. The data is grouped by the number of generations in steps of 5 and shows the mean makespan $\overline{\text{sl}}$, speedup $\overline{\text{speedup}}$, slack $\overline{\text{slack}}$ and efficiency $\overline{\text{efficiency}}$. The number $n$ of valid solutions varies per generation, in this case sometimes no valid solution was found.

Table B.8: IntraCO metrics

| Generation | $\overline{\text{sl}}$ | $\overline{\text{speedup}}$ | $\overline{\text{slr}}$ | $\overline{\text{slack}}$ | $\overline{\text{efficiency}}$ | n |
|---|---|---|---|---|---|---|
| 0 - 5 | 7776 | 1.8 | 9.7 | 5256.5 | 1.66 | 13 |
| 5 - 10 | 7755 | 1.8 | 9.7 | 5234.9 | 1.65 | 6 |
| 10 - 15 | 8165 | 1.7 | 10.2 | 5644.7 | 1.43 | 8 |
| 15 - 20 | 8942 | 1.6 | 11.2 | 6421.7 | 0.51 | 2 |
| 20 - 25 | 7497 | 1.9 | 9.4 | 4977.1 | 2.37 | 1 |
| 25 - 30 | 7529 | 1.9 | 9.4 | 5009.0 | 2.30 | 2 |
| 30 - 35 | 9583 | 1.5 | 12.0 | 7063.1 | 0.81 | 4 |
| 35 - 40 | 8544 | 1.7 | 10.7 | 6023.5 | 1.17 | 5 |
| 45 - 50 | 9400 | 1.5 | 11.7 | 6879.9 | 1.79 | 1 |
| 50 - 55 | 8343 | 1.7 | 10.4 | 5822.9 | 2.02 | 1 |
| 55 - 60 | 8629 | 1.6 | 10.8 | 6109.4 | 1.63 | 4 |
| 60 - 65 | 10578 | 1.3 | 13.2 | 8058.2 | 0.43 | 1 |
| 65 - 70 | 9400 | 1.5 | 11.7 | 6879.9 | 1.89 | 1 |
| 75 - 80 | 8876 | 1.6 | 11.1 | 6355.8 | 0.51 | 1 |
| 80 - 85 | 9225 | 1.5 | 11.5 | 6704.9 | 0.49 | 1 |
| 85 - 90 | 7759 | 1.8 | 9.7 | 5239.2 | 2.29 | 1 |
| 95 - 100 | 10182 | 1.4 | 12.7 | 7662.5 | 1.74 | 2 |

Table B.9 shows the accumulated results of the IntraCO evaluation based on randomly generated graphs. The data is grouped by the number of generations in steps of 5 and shows the mean makespan $\overline{\text{sl}}$, speedup $\overline{\text{speedup}}$, slack $\overline{\text{slack}}$ and efficiency $\overline{\text{efficiency}}$. The number $n$ of valid solutions varies per generation.

Table B.9: GA IntraCO metrics

| Generation | $\overline{\text{sl}}$ | $\overline{\text{speedup}}$ | $\overline{\text{slr}}$ | $\overline{\text{slack}}$ | $\overline{\text{efficiency}}$ | n |
|---|---|---|---|---|---|---|
| 0 - 25 | 4406 | 6.5 | 44.1 | 3604.6 | 5.8 | 7455 |
| 25 - 50 | 6820 | 6.2 | 68.2 | 6018.9 | 5.6 | 3590 |
| 50 - 75 | 5641 | 6.1 | 56.4 | 4839.9 | 5.4 | 2432 |
| 75 - 100 | 5524 | 6.1 | 55.2 | 4723.0 | 5.4 | 2008 |

Table B.10 shows the accumulated results of the CO evaluation. The data is grouped by the configuration overhead $\rho$ and the number of configurations $|C|$ and shows the mean makespan $\overline{\text{sl}}$, speedup $\overline{\text{speedup}}$, slack $\overline{\text{slack}}$ and efficiency $\overline{\text{efficiency}}$. The number $n$ of valid solutions varies per generation.

Table B.10: CO metrics

| $\rho$ | $\lvert C\rvert$ | Generation | $\overline{\text{sl}}$ | $\overline{\text{speedup}}$ | $\overline{\text{slr}}$ | $\overline{\text{slack}}$ | $\overline{\text{efficiency}}$ | n |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 - 25 | 3483 | 6.7 | 34.8 | 2964.8 | 6.5 | 588 |
| | | 25 - 50 | 3753 | 6.7 | 37.5 | 3234.4 | 6.3 | 220 |
| | 2 | 0 - 25 | 2237 | 7.8 | 22.4 | 1719.5 | 7.1 | 2427 |
| | | 25 - 50 | 1769 | 8.9 | 17.7 | 1251.4 | 8.0 | 2385 |
| | 3 | 0 - 25 | 2410 | 7.3 | 24.1 | 1892.2 | 7.2 | 2574 |
| | | 25 - 50 | 1982 | 8.3 | 19.8 | 1463.8 | 8.0 | 2595 |
| 100 | 1 | 0 - 25 | 4255 | 6.5 | 42.6 | 3743.1 | 6.8 | 591 |
| | | 25 - 50 | 5701 | 6.6 | 57.0 | 5188.8 | 6.9 | 219 |
| | 2 | 0 - 25 | 2317 | 7.6 | 23.2 | 1796.6 | 7.5 | 2418 |
| | | 25 - 50 | 1885 | 8.8 | 18.9 | 1364.4 | 8.8 | 2333 |
| | 3 | 0 - 25 | 2211 | 8.2 | 22.1 | 1692.5 | 8.3 | 2645 |
| | | 25 - 50 | 1792 | 9.4 | 17.9 | 1273.8 | 9.3 | 2693 |
| 200 | 1 | 0 - 25 | 3840 | 6.8 | 38.4 | 3332.1 | 8.0 | 637 |
| | | 25 - 50 | 3075 | 6.9 | 30.7 | 2566.4 | 8.0 | 184 |
| | 2 | 0 - 25 | 2575 | 7.6 | 25.8 | 2060.1 | 6.8 | 2489 |
| | | 25 - 50 | 1802 | 9.1 | 18.0 | 1286.5 | 7.9 | 2397 |
| | 3 | 0 - 25 | 2271 | 7.9 | 22.7 | 1754.7 | 7.9 | 2639 |
| | | 25 - 50 | 1717 | 8.9 | 17.2 | 1200.1 | 8.7 | 2611 |

# Acronyms

| | | |
|---|---|---|
| **IntraCO** | Intra-Configuration Optimization | xiv |
| **ASIC** | Application Specific Integrated Circuit | 2 |
| **BRAM** | Block Random Access Memory | 15 |
| **CCR** | Computation-Communication-Ratio | 118 |
| **CI** | Computational Intensity | 82 |
| **CO** | Configuration Optimization | xiv |
| **CPU** | Central Processing Unit | 1 |
| **CXL** | Compute Express Link | 39 |
| **DAG** | Directed Acyclic Graph | 5 |
| **DDR SDRAM** | Double Data Rate Synchronous Dynamic Random-Access Memory | 17 |
| **DR** | Dynamic Reconfiguration | 4 |
| **DSE** | Design Space Exploration | 8 |
| **DSP** | Digital Signal Processor | 1 |
| **EFT** | Earliest Finish Time | 66 |
| **FIFO** | First In First Out | 18 |
| **FPGA** | Field Programmable Gate Array | 2 |
| **GA** | Genetic Algorithm | 23 |
| **GPU** | Graphics Processing Unit | 1 |
| **HEFT** | Heterogeneous Earliest Finish Time | 66 |
| **HLS** | High-Level Synthesis | 3 |
| **HPC** | High Performance Computing | 1 |
| **InterCO** | Inter-Configuration Optimization | xiv |
| **IC** | Integrated Circuit | 1 |
| **ILP** | Integer Linear Programming | 6 |
| **I/O** | Input/Output | 2 |
| **IR** | Intermediate Representation | 21 |
| **JSON** | JavaScript Object Notation | 104 |
| **LB** | Logic Block | 15 |
| **LUT** | Lookup Table | 4 |
| **OOO** | Out-Of-Order | 111 |
| **OS** | Operating System | 18 |

| | | |
|---|---|---|
| **PCIe** | PCI Express | 16 |
| **PE** | Processing Element | xv |
| **PR** | Partial Reconfiguration | 4 |
| **REFT** | Reconfigurable Earliest Finish Time | 66 |
| **RQ** | Research Question | 7 |
| **RTL** | Register-Transfer Level | 4 |
| **SIMD** | Single Instruction, Multiple Data | 82 |
| **SRAM** | Static Random-Access Memory | 16 |
| **SRQ** | Sub-Research Question | 7 |
| **TPU** | Tensor Processing Unit | 2 |

# Bibliography


[1] G. E. Moore, *Cramming more components onto integrated circuits*, McGraw-Hill New York, 1965.

[2] R. H. Dennard, F. H. Gaensslen, H.-N. Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions," *IEEE Journal of solid-state circuits*, vol. 9, no. 5, pp. 256–268, 1974. DOI: 10.1109/jssc.1974.1050511.

[3] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2011, pp. 365–376. DOI: 10.1145/2000064.2000108.

[4] "TOP500." (2023), [Online]. Available: https://www.top500.org/lists/top500/ (visited on 16.12.2023).

[5] J. J. Dongarra, P. Luszczek, and A. Petitet, "The LINPACK benchmark: Past, present and future," *Concurrency and Computation: Practice and Experience*, vol. 15, no. 9, pp. 803–820, 2003. DOI: 10.1002/cpe.728.

[6] M. Qasaimeh, K. Denolf, J. Lo, K. Vissers, J. Zambreno, and P. H. Jones, "Comparing energy efficiency of CPU, GPU and FPGA implementations for vision kernels," in *2019 IEEE international conference on embedded software and systems (ICESS)*, IEEE, 2019, pp. 1–8. DOI: 10.1109/icess.2019.8782524.

[7] S. Sharma, C.-H. Hsu, and W.-c. Feng, "Making a case for a Green500 list," in *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*, Apr. 2006. DOI: 10.1109/IPDPS.2006.1639600.

[8] "Green500." (2022), [Online]. Available: https://www.top500.org/lists/green500/ (visited on 16.2.2022).

[9] "PFN's Supercomputers," PFN's Supercomputers. (2023), [Online]. Available: https://projects.preferred.jp/supercomputers/ (visited on 8.5.2023).

[10] N. P. Jouppi, C. Young, N. Patil, *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th annual international symposium on computer architecture*, 2017, pp. 1–12.

[11] B. Betkaoui, D. B. Thomas, and W. Luk, "Comparing performance and energy efficiency of FPGAs and GPUs for high productivity computing," in *2010 International Conference on Field-Programmable Technology*, IEEE, 2010, pp. 94–101. DOI: 10.1109/fpt.2010.5681761.

[12] "Intel® Stratix® 10 GX 10M FPGA - Product Specifications," Intel. (Mar. 17, 2022), [Online]. Available: https://www.intel.com/content/www/us/en/products/sku/210290/intel-stratix-10-gx-10m-fpga/specifications.html (visited on 17.3.2022).

[13] S. M. S. Trimberger, "Three ages of FPGAs: A retrospective on the first thirty years of FPGA technology," *IEEE Solid-State Circuits Magazine*, vol. 10, no. 2, pp. 16–29, 2018. DOI: 10.1109/MSSC.2018.2822862.

[14] L. Gan, H. Fu, W. Luk, *et al.*, "Solving the global atmospheric equations through heterogeneous reconfigurable platforms," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 8, no. 2, pp. 1–16, 2015. DOI: 10.1145/2629581.

[15] O. Lindtjorn, R. Clapp, O. Pell, H. Fu, M. Flynn, and O. Mencer, "Beyond traditional microprocessors for geoscience high-performance computing applications," *IEEE Micro*, vol. 31, no. 2, pp. 41–49, 2011. DOI: 10.1109/mm.2011.17.

[16] S. Zhou, C. Chelmis, and V. K. Prasanna, "High-throughput and energy-efficient graph processing on FPGA," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Washington, DC, USA: IEEE, May 2016, pp. 103–110. DOI: 10.1109/FCCM.2016.35.

[17] T. De Matteis, J. de Fine Licht, and T. Hoefler, "FBLAS: Streaming linear algebra on FPGA," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, 2020, pp. 1–13. DOI: 10.1109/sc41405.2020.00063.

[18] K. Vipin and S. A. Fahmy, "FPGA dynamic and partial reconfiguration: A survey of architectures, methods, and applications," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–39, 2018. DOI: 10.1145/3193827.

[19] "Gartner says worldwide semiconductor revenue grew 25.1% in 2021, exceeding $500 billion for the first time," Gartner. (2022), [Online]. Available: https://www.gartner.com/en/newsroom/press-releases/2022-01-19-gartner-says-worldwide-semiconductor-revenue-grew-25-point-one-percent-in-2021-exceeding-500-billion-for-the-first-time (visited on 17.2.2022).

[20] D. Clark, "Intel completes acquisition of Altera," *Wall Street JournalTech*, Dec. 29, 2015.

[21] D. Clark, "AMD agrees to buy Xilinx for $35 billion in stock," *The New York TimesTechnology*, Oct. 27, 2020.

[22] "AMD completes acquisition of Xilinx." (2022), [Online]. Available: https://www.amd.com/en/press-releases/2022-02-14-amd-completes-acquisition-xilinx (visited on 17.2.2022).

[23] G. Martin and G. Smith, "High-level synthesis: Past, present, and future," *IEEE Design Test of Computers*, vol. 26, no. 4, pp. 18–25, Jul. 2009. DOI: 10.1109/MDT.2009.83.

[24] R. Nane, V.-M. Sima, C. Pilato, *et al.*, "A survey and evaluation of FPGA high-level synthesis tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, Oct. 2016. DOI: 10.1109/TCAD.2015.2513673.

[25] "Vitis high-level synthesis user guide." (2023), [Online]. Available: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Streaming-Data-Paradigm (visited on 30.1.2023).

[26] L. Wirbel, "Xilinx SDAccel: A unified development environment for tomorrow's data center," *The Linley Group Inc*, 2014.

[27] O. Sinnen, *Task scheduling for parallel systems* (Wiley Series on Parallel and Distributed Computing), A. Y. Zomaya, red. Hoboken, NJ, USA: John Wiley & Sons, Inc., Apr. 20, 2007. DOI: 10.1002/0470121173.

[28] K. D. Pham, A. Vaishnav, M. Vesper, and D. Koch, "ZUCL: A ZYNQ UltraScale+ framework for OpenCL HLS applications," p. 10, 2018.

[29] F. Redaelli, M. Santambrogio, and S. O. Memik, "An ILP formulation for the task graph scheduling problem tailored to bi-dimensional reconfigurable architectures," in *2008 International Conference on Reconfigurable Computing and FPGAs*, Dec. 2008, pp. 97–102. DOI: 10.1109/ReConFig.2008.42.

[30] D. Korolija, T. Roscoe, and G. Alonso, "Do OS abstractions make sense on FPGAs?" *USENIX Symposium on Operating Systems Design and Implementation*, vol. 14, pp. 991–1010,

[31] K. Vipin and S. A. Fahmy, "DyRACT: A partial reconfiguration enabled accelerator and test platform," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2014, pp. 1–7. DOI: 10.1109/FPL.2014.6927507.

[32] K. Krommydas, W.-c. Feng, C. D. Antonopoulos, and N. Bellas, "Opendwarfs: Characterization of dwarf-based benchmarks on fixed and reconfigurable architectures," *Journal of Signal Processing Systems*, vol. 85, no. 3, pp. 373–392, 2016. DOI: 10.1007/s11265-015-1051-z.

[33] P. Jungblut and D. Kranzlmüller, "Optimal schedules for high-level programming environments on FPGAs with constraint programming," in *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, IEEE, 2022, pp. 96–99. DOI: 10.1109/ipdpsw55747.2022.00025.

[34] P. Jungblut and D. Kranzlmüller, "Dynamic spatial multiplexing on FPGAs with OpenCL," in *International Symposium on Applied Reconfigurable Computing*, Rennes, France: Springer, 2021, pp. 265–274. DOI: 10.1007/978-3-030-79025-7_19.

[35] P. Jungblut, "Task scheduling in reconfigurable computing with OpenCL," in *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, New Orleans, LA, USA, Jun. 2021, p. 1023. DOI: 10.1109/IPDPSW52791.2021.00160.

[36] P. Jungblut, "Task scheduling on FPGA-based accelerators without partial reconfiguration," presented at the International Conference on High Performance Computing, Networking, Storage and Analysis (SC) (Dallas, TX, USA), Nov. 17, 2022.

[37] P. Jungblut and K. Fürlinger, "Integrating node-level parallelism abstractions into the PGAS model," in *Proceedings of the 13th International Symposium on High-Level Parallel Programming and Applications*, vol. 13, Porto, Portugal, 2020, pp. 38–56.

[38] K. Fürlinger, J. Gracia, A. Knüpfer, *et al.*, "DASH: Distributed data structures and parallel algorithms in a global address space," in *Software for Exascale Computing-SPPEXA 2016-2019*, Springer, 2020, pp. 103–142.

[39] E. Zenker, B. Worpitz, R. Widera, *et al.*, "Alpaka–An Abstraction Library for Parallel Kernel Acceleration," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, IEEE, 2016, pp. 631–640. DOI: 10.1109/ipdpsw.2016.50.

[40] P. Jungblut and K. Fürlinger, "Portable node-level parallelism for the PGAS model," *International Journal of Parallel Programming*, vol. 49, no. 6, pp. 867–885, Jun. 5, 2021. DOI: 10.1007/s10766-021-00718-x.

[41] P. Jungblut, R. Kowalewski, and K. Fürlinger, "Source-to-source instrumentation for profiling runtime behavior of C++ containers," in *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, Exeter, UK: IEEE, 2018, pp. 948–953. DOI: 10.1109/hpcc/smartcity/dss.2018.00157.

[42] R. Kowalewski, P. Jungblut, and K. Fürlinger, "Engineering a distributed histogram sort," in *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, Albuquerque, NM, USA: IEEE, 2019, pp. 1–11. DOI: 10.1109/cluster.2019.8891005.

[43] J. Von Neumann, "First draft of a report on the EDVAC," *IEEE Annals of the History of Computing*, vol. 15, no. 4, pp. 27–75, 1993. DOI: 10.1109/85.238389.

[44] E. Lubbers and M. Platzner, "ReconOS: An RTOS supporting hard-and software threads," in *2007 International Conference on Field Programmable Logic and Applications*, IEEE, 2007, pp. 441–446.

[45] W. Peck, E. Anderson, J. Agron, J. Stevens, F. Baijot, and D. Andrews, "Hthreads: A computational model for reconfigurable devices," in *2006 International Conference on Field Programmable Logic and Applications*, IEEE, 2006, pp. 1–4. DOI: 10.1109/FPL.2006.311336.

[46] M. Asiatici, N. George, K. Vipin, S. A. Fahmy, and P. Ienne, "Virtualized execution runtime for FPGA accelerators in the cloud," *IEEE Access*, vol. 5, pp. 1900–1910, 2017. DOI: 10.1109/ACCESS.2017.2661582.

[47] A. Vaishnav, K. D. Pham, D. Koch, and J. Garside, "Resource elastic virtualization for FPGAs using OpenCL," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, Dublin, Ireland: IEEE, Aug. 2018, pp. 111–1117. DOI: 10.1109/FPL.2018.00028.

[48] T. Xia, J.-C. Prevotet, and F. Nouvel, "Hypervisor mechanisms to manage FPGA reconfigurable accelerators," in *2016 International Conference on Field-Programmable Technology (FPT)*, Xi'an, China: IEEE, Dec. 2016, pp. 44–52. DOI: 10.1109/FPT.2016.7929187.

[49] F. Benz, A. Seffrin, and S. A. Huss, "Bil: A tool-chain for bitstream reverse-engineering," in *22nd International Conference on Field Programmable Logic and Applications (FPL)*, IEEE, 2012, pp. 735–738. DOI: 10.1109/fpl.2012.6339165.

[50] K. D. Pham, E. Horta, and D. Koch, "BITMAN: A tool and API for FPGA bitstream manipulations," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, IEEE, 2017, pp. 894–897. DOI: 10.23919/date.2017.7927114.

[51] S. Hauck and A. DeHon, Eds., *Reconfigurable computing: the theory and practice of FPGA-based computation* (The Morgan Kaufmann series in systems on silicon). Amsterdam ; Boston: Morgan Kaufmann, 2008, 908 pp.

[52] R. Stefan and S. D. Cotofana, "Bitstream compression techniques for Virtex 4 FPGAs," in *2008 International Conference on Field Programmable Logic and Applications*, Sep. 2008, pp. 323–328. DOI: 10.1109/FPL.2008.4629952.

[53] "Configuration bit stream sizes," Intel. (Apr. 27, 2022), [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683181/current/configuration-bit-stream-sizes.html (visited on 27.4.2022).

[54] K. Papadimitriou, A. Dollas, and S. Hauck, "Performance of partial reconfiguration in FPGA systems: A survey and a cost model," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 4, no. 4, pp. 1–24, Dec. 1, 2011. DOI: 10.1145/2068716.2068722.

[55] *Intel Stratix 10 Configuration User Guide*.

[56] C. Beckhoff, D. Koch, and J. Torresen, "Portable module relocation and bitstream compression for Xilinx FPGAs," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, IEEE, 2014, pp. 1–8. DOI: 10.1109/fpl.2014.6927480.

[57] H. Gu and S. Chen, "Partial reconfiguration bitstream compression for Virtex FPGAs," in *2008 Congress on Image and Signal Processing*, vol. 5, IEEE, 2008, pp. 183–185. DOI: 10.1109/cisp.2008.253.

[58] R. Jia, F. Wang, R. Chen, X.-G. Wang, and H.-G. Yang, "JTAG-based bitstream compression for FPGA configuration," in *2012 IEEE 11th International Conference on Solid-State and Integrated Circuit Technology*, IEEE, 2012, pp. 1–3. DOI: 10.1109/icsict.2012.6467807.

[59] J. H. Pan, T. Mitra, and W.-F. Wong, "Configuration bitstream compression for dynamically reconfigurable FPGAs," in *IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004.*, IEEE, 2004, pp. 766–773. DOI: 10.1109/iccad.2004.1382679.

[60] A. Morales-Villanueva, R. Kumar, and A. Gordon-Ross, "Configuration prefetching and reuse for preemptive hardware multitasking on partially reconfigurable FPGAs," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2016, pp. 1505–1508. DOI: 10.3850/9783981537079_0486.

[61] F. Redaelli, M. D. Santambrogio, and D. Sciuto, "Task scheduling with configuration prefetching and anti-fragmentation techniques on dynamically reconfigurable systems," in *2008 Design, Automation and Test in Europe*, IEEE, 2008, pp. 519–522. DOI: 10.1109/date.2008.4484902.

[62] S. Corbetta, M. Morandi, M. Novati, M. D. Santambrogio, D. Sciuto, and P. Spoletini, "Internal and external bitstream relocation for partial dynamic reconfiguration," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 17, no. 11, pp. 1650–1654, 2009. DOI: 10.1109/tvlsi.2008.2005670.

[63] H. Kalte, G. Lee, M. Porrmann, and U. Ruckert, "Replica: A bitstream manipulation filter for module relocation in partial reconfigurable systems," in *19th IEEE International Parallel and Distributed Processing Symposium*, IEEE, 2005, 8–pp. DOI: 10.1109/ipdps.2005.380.

[64] K. Manev, J. Powell, K. Matas, and D. Koch, "Byteman: A bitstream manipulation framework," in *2022 International Conference on Field-Programmable Technology (ICFPT)*, IEEE, 2022, pp. 1–9. DOI: 10.1109/icfpt56656.2022.9974549.

[65] S. Trimberger, D. Carberry, A. Johnson, and J. Wong, "A time-multiplexed FPGA," in *Proceedings. The 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines Cat. No. 97TB100186)*, IEEE, 1997, pp. 22–28.

[66] W. Chong, S. Ogata, M. Hariyama, and M. Kameyama, "Architecture of a multi-context FPGA using reconfigurable context memory," in *19th IEEE International Parallel and Distributed Processing Symposium*, IEEE, 2005, 7–pp. DOI: 10.1109/ipdps.2005.112.

[67] K. Vipin and S. A. Fahmy, "Automated partial reconfiguration design for adaptive systems with CoPR for Zynq," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, IEEE, 2014, pp. 202–205. DOI: 10.1109/fccm.2014.63.

[68] J. Corbet, A. Rubini, G. Kroah-Hartman, and A. Rubini, *Linux device drivers*, 3rd ed. Beijing ; Sebastopol, CA: O'Reilly, 2005, 615 pp.

[69] P. Liu, J. Wu, and Y. Wang, "Hybrid algorithms for hardware/software partitioning and scheduling on reconfigurable devices," *Mathematical and Computer Modelling*, vol. 58, no. 1-2, pp. 409–420, 2013. DOI: 10.1016/j.mcm.2012.11.001.

[70] Q. Tang, B. Guo, and Z. Wang, "SW/HW partitioning and scheduling on region-based dynamic partial reconfigurable system-on-chip," *Electronics*, vol. 9, no. 9, p. 1362, 2020. DOI: 10.3390/electronics9091362.

[71] "IEEE Standard for VHDL Language Reference Manual," *IEEE Std 1076-2019*, pp. 1–673, Dec. 2019. DOI: 10.1109/IEEESTD.2019.8938196.

[72] D. Andrews, W. Peck, J. Agron, *et al.*, "Hthreads: A hardware/software co-designed multithreaded RTOS kernel," in *2005 IEEE Conference on Emerging Technologies and Factory Automation*, vol. 2, Sep. 2005, 8 pp.–338. DOI: 10.1109/ETFA.2005.1612697.

[73] M. Happe, A. Traber, and A. Keller, "Preemptive hardware multitasking in ReconOS," in *Applied Reconfigurable Computing*, K. Sano, D. Soudris, M. Hübner, and P. C. Diniz, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2015, pp. 79–90. DOI: 10.1007/978-3-319-16214-0_7.

[74] A. Vaishnav, K. D. Pham, J. Powell, and D. Koch, "FOS: A modular FPGA operating system for dynamic workloads," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 13, no. 4, pp. 1–28, Jan. 26, 2020. DOI: 10.1145/3405794.

[75] F. B. Muslim, L. Ma, M. Roozmeh, and L. Lavagno, "Efficient FPGA implementation of OpenCL high-performance computing applications via high-level synthesis," *IEEE Access*, vol. 5, pp. 2747–2762, 2017. DOI: 10.1109/ACCESS.2017.2671881.

[76] L. Struyf, S. De Beugher, D. H. Van Uytsel, F. Kanters, and T. Goedemé, "The battle of the giants-a case study of GPU vs FPGA optimisation for real-time image processing," in *International Conference on Pervasive and Embedded Computing and Communication Systems*, vol. 2, SCITEPRESS, 2014, pp. 112–119. DOI: 10.5220/0004730301120119.

[77] T. Nguyen, C. MacLean, M. Siracusa, D. Doerfler, N. J. Wright, and S. Williams, "FPGA-based HPC accelerators: An evaluation on performance and energy efficiency," *Concurrency and Computation: Practice and Experience*, vol. 34, no. 20, Sep. 10, 2022. DOI: 10.1002/cpe.6570.

[78] J. Cong, Z. Fang, M. Lo, H. Wang, J. Xu, and S. Zhang, "Understanding performance differences of FPGAs and GPUs," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, IEEE, 2018, pp. 93–96. DOI: 10.1109/fccm.2018.00023.

[79] S. Che, M. Boyer, J. Meng, *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE international symposium on workload characterization (IISWC)*, Ieee, 2009, pp. 44–54. DOI: 10.1109/iiswc.2009.5306797.

[80] R. Maestre, M. Fernandez, R. Hermida, and N. Bagherzadeh, "A framework for scheduling and context allocation in reconfigurable computing," in *Proceedings 12th International Symposium on System Synthesis*, Nov. 1999, pp. 134–140. DOI: 10.1109/ISSS.1999.814272.

[81] S. Trimberger, "Scheduling designs into a time-multiplexed FPGA," in *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays - FPGA '98*, Monterey, California, United States: ACM Press, 1998, pp. 153–160. DOI: 10.1145/275107.275135.

[82] K. Bazargan, R. Kastner, and M. Sarrafzadeh, "Fast template placement for reconfigurable computing systems," *IEEE Design & Test of Computers*, vol. 17, no. 1, pp. 68–83, 2000. DOI: 10.1109/54.825678.

[83] O. Diessel, H. ElGindy, M. Middendorf, H. Schmeck, and B. Schmidt, "Dynamic scheduling of tasks on partially reconfigurable FPGAs," *IEE Proceedings-Computers and Digital Techniques*, vol. 147, no. 3, pp. 181–188, 2000. DOI: 10.1049/ip-cdt:20000485.

[84] H. Simmler, L. Levinson, and R. Männer, "Multitasking on FPGA coprocessors," in *Field-Programmable Logic and Applications: The Roadmap to Reconfigurable Computing*, R. W. Hartenstein and H. Grünbacher, Eds., red. by G. Goos, J. Hartmanis, and J. van Leeuwen, vol. 1896, Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 121–130. DOI: 10.1007/3-540-44614-1_13.

[85] R. Maestre, F. Kurdahi, M. Fernandez, R. Hermida, N. Bagherzadeh, and H. Singh, "A framework for reconfigurable computing: Task scheduling and context management," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 9, no. 6, pp. 858–873, Dec. 2001. DOI: 10.1109/92.974899.

[86] J. Noguera and R. M. Badia, "Multitasking on reconfigurable architectures: Microarchitecture support and dynamic scheduling," *ACM Transactions on Embedded Computing Systems*, vol. 3, no. 2, pp. 385–406, May 2004. DOI: 10.1145/993396.993404.

[87] J. Resano and D. Mozos, "Specific scheduling support to minimize the reconfiguration overhead of dynamically reconfigurable hardware," in *Proceedings of the 41st annual conference on Design automation - DAC '04*, San Diego, CA, USA: ACM Press, 2004, p. 119. DOI: 10.1145/996566.996604.

[88] A. Sudarsanam, M. Srinivasan, and S. Panchanathan, "Resource estimation and task scheduling for multithreaded reconfigurable architectures," in *Proceedings. Tenth International Conference on Parallel and Distributed Systems, 2004. ICPADS 2004.*, 2004, pp. 323–330. DOI: 10.1109/ICPADS.2004.1316111.

[89] Y. Qu, J.-p. Soininen, and J. Nurmi, "Using constraint programming to achieve optimal prefetch scheduling for dependent tasks on run-time reconfigurable devices," in *2006 International Symposium on System-on-Chip*, 2006, pp. 1–4. DOI: 10.1109/ISSOC.2006.321973.

[90] Z. Pan and B. E. Wells, "Hardware supported task scheduling on dynamically reconfigurable SoC architectures," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, no. 11, pp. 1465–1474, Nov. 2008. DOI: 10.1109/TVLSI.2008.2000974.

[91] F. Redaelli, M. D. Santambrogio, and D. Sciuto, "Task scheduling with configuration prefetching and anti-fragmentation techniques on dynamically reconfigurable systems," in *2008 Design, Automation and Test in Europe*, 2008, pp. 519–522. DOI: 10.1109/DATE.2008.4484902.

[92] R. Cordone, F. Redaelli, M. A. Redaelli, M. D. Santambrogio, and D. Sciuto, "Partitioning and scheduling of task graphs on partially dynamically reconfigurable FPGAs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 5, pp. 662–675, May 2009. DOI: 10.1109/TCAD.2009.2015739.

[93] Y. Lu, T. Marconi, K. Bertels, and G. Gaydadjiev, "Online task scheduling for the FPGA-based partially reconfigurable systems," in *International Workshop on Applied Reconfigurable Computing*, Springer, 2009, pp. 216–230. DOI: 10.1007/978-3-642-00641-8_22.

[94] J. Teller and F. Ozguner, "Scheduling tasks on reconfigurable hardware with a list scheduler," in *2009 IEEE International Symposium on Parallel Distributed Processing*, May 2009, pp. 1–4. DOI: 10.1109/IPDPS.2009.5161222.

[95] M. Fazlali, M. Sabeghi, A. Zakerolhosseini, and K. Bertels, "Efficient task scheduling for runtime reconfigurable systems," *Journal of Systems Architecture*, vol. 56, no. 11, pp. 623–632, 2010. DOI: 10.1016/j.sysarc.2010.07.016.

[96] F. Redaelli, M. Santambrogio, D. Sciuto, and S. O. Memik, "Reconfiguration aware task scheduling for multi-FPGA systems," *Reconfigurable Computing*, p. 57, 2010.

[97] J. A. Clemente, D. Mozos, and J. Resano, "A replacement technique to maximize task reuse in reconfigurable systems," in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IEEE, 2011, pp. 250–257. DOI: 10.1109/ipdps.2011.149.

[98] Y. M. Lam, "Integrated task clustering, mapping and scheduling for heterogeneous computing systems," *International Journal of Computer Science & Information Technology*, vol. 4, no. 1, p. 127, 2012. DOI: 10.5121/ijcsit.2012.4111.

[99] M. F. Nadeem, I. Ashraf, S. A. Ostadzadeh, S. Wong, and K. Bertels, "Task scheduling in large-scale distributed systems utilizing partial reconfigurable processing elements," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*, May 2012, pp. 79–90. DOI: 10.1109/IPDPSW.2012.6.

[100] J. A. Clemente, I. Beretta, V. Rana, D. Atienza, and D. Sciuto, "A mapping-scheduling algorithm for hardware acceleration on reconfigurable platforms," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 7, no. 2, pp. 1–27, 2014. DOI: 10.1145/2611562.

[101] Y. Ma, J. Liu, C. Zhang, and W. Luk, "HW/SW partitioning for region-based dynamic partial reconfigurable FPGAs," in *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, IEEE, 2014, pp. 470–476. DOI: 10.1109/iccd.2014.6974721.

[102] Y. Sheng, Y. Liu, R. Li, and X. Xiao, "A communication-aware scheduling algorithm for hardware task scheduling model on FPGA-based reconfigurable systems," *Journal of Computers*, vol. 9, no. 11, pp. 2552–2558, Nov. 1, 2014. DOI: 10.4304/jcp.9.11.2552-2558.

[103] G. Wassi, M. E. A. Benkhelifa, G. Lawday, F. Verdier, and S. Garcia, "Multi-shape tasks scheduling for online multitasking on FPGAs," in *2014 9th International Symposium on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, May 2014, pp. 1–7. DOI: 10.1109/ReCoSoC.2014.6861366.

[104] G. Charitopoulos, I. Koidis, K. Papadimitriou, and D. Pnevmatikatos, "Hardware task Scheduling for partially reconfigurable FPGAs," in *Applied Reconfigurable Computing*, K. Sano, D. Soudris, M. Hübner, and P. C. Diniz, Eds., vol. 9040, Cham: Springer International Publishing, 2015, pp. 487–498. DOI: 10.1007/978-3-319-16214-0_45.

[105] D. Pnevmatikatos, K. Papadimitriou, T. Becker, *et al.*, "FASTER: Facilitating analysis and synthesis technologies for effective reconfiguration," *Microprocessors and Microsystems*, vol. 39, no. 4-5, pp. 321–338, Jun. 2015. DOI: 10.1016/j.micpro.2014.09.006.

[106] E. M. Abdali, M. Pelcat, F. Berry, J.-P. Diguet, and D. Heller, "Task clustering approach to optimize the scheduling on a partially dynamically reconfigurable FPGAs for image processing algorithms," in *Proceedings of the 10th International Conference on Distributed Smart Camera*, 2016, pp. 230–231. DOI: 10.1145/2967413.2974042.

[107] X. Xu, Q. Xu, J. Huang, and S. Chen, "An integrated optimization framework for partitioning, scheduling and floorplanning on partially dynamically reconfigurable FPGAs," in *Proceedings of the on Great Lakes Symposium on VLSI 2017*, 2017, pp. 403–406. DOI: 10.1145/3060403.3060447.

[108] S. Chen, J. Huang, X. Xu, B. Ding, and Q. Xu, "Integrated optimization of partitioning, scheduling, and floorplanning for partially dynamically reconfigurable systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 1, pp. 199–212, 2018. DOI: 10.1145/3060403.3060447.

[109] A. Yoosefi and H. R. Naji, "A clustering algorithm for communication-aware scheduling of task graphs on multi-core reconfigurable systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 10, pp. 2718–2732, Oct. 1, 2017. DOI: 10.1109/TPDS.2017.2703123.

[110] B. da Silva, A. Braeken, F. Domínguez, and A. Touhafi, "Exploiting partial reconfiguration through PCIe for a microphone array network emulator," *International Journal of Reconfigurable Computing*, vol. 2018, 2018. DOI: 10.1155/2018/3214679.

[111] A. Khawaja, J. Landgraf, R. Prakash, M. Wei, E. Schkufza, and C. J. Rossbach, "Sharing, Protection, and Compatibility for Reconfigurable Fabric with AmorphOS," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 107–127.

[112] A. Dhar, M. Yu, W. Zuo, X. Wang, N. S. Kim, and D. Chen, "Leveraging Dynamic Partial Reconfiguration with Scalable ILP Based Task Scheduling," in *2020 33rd International Conference on VLSI Design and 2020 19th International Conference on Embedded Systems (VLSID)*, Jan. 2020, pp. 201–206. DOI: 10.1109/VLSID49098.2020.00052.

[113] Q. Tang, Z. Wang, B. Guo, L.-H. Zhu, and J.-B. Wei, "Partitioning and scheduling with module merging on dynamic partial reconfigurable FPGAs," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 13, no. 3, pp. 1–24, 2020. DOI: 10.1145/3403702.

[114] Z. Wang, Q. Tang, B. Guo, J.-B. Wei, and L. Wang, "Resource partitioning and application scheduling with module merging on dynamically and partially reconfigurable FPGAs," *Electronics*, vol. 9, no. 9, p. 1461, 2020. DOI: 10.3390/electronics9091461.

[115] Q. Jiang, J. Xu, and Y. Chen, "A genetic algorithm for scheduling in heterogeneous multicore system integrated with FPGA," in *2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*, New York City, NY, USA: IEEE, Sep. 2021, pp. 594–602. DOI: 10.1109/ISPA-BDCloud-SocialCom-SustainCom52081.2021.00087.

[116] R. Ramezani, "Dynamic scheduling of task graphs in multi-FPGA systems using critical path," *The Journal of Supercomputing*, vol. 77, no. 1, pp. 597–618, Jan. 2021. DOI: 10.1007/s11227-020-03281-3.

[117] J. Xu, H. Shi, and Y. Chen, "Efficient tasks scheduling in multicore systems integrated with hardware accelerators," *The Journal of Supercomputing*, vol. 79, no. 7, pp. 7244–7271, 2023. DOI: 10.1007/s11227-022-04955-w.

[118] J. Teich, S. P. Fekete, and J. Schepers, "Optimization of dynamic hardware reconfigurations," *The Journal of Supercomputing*, vol. 19, no. 1, pp. 57–75, 2001. DOI: 10.1023/A:1011188411132.

[119] J. Oppermann, M. Reuter-Oppermann, A. Koch, and O. Sinnen, "'Optimising' high-level synthesis in CIRCT," 2022.

[120] M. Urbach and M. B. Petersen, "HLS from PyTorch to System Verilog with MLIR and CIRCT," *Latte'22*, 2022.

[121] M. Mitchell, *An Introduction to Genetic Algorithms*. Mar. 2, 1998. DOI: 10.7551/mitpress/3927.001.0001.

[122] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications*, 1978. DOI: 10.1145/359545.359563.

[123] D. A. Padua, Ed., *Encyclopedia of parallel computing* (Springer reference). New York, NY: Springer, 2011.

[124] Compute Express Link Consortium, *Compute express link (CXL) specification 3.0*, Available online at: https://www.computeexpresslink.org/, 2022.

[125] O. Sinnen and L. Sousa, "Communication contention in task scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 16, no. 6, pp. 503–515, Jun. 2005. DOI: 10.1109/TPDS.2005.64.

[126] "FPGA optimization guide for Intel® oneAPI toolkits." (2023), [Online]. Available: https://www.intel.com/content/www/us/en/develop/documentation/oneapi-fpga-optimization-guide/top/optimize-your-design/throughput-1/pipes.html (visited on 30.1.2023).

[127] Khronos OpenCL Working Group, "The OpenCL C specification, version 2.0," Specification, 2015.

[128] D. G. Feitelson, *Workload modeling for computer systems performance evaluation*. Cambridge: Cambridge University Press, 2015. DOI: 10.1017/CBO9781139939690.

[129] L. Perron and V. Furnon, *OR-Tools*, version 9.3, Google, Mar. 15, 2022.

[130] J. D. Ullman, "NP-complete scheduling problems," *Journal of Computer and System sciences*, vol. 10, no. 3, pp. 384–393, 1975. DOI: 10.1016/s0022-0000(75)80008-0.

[131] V. Boudet, "Heterogeneous task scheduling: A survey," *Research Report RR-6895*, 2001.

[132] H. Oh and S. Ha, "A static scheduling heuristic for heterogeneous processors," in *European Conference on Parallel Processing*, Springer, 1996, pp. 573–577. DOI: 10.1007/bfb0024750.

[133] H. Topcuoglu, S. Hariri, and Min-You Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 3, pp. 260–274, Mar. 2002. DOI: 10.1109/71.993206.

[134] C. N. Potts and V. A. Strusevich, "Fifty years of scheduling: A survey of milestones," *Journal of the Operational Research Society*, vol. 60, S41–S68, 2009. DOI: 10.1057/jors.2009.2.

[135] A. K. Maurya and A. K. Tripathi, "On benchmarking task scheduling algorithms for heterogeneous computing systems," *The Journal of Supercomputing*, vol. 74, no. 7, pp. 3039–3070, Jul. 2018. DOI: 10.1007/s11227-018-2355-0.

[136] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, Dec. 1, 1959. DOI: 10.1007/BF01386390.

[137] F. Mayer, M. Knaust, and M. Philippsen, "OpenMP on FPGAs - a survey," in *OpenMP: Conquering the Full Hardware Spectrum*, X. Fan, B. R. de Supinski, O. Sinnen, and N. Giacaman, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2019, pp. 94–108. DOI: 10.1007/978-3-030-28596-8_7.

[138] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009. DOI: 10.2172/1407078.

[139] B. da Silva, A. Braeken, E. H. D'Hollander, and A. Touhafi, "Performance modeling for FPGAs: Extending the roofline model with high-level synthesis tools," *International Journal of Reconfigurable Computing*, vol. 2013, pp. 1–10, 2013. DOI: 10.1155/2013/428078.

[140] Z. Wang, B. He, W. Zhang, and S. Jiang, "A performance analysis framework for optimizing OpenCL applications on FPGAs," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Barcelona, Spain: IEEE, Mar. 2016, pp. 114–125. DOI: 10.1109/HPCA.2016.7446058.

[141] S. Wang, Y. Liang, and W. Zhang, "FlexCL: An analytical performance model for OpenCL workloads on flexible FPGAs," in *Proceedings of the 54th Annual Design Automation Conference 2017*, Austin TX USA: ACM, Jun. 18, 2017, pp. 1–6. DOI: 10.1145/3061639.3062251.

[142] H. R. Zohouri, N. Maruyama, A. Smith, M. Matsuda, and S. Matsuoka, "Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, (Salt Lake City, Utah), ser. SC '16, Piscataway, NJ, USA: IEEE Press, 2016, 35:1–35:12. DOI: 10.1109/sc.2016.34.

[143] L.-N. Pouchet. "Polybench: The polyhedral benchmark suite." (2012), [Online]. Available: http://www.cs.ucla.edu/pouchet/software/polybench.

[144] I. Ahmad and M. K. Dhodhi, "Multiprocessor scheduling in a genetic paradigm," *Parallel Computing*, vol. 22, no. 3, pp. 395–406, 1996. DOI: 10.1016/0167-8191(95)00068-2.

[145] M. S. Benten and S. M. Sait, "Genetic scheduling of task graphs," *International Journal of Electronics*, vol. 77, no. 4, pp. 401–415, 1994. DOI: 10.1080/00207219408926072.

[146] Y.-K. Kwok and I. Ahmad, "Efficient scheduling of arbitrary task graphs to multiprocessors using a parallel genetic algorithm," *Journal of Parallel and Distributed Computing*, vol. 47, no. 1, pp. 58–77, 1997. DOI: 10.1006/jpdc.1997.139.

[147] H.-Y. Liu and L. P. Carloni, "On learning-based methods for design-space exploration with high-level synthesis," in *Proceedings of the 50th Annual Design Automation Conference*, Austin Texas: ACM, May 29, 2013, pp. 1–7. DOI: 10.1145/2463209.2488795.

[148] R. Kastner, J. Matai, and S. Neuendorffer, "Parallel programming for FPGAs," *ArXiv e-prints*, May 2018.

[149] "Note sur une méthode de résolution des équations normales provenant de l'application de la méthode des Moindres Carrés a un système d'équations linéaires en nombre inférieur a celui des inconnues.," *Bulletin géodésique*, vol. 2, no. 1, pp. 67–77, Apr. 1, 1924. DOI: 10.1007/BF03031308.

[150] R. E. Lord, J. S. Kowalik, and S. P. Kumar, "Solving linear algebraic equations on an MIMD computer," *Journal of the ACM (JACM)*, vol. 30, no. 1, pp. 103–117, 1983. DOI: 10.1145/322358.322366.

[151] L.-C. Canon, M. E. Sayah, and P.-C. Héam, "A comparison of random task graph generation methods for scheduling problems," in *Euro-Par 2019: Parallel Processing: 25th International Conference on Parallel and Distributed Computing, Göttingen, Germany, August 26–30, 2019, Proceedings 25*, Springer, 2019, pp. 61–73. DOI: 10.1007/978-3-030-29400-7_5.

[152] P. Erdős and A. Rényi, "On random graphs. I.," *Publicationes Mathematicae Debrecen*, vol. 6, no. 3-4, pp. 290–297, Jul. 1, 2022. DOI: 10.5486/PMD.1959.6.3-4.12.

[153] J. G. Siek, L.-Q. Lee, and A. Lumsdaine, *The boost graph library: user guide and reference manual*. Pearson Education, 2001.

[154] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2009.

[155] J. Kuipers and G. Moffa, "Uniform random generation of large acyclic digraphs," *Statistics and Computing*, vol. 25, no. 2, pp. 227–242, Mar. 2015. DOI: 10.1007/s11222-013-9428-y.

[156] T. Bray, "The JavaScript object notation (JSON) data interchange format," Internet Engineering Task Force, Request for Comments RFC 8259, Dec. 2017, 16 pp. DOI: 10.17487/RFC8259.

[157] U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, and M. S. Marshall, "GraphML progress report structural layer proposal," in *Graph Drawing: 9th International Symposium, GD 2001 Vienna, Austria, September 23–26, 2001 Revised Papers 9*, Springer, 2002, pp. 501–512. DOI: 10.1007/3-540-45848-4_59.

[158] R. Polli, "REST API media types," Internet Engineering Task Force / Internet Engineering Task Force, Internet-draft draft-ietf-httpapi-rest-api-mediatypes-02, Sep. 7, 2022, 14 pp.