POLITECNICO
MILANO 1863

POLITECNICO DI MILANO
DIPARTIMENTO DI ELETTRONICA, INFORMAZIONE E BIOINGEGNERIA
DOCTORAL PROGRAMME IN INFORMATION TECHNOLOGY

CAOS: CAD AS AN ADAPTIVE OPEN-PLATFORM SERVICE FOR HIGH PERFORMANCE RECONFIGURABLE SYSTEMS

Doctoral Dissertation of:
Marco Rabozzi

Supervisor:
Prof. Marco D. Santambrogio

Tutor:
Prof. Pier Luca Lanzi

The Chair of the Doctoral Program:
Prof. Andrea Bonarini

2018 – XXXI Cycle

Abstract

In current years we are assisting at a new era of computer architectures, in which the need for energy-efficiency is pushing towards hardware specialization and the adoption of heterogeneous systems. This trend is reflected in the High Performance Computing (HPC) domain that, in order to sustain the ever-increasing demand for performance and energy efficiency, started to embrace heterogeneity and to consider hardware accelerators such as Graphics Processing Units (GPUs), Field Programmable Gate Arrays (FPGAs) and dedicated Application-Specific Integrated Circuits (ASICs). Among the available solutions, FPGAs, thanks to their advancements, currently represent a very promising candidate, offering a compelling trade-off between efficiency and flexibility that is arguably the most beneficial. FPGA devices have also attained renewed interests in recent years as hardware accelerators within the cloud domain. Tangible examples of this are the Amazon EC2 F1 instances, which are compute instances equipped with Xilinx UltraScale+ FPGA boards. The possibility to access FPGAs as on demand resources is a key step towards the democratization of the technology and to expose it to a wide range of application domains.

Despite the potential benefits given by embracing reconfigurable hardware both in the HPC and cloud contexts, we notice that one of the main limiting factor to the widespread adoption of FPGAs is complexity in programmability, as well as the effort required to port a pure software solution to an efficient hardware-software implementation targeting reconfigurable heterogeneous systems.

The main objective of this dissertation is the development of CAD as an Adaptive Open-platform Service (CAOS), a platform able to guide the application developer in the implementation of efficient hardware-software solutions for high performance reconfigurable systems. The platform aims at assisting the designer starting from the high level analysis of the code, towards the definition of the functionalities to be accelerated on the reconfigurable nodes. Furthermore, the platform guides the user in the selection of a suitable architectural template for the FPGA devices and the final implementation of the system together with its runtime support. Finally, CAOS has been designed to facilitate the integration of external contributions and to foster research on the development of Computer Aided Design (CAD) tools for accelerating software applications on FPGA-based systems.

Sommario

NEGLI ultimi anni stiamo assistendo ad una nuova era nello sviluppo delle architetture dei calcolatori, nella quale la necessità di soluzioni energeticamente efficienti, sta stimolando la ricerca verso hardware specializzato e l'utilizzo di sistemi eterogenei. Questo trend si riflette anche nel dominio HPC che, dovendo sostenere una richiesta di prestazioni ed efficienza energetica in crescente aumento, ha iniziato a considerare l'uso di sistemi eterogenei e di acceleratori hardware come GPU, FPGA ed ASIC. Tra le diverse soluzioni disponibili, le FPGA, grazie ai recenti avanzamenti tecnologici, rappresentano al momento un candidato molto promettente, offrendo un bilanciamento competitivo tra efficienza e flessibilità. Negli ultimi anni, i dispositivi FPGA hanno ottenuto una rinnovata attenzione nell'ambito cloud. Esempi tangibili sono le istanze Amazon EC2 F1, ovvero istanze di computazione connesse con schede Xilinx UltraScale+. La possibilità di accedere alle FPGA come risorse on demand rappresenta un passo fondamentale verso la democratizzazione di questa tecnologia e verso il suo utilizzo in un ampio insieme di domini applicativi.

Nonostante i potenziali benefici dati dall'utilizzo di hardware riconfigurabile sia nell'ambito HPC che in quello cloud, notiamo che uno dei maggiori fattori limitanti alla diffusione delle FPGA è la loro complessità nell'essere programmate e la difficoltà nello sfruttare la tecnologia per implementare sistemi hardware-software efficienti partendo da una pura implementazione software.

L'obiettivo principale di questa tesi è lo sviluppo di CAOS, una piattaforma in grado di guidare lo sviluppatore verso l'implementazione di

soluzioni hardware-software efficienti per sistemi riconfigurabili ad alte prestazioni. La piattaforma mira ad assistere lo sviluppatore partendo dalla analisi di alto livello del codice e guidandolo verso la definizione delle funzionalità da accelerare sui nodi di computazione riconfigurabili. Infine, supporta lo sviluppatore nella selezione di un template architetturale per realizzare l'acceleratore e nell'implementazione del sistema insieme al supporto runtime. Inoltre, il design di CAOS è stato realizzato con l'obiettivo di facilitare l'integrazione di contributi esterni e di promuovere la ricerca nello sviluppo di nuovi strumenti per l'accelerazione di applicazioni su sistemi basati su FPGA.

Contents

List of Figures

List of Figures

List of Tables

List of Acronyms

ACO Ant Colony Optimization

API Application Programming Interface

ASIC Application-Specific Integrated Circuit

BRAM Block RAM

CAD Computer Aided Design

CAOS CAD as an Adaptive Open-platform Service

CLB Control Logic Block

CPM Critical Path Method

DAG Directed Acyclic Graph

DDR Double Data Rate

DFE Dataflow Engine

DSE Domain Space Exploration

DSL Domain Specific Language

DSP Digital Signal Processing

EC2 Elastic Compute Cloud

EXTRA Exploiting eXascale Technology with Reconfigurable Architectures

List of Acronyms

FF Flip Flop

FPGA Field Programmable Gate Array

GALS Globally Asynchronous Locally Synchronous

GA Genetic Algorithm

GPP General Purpose Processor

GPU Graphics Processing Unit

GP General Purpose

HDL Hardware Description Language

HLS High-Level Synthesis

HPC High Performance Computing

HPWL half-perimeter wire length

HP High Performance

HW Hardware

ICAP Internal Configuration Access Port

II Initiation Interval

ILP Integer Linear Programming

IRL Irreducible Realization List

IR Intermediate Representation

ISL Iterative Stencil Loop

LLVM Low-Level Virtual Machine

LUT Look-Up Table

MIG Memory Interface Generator

MILP Mixed-Integer Linear Programming

PA Proposed Approach

PDR Partial Dynamic Reconfiguration

PE Processing Element

PL Programmable Logic

PR Partial Reconfiguration

PS Programmable System

RCSP Resource Constrained Scheduling Problem

RR Reconfigurable Region

SA Simulated Annealing

SoC System-on-Chip

SSA Static Single Assignment

SST Streaming Stencil Time-step

SW Software

TSP Traveling Salesman Problem

UI User Interface

URAM Ultra RAM

CHAPTER 1

Introduction

Over the last 40 years, software performance has benefited from the exponential improvement of General Purpose Processors (GPPs) that resulted from a combination of architectural and technological enhancements (Figure 1.1). In 1974, under some assumptions, Robert H. Dennard showed that reducing the dimension of a MOSFET transistor by a factor k leads to a power-delay product improvement of k^3, which translates into the possibility to integrate a factor of k^2 transistors on the same die area and to scale the chip frequency k times without increasing the overall power consumption [28]. Said in other words, given a fixed die area, it is possible to double the number of transistors and increase the clock frequency by about 40% with the same power consumption. This result is referred as Dennard scaling and paired with the well known Moore's law [64], which states that the number of transistors doubles roughly every 18 months, has been one of the main source of performance improvement from the 1970s to the early 2000s. However, despite Dennard scaling served us well, around 2005 it was no more possible to keep the k^3 power-delay improvement due to issues in reducing the threshold voltage [12]. Hence, a reduction of transistor size by a factor of k settled to a more conservative k^2 improvement in the power-delay product. Furthermore, with the failure of Dennard scaling and

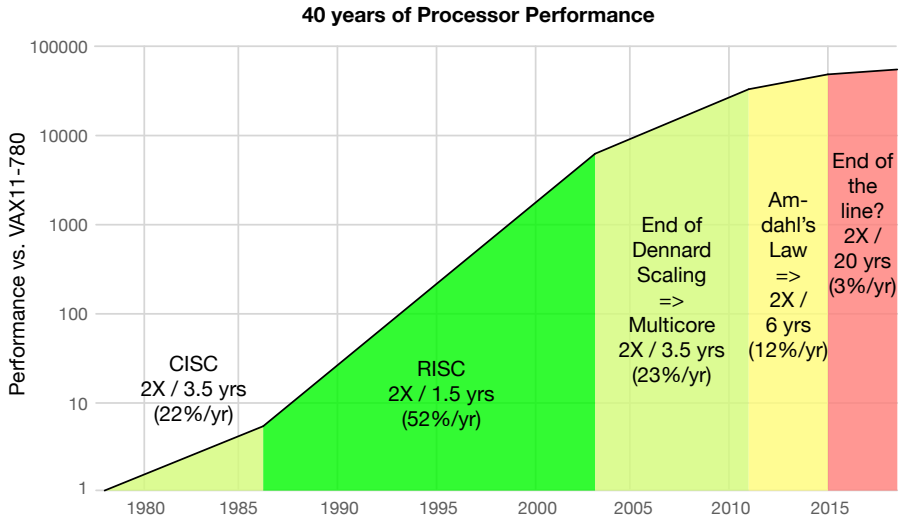


Figure 1.1: *40 Years of processor performance improvement based on the SPECintCPU benchmark. Image and data adapted from [46].*

the continuous shrinkage of transistors, it was no more possible to scale frequency due to the unmanageable power density to dissipate [101]. The phenomenon is referred as the *power wall* and led the way to multiprocessors design as an alternative approach to achieve higher performance. Despite the technological and architectural achievements of GPP, the performance measured on standard benchmarks in the last 3 years only improved at a rate of about 3% per year [46]. After the failing of Dennard scaling, the current diminishing performance improvements of GPP resides in the difficulty to efficiently extract more fine-grained and coarse-grained parallelism from software.

Considering the shortcomings of GPP, in current years we are assisting at a new era of computer architectures in which the need for energy-efficiency is pushing towards hardware specialization and the adoption of heterogeneous systems. This trend is also reflected in the High Performance Computing (HPC) domain that, in order to sustain the ever-increasing demand for performance and energy efficiency [22], started to embrace heterogeneity and to consider hardware accelerators such as Graphics Processing Units (GPUs) [96], Field Programmable Gate Arrays (FPGAs) [76] and dedicated Application-Specific Integrated Circuits (ASICs) [97] along with standard CPU.

Albeit ASICs show the best performance and energy efficiency figure, they are not cost-effective solutions due to the diverse and ever-evolving

HPC workloads and the high complexity of their development and deployment, especially for HPC. Programmability is thus an essential feature that accelerators deployed in datacenters have to enjoy, the two most common representative being GPUs and FPGAs. Among the available solutions, FPGAs, thanks to their advancements, currently represent the most promising candidate, offering a compelling trade-off between efficiency and flexibility that is arguably the most beneficial. Indeed, FPGA are becoming a valid HPC alternative to GPU, as they provide very high computational performance with superior energy efficiency by employing customized datapaths and thanks to hardware specialization.

FPGA devices have also attained renewed interests in recent years as hardware accelerators within the cloud domain. Tangible examples of this are the Amazon EC2 F1 instances, which are compute instances equipped with Xilinx UltraScale+ FPGA boards, as well as the FPGA-accelerated cloud servers from Huawei, Baidu, Tancent and the Alibaba Cloud and Nimbix FPGA-based instances. The possibility to access FPGAs as on demand resources is a key step towards the democratization of the technology and to expose them to a wide range of potential domains, ranging from computational chemistry [14], genomics [30], finance [72], image processing [52] and machine learning [116] to cite a few.

Another distinguishing feature of FPGAs which has become common in recent years, is their ability to support Partial Dynamic Reconfiguration (PDR), which allows to reconfigure at runtime only a small portion of the device while the rest of the system keep running. Such capability opens up a new degree of freedom in developing FPGA-based system [95] and enables the designer to overcome the resource constraints imposed by the device by time-multiplexing the hardware resources when needed.

1.1 The FPGA programmability challenge

Despite the potential benefits given by embracing reconfigurable hardware in both the HPC and cloud contexts, we notice that one of the main limiting factor to the widespread adoption of FPGAs is complexity in programmability as well as the effort required to port a pure software solution to an efficient hardware-software implementation targeting reconfigurable heterogeneous systems [5]. During the past decade we have seen significant progress in High-Level Synthesis (HLS) tools which partially mitigate this issue by allowing to translate functions written in a high-level languages such as C/C++ to an hardware description language suitable for hardware synthesis. Nevertheless, current tools still require experienced users in or-

der to achieve efficient implementations. In most cases indeed, the proposed workflows require the user to learn the usage of specific optimization directives, code rewriting techniques and, in other cases, to master domain specific languages. In addition to this, most of the available solutions focus on the acceleration of specific kernel functions and leave to the user the responsibility to explore hardware/software partitioning as well as to identify the most time-consuming functions which might benefit the most from hardware acceleration.

Furthermore, an additional degree of complexity is introduced when PDR is employed for the target FPGA-based design. Indeed, PDR requires the designer to partition the design on different Reconfigurable Regions (RRs), define the floorplan of the RRs on the FPGA, and, finally, decide how to schedule the execution of the module on the RRs in order to minimize the reconfiguration overhead. All such decisions impact on the quality and performance of the final design, despite current vendors tools offer none or very small guidance on how to perform such tasks.

1.2 Contributions

The main objective of this research is the development of a platform able to guide the application developer in the implementation of efficient hardware-software solutions for high performance reconfigurable systems. The platform aims at assisting the designer starting from the high level analysis of the code, towards the definition of the functionalities to be accelerated on the reconfigurable nodes, the selection of a suitable architectural template for the FPGA devices and the final implementation of the system together with its runtime support. Most of the work done in this thesis has been developed in the context of the Exploiting eXascale Technology with Reconfigurable Architectures (EXTRA) project and shares with it the same vision [83]. The following sections provides more details on the specific contributions in the context of the proposed platform.

1.2.1 An open-research platform to democratize high performance reconfigurable systems

The platform, dubbed as CAD as an Adaptive Open-platform Service (CAOS), targets both application developers and researches while its design has been conceived focusing on three key principles: usability, interactivity and modularity. From a usability perspective, the framework supports application designers with low expertise on reconfigurable heterogeneous systems in quickly optimizing their code, analyzing the potential performance gain

and deploying the resulting application on the target reconfigurable architecture. Nevertheless, the platform does not aim to perform the analysis and optimizations fully automatically, but instead interactively guides the users towards the design flow, providing suggestion and error reports at each stage of the process. Finally, CAOS is composed of a set of independent modules accessed by the CAOS flow manager that orchestrates the execution of the modules according to the current stage of the design flow. Each module is required to implement a set of well-defined Application Programming Interface (API) so that external researchers can easily integrate their implementations and compare them against the ones already offered by CAOS.

1.2.2 New methodologies for accelerating high-level applications through FPGA devices

A general method for translating high-level functions into FPGA-accelerated kernels has been proposed within CAOS. The core idea revolves on matching a software function with an architectural template, which is a characterization of the accelerator both in terms of its computational model and the communication with the off-chip memory. An architectural template constrains the architecture to be generated on the reconfigurable hardware and poses restrictions on the application code that can be accelerated, so that the number and types of optimizations available can be tailored for a specific type of hardware implementation. Within this context, we have integrated three architectural templates within CAOS:

- **Master/Slave architectural template**: it targets a relatively large set of C/C++ codes and it is well suited for algorithms whose working set that can be efficiently tiled so that the resulting accelerator operates on a subset of the application data that is block-transferred to and from the FPGA local memory. The Master/Slave architectural template explores a set of alternative hardware implementations by applying optimization such as loop pipelining, loop unrolling, memory partitioning and on-chip caching. The optimizations are tested through Xilinx Vivado HLS, while the template supports Xilinx SDAccel and allows to seamlessly implement the final system on Amazon F1 instances in the cloud.

- **Dataflow architectural template**: it specifically targets dataflow-like computations, which have proven to be very effective when implemented on FPGA. Starting from a high level language supported by

Low-Level Virtual Machine (LLVM), it generates a dataflow intermediate representation of the target function and translates it into a chosen target language suitable for hardware synthesis. The bitstream generation is handled by a backend synthesis tool of choice which supports the dataflow computational paradigm. By means of this template, software developers with little to none experience in FPGA programming were able to achieve in a about a day of work speedup comparable to bespoke implementations.

- **Streaming architectural template**: it specifically targets stencil computation written in C and leverages the Streaming Stencil Time-step (SST) technology [16] for implementing the final accelerator on the target reconfigurable system. Within this context, we proposed a design exploration algorithm that jointly maximizes the number of processing elements that can be instantiated on the target FPGA and identifies a floorplan of the design that minimizes the inter-component wire-length in order to allow implementing the system at a higher clock frequency.

1.2.3 CAD algorithms for partially reconfigurable FPGA designs

The CAOS backend has been conceived in order to support floorplanning, scheduling and mapping for PDR. In this dissertations we provide new approaches to automatically perform such steps in order to facilitate the integration of architectural templates that require the usage of PDR. More specifically we present:

- A novel floorplanning automation framework, integrated in the Xilinx toolchain, which is based on an explicit enumeration of the possible placements of each region. Moreover, we propose a genetic algorithm, enhanced with a local search strategy, to automate the floorplanning activity on the defined direct problem representation. Experimental results demonstrated the effectiveness of the proposed direct problem representation and the superiority of the defined genetic algorithm engine with respect to the other approaches in terms of exploration time and identified solution.
- A new scheduling technique for partially-reconfigurable FPGA-based systems that allows to achieve high quality results in terms of overall application execution time. The proposed algorithm exploits the notion of resource efficient task implementations in order to reduce the overhead incurred by partial dynamic reconfiguration and increase the

number of concurrent tasks that can be hosted on the reconfigurable logic as hardware accelerators. We evaluated a fast deterministic version of the scheduler that is able to find good quality solutions in a small amount of time and a randomized version of the approach that can be executed multiple times to improve the final result.

1.3 Sources

This dissertation refers to and possibly extends the following publications:

- **Floorplanning Automation for Partial-Reconfigurable FPGAs via Feasible Placements Generation**, M. Rabozzi, G. C. Durelli, A. Miele, J. Lillis and M. D. Santambrogio. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2016.
- **Resource-efficient scheduling for partially-reconfigurable FPGA-based systems**, A. Purgato, D. Tantillo, M. Rabozzi, D. Sciuto, M. D. Santambrogio. Appeared in the proceedings of Reconfigurable Architecture Workshop (RAW), 2016.
- **Heterogeneous Exascale Supercomputing: The Role of CAD in the exaFPGA Project**, M. Rabozzi, G. Natale, E. Del Sozzo, A. Scolari, L. Stornaiuolo and M. D. Santambrogio. Appeared in Proceedings of the Conference on Design, Automation & Test in Europe, 2017.
- **A CAD Open Platform for high performance reconfigurable systems in the EXTRA project**, M. Rabozzi, R. Brondolin, G. Natale, E. Del Sozzo, M. Huebner, A. Brokalakis, C. Ciobanu, D. Stroobandt and M. D. Santambrogio. Appeared in IEEE Computer Society Annual Symposium on VLSI (ISVLSI), 2017.
- **Optimizing streaming stencil time-step designs via FPGA floorplanning**, M. Rabozzi, G. Natale, B. Festa, A. Miele, M. D. Santambrogio. Appeared in 27th International Conference on Field Programmable Logic and Applications (FPL), 2017.
- **The role of CAD frameworks in heterogeneous FPGA-based cloud systems**, L. Di Tucci, M. Rabozzi, L. Stornaiuolo and M. D. Santambrogio. Appeared in IEEE International Conference on Computer Design (ICCD), 2017.
- **OXiGen: A tool for automatic acceleration of C functions into dataflow FPGA-based kernels**, F. Peverelli, M. Rabozzi, E. Del Sozzo,

M. D. Santambrogio. Appeared in the proceedings of Reconfigurable Architecture Workshop (RAW), 2018.

1.4 Outline

The context, motivations and the design of the proposed CAOS platform are described in Chapter 2. After having introduced the core of the CAOS platform, Chapters 3 to 5 delve into the Master/Slave, Dataflow and Streaming architectural templates that have been integrated in CAOS. Each architectural template provides support for different types of input codes and the generated hardware accelerator is dependent upon the specific architectural template. In the second part of the dissertation, we focus on algorithms and methods to support FPGA designs that require PDR. More specifically, Chapter 6 discuss the proposed novel floorplanning methodology for partially reconfigurable designs, while Chapter 7 presents a new scheduling technique for Directed Acyclic Graph (DAG) of compute tasks on a set of partially reconfigurable regions. Both approaches can be leveraged within the CAOS backend to support future architectural templates that make use of PDR. Finally, Chapter 8 draws the conclusions and discusses future research directions.

1.5 How to read this thesis

The order of the chapters in this dissertation has been conceived to facilitate the reader in understanding the overall CAOS platform starting from its main design and principles discussed in Chapter 2, moving then to the supported architectural templates described in Chapters 3 to 5, and, finally, discussing new approaches and algorithms in Chapters 6 and 7 to enable the support for partially-reconfigurable designs within CAOS. Nevertheless, most of the chapters can also be read independently. If the reader is interested in a specific architectural template and not in the entire CAOS platform, he/she can directly refer to one of Chapters 3 to 5 and skip the corresponding section discussing the template integration within CAOS. However, if the reader is also interested in learning how the specific architectural template can be used through CAOS, we encourage the reader to read Chapter 2 first. Similarly, the reader interested in the floorplanning and scheduling techniques presented in Chapter 6 and Chapter 7, can refer directly to the corresponding chapter or start with Chapter 2 first if he/she is also interested in their potential applications within the CAOS platform.

CHAPTER 2

The CAOS platform

This chapter sets the context in which the proposed CAOS platform has been devised and presents its design flow and peculiarities. The platform aims at providing a fully integrated solution for automating and assisting all the steps for accelerating applications on High Performance Computing (HPC) and cloud systems featuring FPGA devices. CAOS also provides practical interfaces to enable extensions and enhancements of its functionalities and promote contribution from the community, ultimately pushing the adoption of reconfigurable hardware in the HPC and cloud domains.

2.1 Background and motivations

This section reviews the background material at the base of CAOS, highlighting limitations and open challenges that motivate this work. Despite the plethora of High-Level Synthesis (HLS) tools that have emerged both in the academia in industry in the last years [68], the platforms, frameworks and design flows that targets the full software stack for the implementation of high-level applications onto a target FPGA-based system are only a handful [36, 45, 50, 60, 78, 94, 103, 109].

Since HPC and cloud systems comprise multiple nodes, the description of nodes' interconnection and resources can be formalized through the concept of *template*, which describes the system organization at a certain granularity. We can distinguish three main templates: a *system template* describes how the nodes are interconnected, a *node template* describes the computational resources each node contains and an *architectural template* describes the computational kernels deployed on the Field Programmable Gate Array (FPGA). Throughout this section we will use these three categories to review the relevant work around CAOS.

2.1.1 Existing workflows for FPGA development

Due to the diversity of domains and vendors, different development practices and tool suites have established to aid developers, yet at the cost of a strong lock-in. As an example, Xilinx provides different tool sets and flows like the Vivado Suite, SDAccel [109] and SDSoC [50], while Altera provides "Quartus Prime", "FPGA SDK for OpenCL" and others for the same domains. For instance, the Vivado Suite is used for custom FPGA development and leaves the user in control of the whole design process, allowing free choice of the templates (system, node, architectural) but exposing most of the complexity of this process. To help System-on-Chip (SoC) designers, Xilinx's MPSoC simplifies profiling, HLS and hardware/software interfacing, greatly reducing engineering time and effort; yet, it is applicable only to SoC development and assumes fixed templates at all system levels. Finally, SDAccel allows developing FPGA-based applications at a higher level of abstraction than Vivado, targeting PCIe-attached accelerators. SDAccel eases acceleration by assisting the user during the HLS process and automatizing the subsequent phases, including the generation of hardware/software interfaces and runtime management, but leaves no control over the templates and the system parameters. Another workflow is Dyplo by Topic [103], which targets embedded applications and focuses on simplifying the development of partially reconfigurable solutions, assum-

ing a single-node system with CPU+FPGA; Dyplo uses a fixed architectural template based on a central ring bus for communication, which simplifies runtime reconfiguration. Based on the dataflow model [29], Maxeler Technologies [60] provides integrated solutions for distributed applications with cluster-shared accelerators or tightly coupled CPU+FPGA nodes. The Maxeler tools are based on a custom java-based language (named MaxJ), which automates the synthesis phase and hides many implementation details, fixing the node and system templates on the basis of the workflow initially chosen.

On the academic side, several works also attempt to establish design methodologies [36, 45, 78, 94], but fall short of the freedom of templates choice and of flexibility. The FASTER project [94] had a key role in defining new ways to support reconfigurability both at granular and modular level while establishing a methodology for exploring HW/SW partitioning end tasks scheduling. However, it does not explicitly consider the concept of architectural template in order to distinguish among the different accelerator implementations to efficiently target specific codes, as well as not specifically providing ways to integrate external modules and contributions within the platform. Indeed, we think that the latter consideration is fundamental in order to stimulate continuous research on Computer Aided Design (CAD) tools for FPGA.

In [78], the authors focus on multi-fpga systems as well as on partial dynamic reconfiguration and propose a methodology to partition a design onto the target system in order to fit within the resource constraints of the reconfigurable logic. Despite the flow targets the final implementation of the system, it does so starting from a Hardware Description Language (HDL) definition of the application and does not explicitly support high-level languages. On the other side, LegUp is a popular open-source HLS tool that also provides means to profile the application in order to help the user in identifying portions of the code would benefit from hardware acceleration [36]. Furthermore, it also transparently generates the required interfaces and host code to implement the final application. Nevertheless, the approach currently lacks an automatic design space exploration to optimize the implementation of the kernels while it is still up to the user to define the functions to accelerate. On the opposite side, more recent efforts like Darkroom [45] propose a workflow for image and video processing kernels that is based on a Domain Specific Language (DSL), in order to automatize the port of such kernels to FPGA. However, a DSL is too restrictive for the plurality of HPC and cloud applications, which have much more complex algorithmic patterns.

2.1.2 Unsolved issues and motivations behind CAOS

As the research is still struggling with the complexity of accelerating applications on FPGA-based systems and the quest for automated solutions, practitioners still resort to full-custom designs in order to achieve high performance, and the learning curve remains very steep. The exploration of solutions, practices and architectural paradigms for FPGAs is likely to continue, and possibly broaden, in the coming years [4, 100, 102], rising two main needs. The first need is to devise a general enough workflow for FPGA acceleration taking into account the complexity of HPC and cloud systems, whose organization is typically hierarchical. This workflow should allow the designer to *choose the organization of the system* at the various levels, corresponding to the three templates introduced at the beginning of this section, while guiding the user towards the different design choices. The second need is to have a common platform for implementing the solutions the research proposes over time, ensuring *"pluggability" of components*. For example, users willing to showcase profiling techniques or resource estimation algorithms should easily plug into the system and compare their work with others.

2.2 A bird's eye view on CAOS

The entire CAOS platform has been designed around the following principles, which are independent of the technology used to build it:

- **[Usability]** Usually the skills needed to take advantage of the hardware architectures to accelerate applications are very high. The modern HLS and hardware design tools aim at facilitating some steps of this process, but they still require a specialized knowledge of the target architecture. Also, the user need to manually analyze the application to figure out what are the techniques to use to obtain the desired result. One of the objectives of the CAOS platform is to allow users with low expertise on reconfigurable heterogeneous systems to quickly optimize their applications, analyze the potential performance gain and deploy the final design on the target architecture. This is possible thanks to the automation of certain procedures that are able to move along the different branches of the flow and through the various optimizations and implementations in hardware of the application. Moreover, each operation performed by the platform will be illustrated and explained to the user so that he can improve his experience and the results with each use.

- **[Interactivity]** As mentioned, the Usability feature of the CAOS platform is intended to provide guidance to novice users through the optimization flow with automatic procedures. Since this might be a constraint for more experienced users, who may want to apply their own techniques and knowledge to accelerate their applications and exploit the target architecture, all processes have been made semi-automatic. This means that, in addition to offering suggestions and reporting possible errors, each phase of the flow allows the user to specify or change the solutions provided by the platform and the way in which they are obtained. In this way users can customize the stage where they have more experience and be guided by the platform in the others. All the options proposed by the semi-automatic system are customizable and the user can skip the actions of each step replacing them with their own. It is also possible to go back and forward along the phases and modify the previous options knowing the results of the following steps.
- **[Modularity]** One of the strongest points of the CAOS platform is to implement a flow that has been divided into modules, each of which implements its own function. This allows to better manage the Usability and the Interactivity by defining and examining the inputs and the outputs of each step, and ensures separation of concerns among the components of the platform, which can be used separately from each other. Moreover, each module provides a set of well defined Application Programming Interfaces (APIs) to allow seamless integration of different versions of the analysis tools, design exploration algorithms and performance models leveraged within the flow. In this way the entire platform is scalable, extensible and upgradeable by experienced users who can improve the modules with their skills and knowledge.
- **[Well Defined Interfaces]** Modularity allows to generalize the flow of process that accelerates software applications by exploiting the FPGA architecture. To do that, it is necessary to define interfaces that ensure isolation and integrity between the modules. The goal is to generalize the inputs and the outputs of each module to define interfaces that guarantee its replaceability. To replace a module means that an expert user or company can build its own implementations of modules with the only constraint that they have to respect the defined interfaces.

The platform expects the application designer to provide the application code written in a high level language such as C / C++, one or mul-

tiple datasets to be used for code profiling and a description of the target reconfigurable system. In order to narrow down and simplify the set of possible optimizations and analysis that can be performed on a specific algorithm, CAOS allows the user to accelerate its application using one of the available architectural templates. An *architectural template* is a characterization of the accelerator both in terms of its computational model and the communication with the off-chip memory. As a consequence, an architectural template constrains the architecture to be implemented on the reconfigurable hardware and poses restrictions on the application code that can be accelerated, so that the number and types of optimizations available can be tailored for a specific type of implementation. Furthermore, CAOS is meant to be orthogonal and build on top of tools that perform high level synthesis, place and route and bitstream generation. Code transformations and optimizations are performed at the source code level while each architectural template has its own requirements in terms of high level synthesis and hardware synthesis tools to use.

CAOS currently support three different architectural templates: the Master-Slave, the Dataflow and the Streaming architectural templates. The Streaming architectural template targets stencil codes written in C and leverages the Streaming Stencil Time-step (SST) architecture proposed by [17] for its implementation. Within this context, CAOS offers a design exploration algorithm [87] that jointly maximizes the number of SST processors that can be instantiated on the target FPGA and identifies a floorplan of the design that minimizes the inter-component wire-length in order to allow implementing the system at a higher frequency. On the other hand, the Master-Slave architectural template [31] poses less restrictions on the final accelerator and source code, it requires that the C / C++ algorithm can be efficiently tiled so that the resulting accelerator operates on a subset of the application data that is block-transferred to and from the FPGA local memory. Finally, the Dataflow architectural template [79] targets dataflow applications and exploits the MaxCompiler within the backend to efficiently implement the application as a Maxeler DFE (Dataflow Engine).

2.3 CAOS design flow

As shown in Figure 2.1, the overall CAOS design flow is subdivided into three main flows: the frontend flow, the function optimization flow and the backend flow. The main goal of the frontend is to analyze the application provided by the user, match the application against one or more architectural templates available within the platform, profile the user application

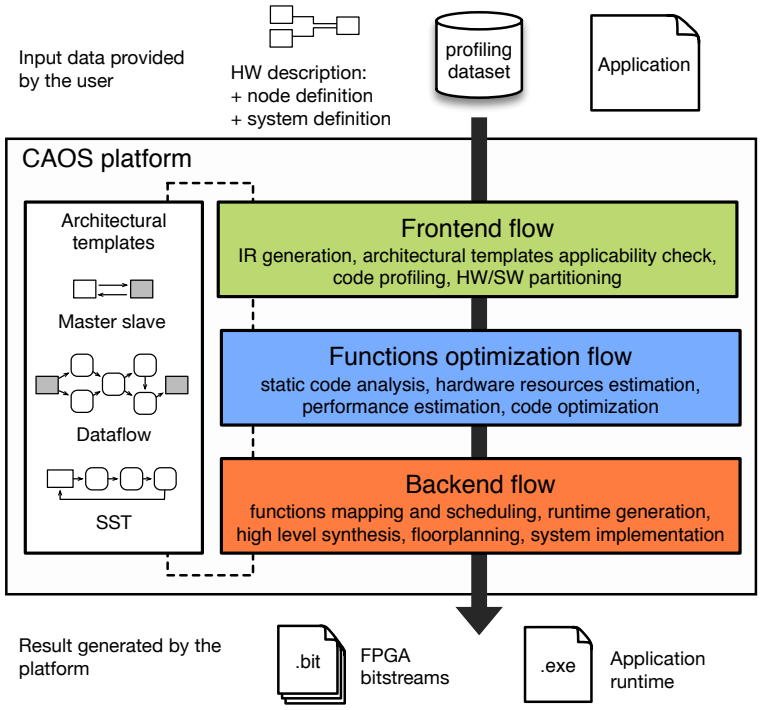


Figure 2.1: *High level overview of the CAOS platform. The overall design flow can be divided in three main parts: the* frontend*, the* functions optimization *and the* backend *flow. The application code, datasets to profile the application and an HW description constitute the input data provided by the designer. The final outputs generated by the platform are the bitstreams, that the user can use to configure the boards, and the application runtime, needed to run the optimized version of his/her application.*

against the user specified datasets and, finally, guide the user through the hardware/software partitioning of the application to define the functions of the application that should be implemented on the reconfigurable hardware. The function optimization flow performs a static analysis and a hardware resource estimation of the functionalities to be accelerated on the FPGA. Such analyses are dependent upon the considered architectural template and the derived information are used to estimate the performance of the hardware functions and to derive the optimizations to apply (such as loop pipelining, loop tiling and loop unrolling). After one or more iterations of the function optimization flow, the resulting functions are given to the backend flow in which the desired architectural template for implementing the system is selected and the required high level synthesis and hardware synthesis tools are leveraged to generate the final FPGA bitstreams. Within

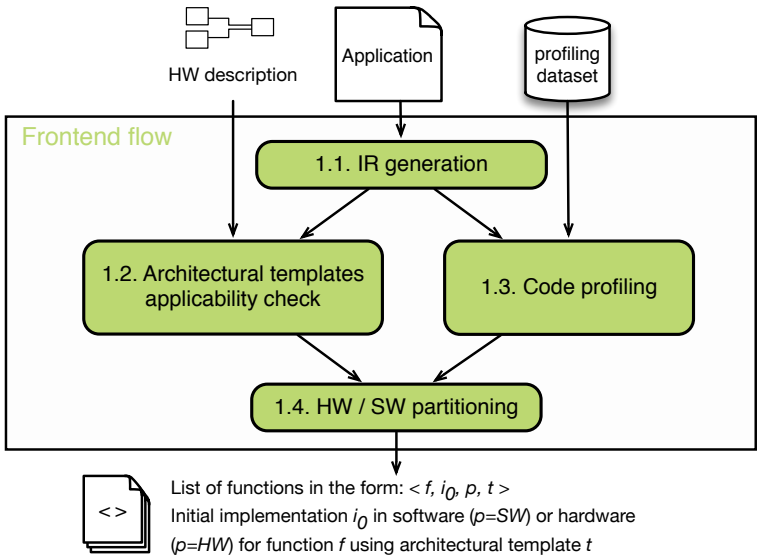


Figure 2.2: *Presentation of the CAOS platform frontend flow that, starting from the user inputs, produces an intermediate description of the application functions. The frontend consists in the following phases: 1.1. IR generation that provides an Intermediate Representation (IR) of the application; 1.2. architectural templates applicability check that analyzes the code to verify which functions fit a specific architectural template; 1.3. code profiling that hands the measured times, in terms of relative and absolute execution time of each function; 1.4. HW / SW partitioning that filters the candidate functions for hardware acceleration, by considering also the profiling data.*

the backend, CAOS takes care of generating the host code for running the FPGA accelerators and optionally guides the place and route tools by floor-planning the system components.

2.3.1 Frontend

The main phases of the frontend flow are depicted in Figure 2.2. This flow represents the first part of the entire design flow and takes as input the code of the user application, the datasets to profile it and a user-selected HW description, which will determine also all the subsequent stages. After completion, the frontend flow outputs a list of decorated functions defined by the tuples $< f, i_0, p, t >$, where f is the function identifier, i_0 is its initial implementation, p specifies the implementation type (either HW or SW) and t identifies the target architectural template. For each function f in the original application, multiple tuples with different target architectural templates t can be generated.

Listing 2.1: *Example of a program description JSON file. The corresponding code is in C language and both gcc and llvm compiler are supported. The "-O3" flag should be used for building the executable with both compilers.*

```
{
  "language": "C",
  "supportedCompilers":
  [
    {
      "compiler": "gcc",
      "arguments": "-O3"
    },
    {
      "compiler": "llvm",
      "arguments": "-O3"
    }
  ]
}
```

In the following subsections we provide a detailed description of the phases and corresponding modules within the CAOS frontend.

IR Generation

When a new CAOS project is created, the design starts from the IR generation phase, where the user is requested to provide an archive containing the source files of their application written in a high level language such as C / C++. Aside from the code archive, the user also needs to provide a properly formatted JSON file that describes essential informations on the application. As show in Listing 2.1, the file specifies the source code language, the list of support compilers as well as their compiler flags that should be used for building the application.

Once the code and the program description file are uploaded, the user can start the CAOS design flow by running the IR generation module. After execution, the module provides the list of functions within the application, the static callgraph describing the caller / callee relations as well as the Low-Level Virtual Machine (LLVM) IR generated from the source code. Such data is leveraged in all the subsequent phases. CAOS indeed works at function granularity in order to perform hardware / software partitioning and to optimize the candidate kernels for hardware acceleration.

Architectural template applicability check

Within the subsequent phase of the CAOS flow, the user is requested to provide a JSON file describing the architecture being targeted by the appli-

cation. An example of architecture description file is shown in Listing 2.2. The architecture description is a two-level specification consisting of a node definition and a system definition. The node definition portion of the specification defines the set of devices comprising a node, which can be either software cores, accelerators or FPGA boards. The specification allows also to define how the different components within a node are interconnected together. Similarly, the system level specification describe how many nodes are available within the system and how they are interconnected together. Given the architecture description and the result from the IR generation phase, the architectural template applicability check verifies which architectural template, among the ones supported by CAOS, is compatible with the given architecture description. Furthermore, depending on the architectural template, the phase also verifies which are the functions that can be accelerated in hardware using the supported architectural templates. If the applicability check module determines that a given function can be accelerated in hardware, it also assumes that all the functions called, either directly or transitively, can be accelerated as well together with the function. The output of the module also provides diagnostic information to allow the user to identify why a given function cannot be accelerated in hardware with a given architectural template. The user can decide to modify the code and repeat the design flow up to this phase in order to enlarge the set of functions that CAOS is able to support for hardware acceleration. It is worth noting that the type of checks performed by this module are very dependent on the architectural template being verified. In Chapters 3 to 5, within the corresponding CAOS integration sections, we provide more details on the specific source code constraints taken into account.

Profiling

The profiling phase allows the designer to profile the application execution on one or more datasets in order to identify the functions in which the highest amount of time is spent. Such information is then leveraged by the subsequent phase of the platform in order to provide a suitable hardware / software partitioning of the application that maximizes performance. In order to perform the profiling, the user is asked to provide, for each dataset, an archive containing the input files needed by the application, as well as the arguments needed to invoke the application. CAOS then launches the profiling module on the given datasets and reports the relative percentage of time spent in each function for each dataset. Currently, the default module available in CAOS leverages on the Linux perf profiler [62], nevertheless, as will be discussed in the next sections, external researchers are allowed

Listing 2.2: *Example of an architecture description JSON file that specifies a system composed of a single Amazon EC2 F1 instance equipped with a Xilinx XCVU9P FPGA.*

```
{
  "nodeDefinition" : {
    "deviceTypes" : {
      "f1-fpga" : {
        "type" : "board",
        "vendor" : "Xilinx",
        "partNumber" : "XCVU9P-FLGB2104-2-I"
      },
      "intel-vcore" : {
        "type" : "cpu",
        "vendor" : "intel",
        "partNumber" : "-"
      }
    },
    "devices" : {
      "f1-fpga-instance" : {
        "type" : "f1-fpga"
      },
      "cpu" : {
        "type" : "intel-vcore",
        "host" : true
      }
    },
    "connectionTypes" : {
      "pcie_gen3" : {
        "standard" : "PCIe",
        "bandwidth" : "15.75 GB/s",
        "duplex" : "full",
        "version" : "3.0"
      }
    },
    "connections" : [
      {
        "source" : "f1-fpga_instance",
        "target" : "cpu",
        "type" : "pcie_gen3"
      }
    ]
  },
  "system" : {
    "nodes" : ["f1-fpga_node"],
    "connectionTypes" : {},
    "connections" : []
  }
}
```

to implement their own profiling module.

HW/SW partitioning

The partitioning phase of CAOS relies on the information provided in the previous frontend phases in order to determine which functions to accelerate on the available reconfigurable hardware. More specifically, the corresponding hardware / software partitioning module searches for the subtree of the static callgraph that takes the highest percentage of the overall execution time and that, at the same time, consists of functions which are all amenable for hardware acceleration. At the current state, the CAOS flow is optimized for accelerating a single function subtree from the static callgraph, nevertheless, the CAOS interfaces allow to support modules that operate on multiple function subtrees. Once the module is executed, the user can also modify the suggested partitioning from the CAOS user interface in case he or she desire a different solution.

2.3.2 Functions optimizations

The function list generated by the frontend is given as input to the CAOS functions optimization flow detailed in Figure 2.3. The function optimization flow operates on the functions that are selected for hardware acceleration from the frontend and its goal is to improve their initial implementations by applying one or more optimizations and code transformations. The flow can be repeated multiple times in order to optimize the candidate hardware functions. After each iteration, the initial implementation i_0 of a function is updated with its optimized implementation i_j. The optimization cycle then repeats from phases 2.1 and 2.2 until no more optimizations are identified or the required goal (e.g., performance or area) is achieved. The updated tuples $< f, i_j, p, t >$ are then passed to the architectural template selection phase (3.1), which is the first phase of the backend flow. It is worth noting that the code is kept to its high-level form, this allows the user to easily keep track of the changes applied to the code. Furthermore, optimizations that cannot be directly applied as code transformations are annotated in the form of pragmas and optimization directives that are taken into account for the analysis performed within the CAOS modules. The details of the phases and corresponding modules within the CAOS function optimization flow are described next.

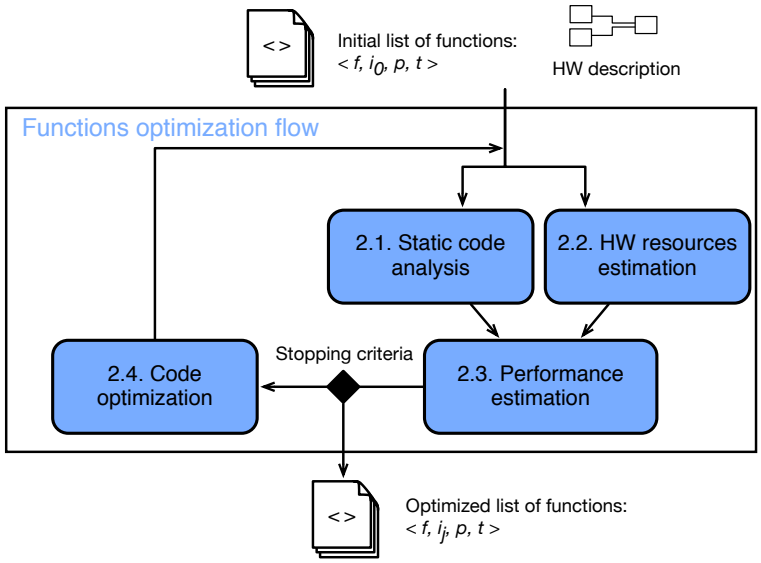


Figure 2.3: *Representation of the CAOS functions optimization flow. The list of functions produced by the frontend are analyzed and optimized by means of the following phases: 2.1 static code analysis that performs a static analysis of the code to derive a set of metrics; 2.2 HW resources estimation that estimates the resource requirements on the reconfigurable hardware for the candidate functions; 2.3 performance estimation that estimates the final performance of the function and determines the set of optimizations that can be applied to the code; 2.4 code optimization that let the user decide which optimization to apply among the identified ones.*

Static code analysis

The static code analysis phase represents the first step of the CAOS function optimization flow which starts after having performed the hardware / software partitioning of the application. The static code analysis module extracts useful information from the source code that can be leveraged to predict performance or to explore possible optimizations in the subsequent phases. Depending on the specific architectural template that is considered for the current function, the static code analysis module returns different types of metrics. Examples of metrics include: the operational intensity of the code, the identification of input and output parameters, bounds on the data size per function invocation or ideal throughput, the estimated number of cycles needed to compute the function after HLS synthesis, the set of local arrays together with their sizes, the iteration counts at different loop nest levels and so on.

Hardware resources estimation

The second step of the function optimization flow performs an estimation of the hardware resources that are needed in order implement the current version of the function on the target reconfigurable hardware. Even in this case, the method used to carry out the estimation can be dependent on the architectural template being considered. Currently, the Dataflow architectural template relies on the LLVM representation of the function and assumes that each operation is performed by a dedicated hardware module, since this is in general highly efficient for a pure dataflow implementation. Leveraging this information and assumption the module is then able to quickly derive a good estimation of the used resources. On the other hand, for the Master/Slave architectural template, the current default implementation of the module leverages on Vivado HLS due to more complex nature of functions supported by such template. Finally, for the Streaming architectural template, the module leverages a combination of Vivado HLS estimates and analytical formulas that depends on the number of processing elements being instantiated within the final design.

Performance estimation

The performance estimation phase leverages the data generated within the static code analysis and the hardware resource estimation phases. The corresponding CAOS module performs two different tasks. First, it estimates the achievable performance of the current implementation of the function and, second, it performs a design space exploration to identify possible optimization opportunities to improve the current implementation. For each optimization, the module also provides an estimation of the hardware resources and performance associated with it. The architectural template abstraction is especially useful for this phase as it narrows down the search space, since each architectural template poses constraints both on the types of supported input code and the corresponding hardware accelerator being implemented. Again, more details on the specific tasks performed at this stage are detailed in Chapters 3 to 5.

Code optimization

The last phase of the functions optimization flow is code optimization. Such phase is tightly coupled with the previous phase since it allows the user to select one of the optimizations suggested by the performance estimation. The module takes as input the optimization selected by the user as well as its parameters and modifies the code accordingly and / or updates the opti-

mizations directives. Once the optimization is applied to the code, the user can either repeat the function optimization flow to search for further optimizations or move to the final implementation phase. After having applied a given optimization, CAOS rerun the IR generation phase on the current version of the code in order to keep it in sync with the applied changes. This is needed so that further iterations of the code optimization flow, or modules within the CAOS backend, can safely rely on the information of the CAOS IR. The reason why we let the user select the desired optimization instead of doing the choice automatically, is to provide more control over the design process. Furthermore, different optimizations might lead to different pareto-optimal solutions in terms of resource consumption and estimated performance that the designer might be willing to exploit. It is also worth noting that while CAOS allows to repeat the optimization process multiple times, a researcher willing to integrate its own architectural template and/or modules can in principle show the user a single optimization opportunity as a result of the best solution found from the design space exploration. This is currently the case for the streaming architectural template discussed in Chapter 5, while the Master/Slave architectural template and the dataflow architectural template discussed in Chapter 3 and Chapter 4 follow the described iterative approach.

2.3.3 Backend

The flow of the CAOS backend is presented in Figure 2.4. The backend receives the functions list generated by the functions optimization flow and the hardware description given by the user. In case multiple architectural templates are considered, the first step of the backend guides the selection of the architectural template for the final system implementation (phase 3.1). Indeed, in order to simplify the backend flow, CAOS currently limits the implementation of the system to a single architectural template. Nevertheless, such limitation might be removed in future versions of the platform. Then, functions mapping and scheduling (phase 3.2), maps and schedules the execution of the HW functions on the reconfigurable logic, possibly consisting of multiple FPGAs. Depending on the selected architectural template, the mapping and scheduling algorithm can also take into account Partial Dynamic Reconfiguration (PDR) [23]. The results of phase 3.2 and the HW description are then used in runtime generation (phase 3.4) to generate the runtime for the system. Simultaneously, the hardware functions are further processed to obtain the final FGPA bitstreams. The HW functions synthesis (phase 3.3) performs HLS and hardware synthesis to

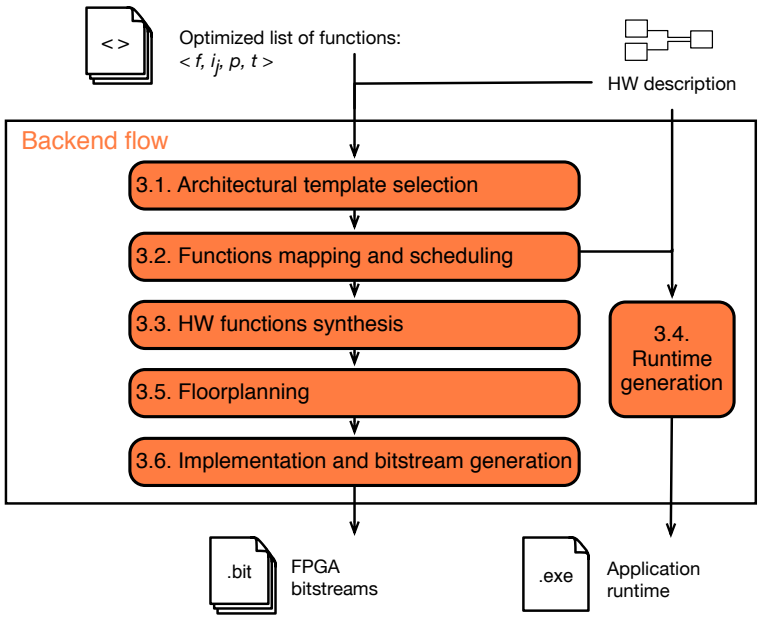


Figure 2.4: *Description of the CAOS backend that produces the final system implementation and the application runtime. The backend consists in the following phases: 3.1. architectural template selection that guides the selection of the architectural template for the final system implementation; 3.2. functions mapping and scheduling that maps and schedules the execution of the HW functions on the reconfigurable logic; 3.3. HW functions synthesis that performs HLS and hardware synthesis to generate the functions' netlists; 3.4. runtime generation that generates the runtime and host code for the system; 3.5. floorplanning that, if needed, constrains the placement of the design modules on specific locations on the reconfigurable logic; 3.6. implementation and bitstream generation that performs the place and route of the HW functions and generates the FPGA bitstreams.*

generate the functions' netlists. Florplanning (phase 3.5) is then optionally executed if the architectural template leverages PDR, or if the architectural template requires to specify floorplanning constraints, as for the Streaming architectural template discussed in Chapter 5. Finally, the implementation and bitstream generation (phase 3.6) performs the place and route of the HW functions and generates the FPGA bitstreams. The output of the CAOS platform are the bitstreams, that the user can use to configure the FPGAs, and the application runtime, needed to run the optimized version of the user application.

Differently from the frontend and the functions optimization flows, the phases of the backend flow are currently deployed in a single macro module. Indeed, while logically separated, most of the phases described in the

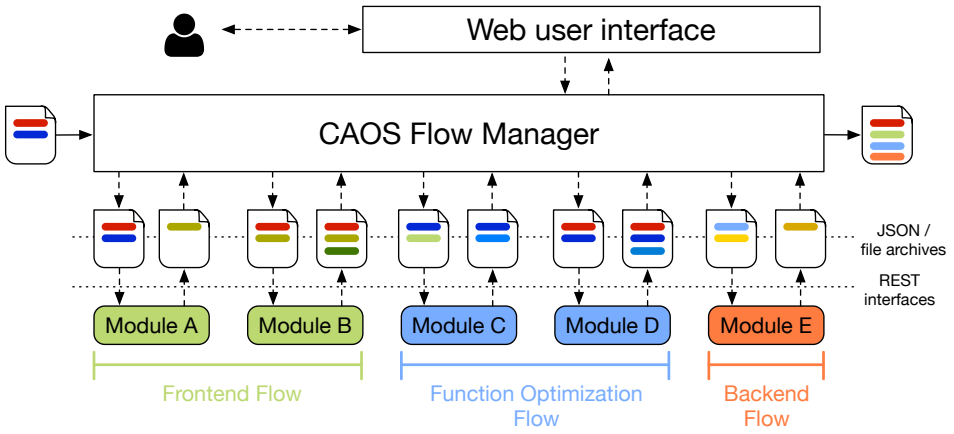


Figure 2.5: *Representation of the CAOS infrastructure in terms of its main components and their interaction. The CAOS modules exchange structured JSON files and raw file archives with the CAOS flow manager by means of REST APIs. The CAOS flow manager holds the state of the current project and handles the execution of the different modules when required. Finally, the user interacts with CAOS through a web interface that, in turn, send requests and read the state of the current project from the CAOS flow manager.*

backend are very dependent on the specific architectural template and inter-dependent from each other. Additionally, while we envision architectural templates that make explicit use of PDR and hence require the scheduling and mapping as well as the floorplanning phases extensively, the architectural templates currently integrated into CAOS do not leverage such features. Nevertheless, subsequent version of the platform will consider splitting the phases in completely independent modules. Chapters 6 and 7 shows mapping and scheduling as well as floorplanning methodologies that can be leveraged by architectural templates that exploits PDR, furthermore, these chapters also formally define the input and output required by both phases.

2.4 CAOS infrastructure

The principles described in Section 2.2 and the modular design depicted in Section 2.3 help us in the definition of an *as-a-Service* architecture for the CAOS platform. On one hand, the platform has to enable a fast and reliable design process possibly relying on a cloud-oriented infrastructure. On the other hand, the platform should provide sufficiently high scalability and sufficient performance to support several users at a time in a *multitenant*

environment.

To this aim, the CAOS platform is organized as a microservices architecture, leveraging Docker [32] application containers to isolate modules and to provide scalability to the whole infrastructure. Each module is deployed in a single container, and several implementations of the same module can coexist to provide different functionalities to different users. Moreover, each module implementation can be replicated to scale horizontally depending on system load. The modules are connected together and driven by the *CAOS Flow manager* shown in fig. 2.5, which serves the User Interface (UI) and provides the glue logic that routes each request to the proper module.

The interaction between the flow manager and the CAOS modules is performed by means of data transfer objects defined with a JSON DSL. The flow manager is aware of the full state of the design flow and is in charge of: 1) requesting the needed input from the user (e.g. the application description), 2) generating the specific JSON input files needed by the modules, and 3) collecting the output from the specific modules and integrate them in the state of the current design flow. The user can specify at each phase the desired module implementation, depending on several factors like architectural templates, software languages or code profilers, and the platform will take care of routing the request to the proper module automatically. Moreover, the platform can leverage modules deployed remotely by simply specifying their IP address. Another advantage of the proposed infrastructure is that, thanks to Docker containers, it can also be easily deployed on cloud instances, possibly featuring FPGA boards, such as the Amazon EC2 F1 instances. This allows a complete design process in the cloud in which the user can optimize the application through a web UI, while the final result of the CAOS design flow can be directly tested and run on the same cloud instance.

2.5 Modules integration

One of the goals of the CAOS platform described in section 2.2 is to let experienced users and developers to integrate their tools in the design flow and to provide a general interface that can be used to expand the platform. Up to now, CAOS supports the integration of new implementations of the currently available modules specified in section 2.3. Each module can be reimplemented integrating existing tools, as well as completely new implementations. The developers are free to adopt the best tool that fits their needs, as long as the module provides the implementation of the REST

Table 2.1: *APIs a module should support to integrate with the CAOS flow manager*

API name	REST API type	URI structure
info	GET	/info
submit	POST	/submit
task state	GET	/state/<taskId>
log	GET	/log/<taskId>
result	GET	/result/<taskId>/<filename>

APIs described in table 2.1. This way, the CAOS platform can reach and communicate with the module knowing only its IP address and default port.

The CAOS module APIs describe a wide set of operations that can be done with a module. At first, the *info* API allows the flow manager to know the functionality provided by the module, as well as the expected input data (either JSON files or file and code packages) and the number of tasks the module can execute in parallel. The *submit* API let the flow manager send data to be computed by the module. This API typically supports huge files, as it can be used to send code packages or profiling datasets. The *task state* API queries the module at a regular time basis, looking for task status and task completion notifications specifying the task id as input. At the same time, the flow manager queries the module asking for logging data with the *log* API. Finally, when the module completes its job, the flow manager retrieves the result of the computation by means of the *result* API, specifying the task id and the name of the file to be retrieved.

The APIs defined in Table 2.1 are shared by all the CAOS modules. However, each module type has its own specification in terms of the structure of the JSON file content to transmit and receive from the CAOS flow manager, as well as whether it also need to receive input files and provide raw files as output. As an example, the hardware resources estimation module, receives from the CAOS flow manager: the architecture description, the CAOS IR, the list of target functions to accelerate on hardware and a raw archive containing the application sources. The module provides as output a JSON file containing the estimated amount of resources used by each candidate function. Listing 2.3 shows an example of an input JSON file that is sent from the CAOS flow manager to the hardware resources estimation module, while Listing 2.4 shows the response sent by the module after completion.

Listing 2.3: *Simplified example of an input JSON file provided to the hardware resources estimation module in the function optimization flow. The architecture description is provided in the "architecture" dictionary. The program description is contained in the "language" and "supportedCompilers" keys. "codeArchive" specifies the name of the archive containing the application code. "functions" and "callgraph" contain the CAOS IR, while "architecturalTemplate" contains information on the functions that need to be accelerated in hardware.*

```
{
  "architecture": { ... },
  "language": "c++",
  "supportedCompilers": [ ... ],
  "codeArchive": "code.tar.gz",
  "functions": { ... },
  "callgraph": { ... },

  "architecturalTemplate": {
    "targetConfiguration": {
      "devices": [
        "f1-fpga-instance"
      ]
    },
    "supportedDevices": [
      "f1-fpga-instance"
    ],
    "id": "MasterSalve",
    "functions": {
      "vector_add(int*, int*, int*)": {
        "hardwareAcceleration": true
      },
      "main": {
        "hardwareAcceleration": false
      }
    },
    "type": "MasterSlave"
  }
}
```

Listing 2.4: *Example of an output JSON file generated by the hardware resources estimation module in response to Listing 2.3.*

```
{
  "vector_add(int*, int*, int*)" : {
    "resourceEstimation" : {
      "f1-fpga" : {
        "DSP48E":10,
        "BRAM_18K":34,
        "LUT":90,
        "FF":7,
        "URAM":0
      }
    }
  }
}
```

2.6 Final remarks

In this chapter we presented the motivations and the design flow of CAD as an Adaptive Open-platform Service (CAOS), a platform designed to provide a fully integrated development experience for accelerating applications on reconfigurable hardware. The platform has been conceived to automate or heavily assist all the steps involved in the development flow, whereas its infrastructure facilitates external researches in integrating their own custom modules. In the context of CAOS, we also introduced the concept of *architectural template* as a mean to address the complexity in accelerating high-level source codes. Within the architectural template applicability check phase, CAOS can validate which architectural template, among the ones available, can be adopted in order to accelerate a specific function. This opens also the possibility to extend the platform to support additional architectural templates that can also leverage on advanced FPGA features such as PDR. In Chapters 3 to 5 we describe the architectural templates currently available in CAOS, while Chapters 6 and 7 cover floorplanning and mapping and scheduling techniques that can be adopted to support PDR within CAOS.

CHAPTER 3

Master/Slave architectural template

This chapter describes the Master/Slave architectural template for the acceleration of a relatively large set of C/C++ codes on the Field Programmable Gate Array (FPGA) reconfigurable logic. In particular, after having defined the target communication model, system architectures and codes, we propose a semi-automated design space methodology to optimize the performance of the accelerator based on High-Level Synthesis (HLS). Finally, the chapter discusses the integration of the architectural template within the CAOS platform presented in Chapter 2 and reports experimental evaluation an the N-Body physical simulation.

3.1 Introduction

One of the crucial aspects when designing FPGA-based hardware accelerators is the management of the data transfer to and from the host system. Indeed, the communication model has a direct impact on how the accelerator can perform the actual computation.

When we consider accelerating high-level software functions on FPGA, the selection of a specific communication model can make the difference between a relatively straight-forward HLS translation, the need to perform major code restructuring to adapt the computation for the communication model, or being inherently not able to map the computation with the specific communication model.

In this chapter, we describe the Master/Salve architectural template which goes in the direction to ease the HLS process as well as to support a relatively large set of C/C++ codes. The architectural template makes use of memory mapped Master/Slave interfaces for the data transfer between the accelerators and a global memory that can be also accessed by the host system. The generality offered by the memory mapped interfaces allows, in most cases, to accelerate software functions on FPGA by means of HLS with relatively small changes to the original source code.

We start the chapter with Section 3.2 that provides a detailed description of the target architecture and communication model of the Master/Slave architectural template which is key for allowing the translation of a large set of C/C++ functions to a corresponding Hardware Description Language (HDL) implementation. More details about the types of computations and source codes supported by our methodology are described in Section 3.3. Then, in Section 3.4 we present the proposed design space exploration for optimizing the performance of the hardware implementation of the candidate functions, while in Section 3.5 we discuss on how we integrated the Master/Slave architectural template within the CAD as an Adaptive Open-platform Service (CAOS) platform. Section 3.6 showcases the results achieved by leveraging the Master/Slave architectural in CAOS on the N-Body simulation case study. Finally, Section 3.7 concludes the chapter.

3.2 Communication model and target systems

The Master/Slave architectural template targets systems that provide a shared Double Data Rate (DDR) memory that can be accessed both by the host running the software portion of the application and by the FPGA devices on which we implement our accelerated functionalities (also referred as

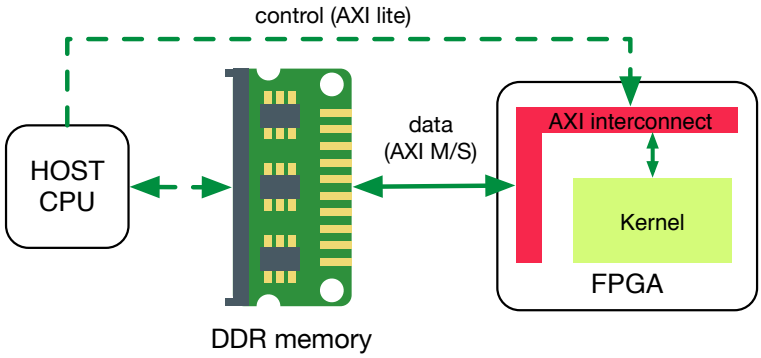


Figure 3.1: *Communication model of the Master/Slave architectural template: the data transfer between the host and the accelerator is accomplished via a shared DDR memory. The accelerated kernel function has access to the DDR memory via an AXI4 Master/Slave memory mapped interface, while the host system controls the execution of the accelerator through an AXI4-Lite interface.*

kernel). As an additional constrain, the template also requires that the communication between the accelerator and the DDR memory is performed via memory mapped interfaces. Such requirements allow to standardize the data transfer to and from the hardware accelerator as well as to support random memory accesses to pointer arguments of the target C/C++ function.

Figure 3.1 shows the communication model of the Master/Slave architectural template. The host system is responsible for providing the input data for the accelerator on the shared DDR memory and also controls the execution of the accelerator. Once the accelerated computation is completed, the result is provided on the shared DDR memory and made available to the host system for further processing.

In recent years, FPGA vendors have provided a vast variety of computing systems tightly integrated with FPGA devices that find applications in domains ranging from High Performance Computing (HPC) [92, 93], cloud [26, 84] and embedded systems [43, 81]. Among the systems that provide a suitable solution to implement the Master/Slave communication model, we currently support two system families, namely the Xilinx Zynq System-on-Chip (SoC) which is widely adopted in the embedded domain and the recently released Amazon EC2 F1 instances for high performance accelerated cloud computing.

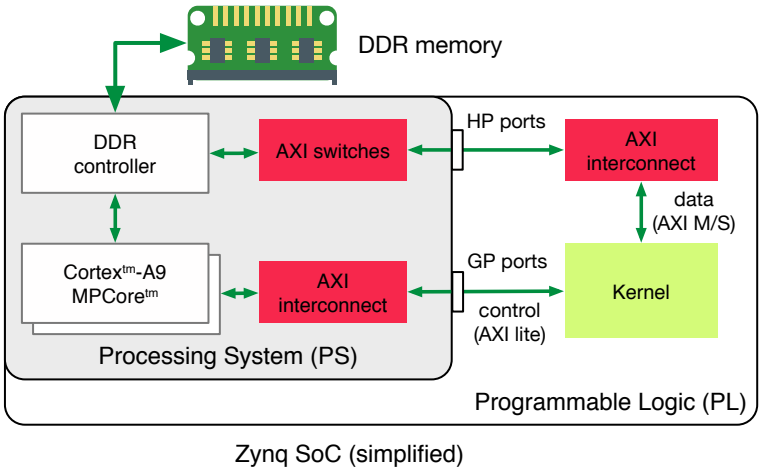


Figure 3.2: *Host and accelerator system architecture for the Xilinx Zynq SoC.*

3.2.1 Xilinx Zynq SoC

A simplified representation of the Xilinx Zynq SoC architecture is depicted in Figure 3.2. The Zynq SoC consists of two main components: the Programmable Logic (PL), containing the FPGA programmable resources, and the Programmable System (PS), featuring two ARM Cortex^{tm}-A9 processors connected in a multi-processor configuration sharing a 512KB L2 cache. The PL has also access to the shared L2 cache via an AXI4 64bit master interface via General Purpose (GP) ports. Such ports provide a convenient way to control the kernel via AXI4-Lite interface from the PS. Furthermore, the PL has also a set of dedicated High Performance (HP) AXI4 master ports that enable direct access to the external DDR memory.

In the context of the Master/Slave architectural template, the ARM Cortex^{tm}-A9 within the PS represents the host system, while the PL contains the accelerated kernel implemented on the reconfigurable logic. The host system is in charge of loading the necessary input data on the external DDR memory and to start and monitor the execution of the kernel. On the other hand, the kernel reads the data from the external memory via the AXI4 Master/Slave interface connected to the HP ports and writes back the result to the memory once available.

3.2.2 Amazon EC2 F1 instances

Amazon has been one of the first companies to introduce cloud computing with FPGAs, opening the world of reconfigurable computing to the masses

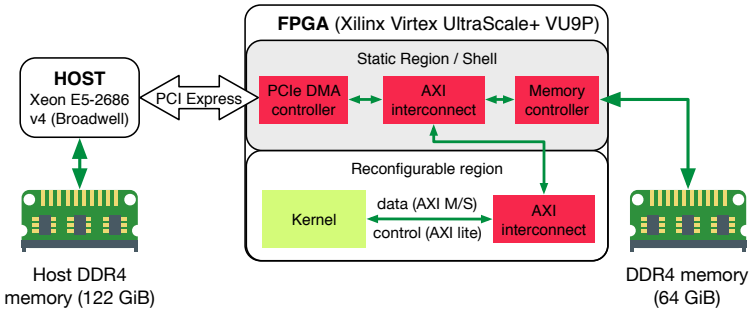


Figure 3.3: *Host and accelerator system architecture for an Amazon Elastic Compute Cloud (EC2) F1 instance equipped with a single FPGA.*

and allowing users to focus only on creating and optimizing applications, rather than spending time in installing the devices and the relative drivers. Amazon EC2 F1 instances are, in fact, compute instances equipped with Xilinx UltraScale+ VU9P FPGAs. They allow to launch multiple instances equipped with 1, 2 or even 8 FPGA devices connected via PCIe to the host system. Figure 3.3 represents the interconnection of the host and FPGA components within an Amazon EC2 F1 instance equipped with a single FPGA. The Intel Xeon host processor has a dedicated DDR memory and communicates with the FPGA fabric via a PCIe x16 gen3 interface. The FPGA resources are logically divided into a static region and a reconfigurable region. The static region, also referred as shell, contains the PCIe DMA controller to interface the host system and a memory controller to access a dedicated DDR4 memory. Furthermore, the static region also provides AXI4 ports to interface the accelerated kernel function within the reconfigurable region.

3.3 Target computations

The Master/Slave architectural template supports most of the features of C/C++, however it still requires some restrictions in order to ensure that the code can be successfully synthesized with Vivado HLS [111], which is leveraged by the architectural template in order to generate the final HDL. Furthermore, we also consider restrictions that apply for using the code within the SDAccel [109] workflow, required when targeting Amazon EC2 F1 instances.

3.3.1 Code requirements

The granularity for software acceleration is at the function level. In particular, we refer to the target function for hardware acceleration as *top function*. All the functions called either directly or transitively from the top function are defined *sub-functions*, while the *function tree* represents the set of the *sub-functions* together with the *top function* itself. Given these definitions, a *top function* have to satisfy the following conditions in order to be supported by the Master/Slave architectural template:

- The *function tree* cannot access globally defined variables.
- There must be no cycles in the static callgraph of the *function tree* (i.e. no recursion).
- C constructs within the *function tree* must be of a fixed size (e.g.: local array sizes should be fixed).
- Pointer casting in the *function tree* must occur only on native C types, while array of pointers cannot point to additional pointers.
- The *function tree* must not dynamically create or destroy objects (i.e. *new* and *delete* are not supported) nor using dynamic polymorphism and dynamic virtual function calls.
- There must be no *syscall* in the *function tree*.
- There must be no function pointers in the *function tree*.
- The *top function* return value must be `void`. This is needed to simplify the interface generation for the SDAccel flow.
- The *top function* arguments must be either scalar or arrays with specified size. Although the size information is not required by the C standard, it is leveraged by the template in order to identify a bound on the data transfer size and it is useful for optimization purposes.

The previously described requirements allow the architectural template to synthesize and implement the target function tree, provided that enough resources are available within the FPGA. Nevertheless, as we will discuss in the next section, the design space exploration leverages on latency estimates in order to evaluate the quality of different optimizations. If the code contains data-dependent loop bounds, such estimates might not be available, since the execution time of the function largely depends on information not available at compile-time. Hence, in order to evaluate different

optimizations, we also require that trip count bounds can be inferred for all the loops of the *function tree*.

It is possible to verify the latter condition with scalar evolution analysis. The scalar evolution framework [104], allows to identify how scalar variables (such as integers) evolves throughout the iterations of a loop. In particular, by applying such analysis together with the identification of the loop induction variables and by analyzing the exit condition of the loops, it is possible to infer the loop trip count bounds, provided they do not depend on runtime conditions. Loop trip count bounds can be easily obtained from Low-Level Virtual Machine (LLVM) by running the LLVM `opt` command on the LLVM Intermediate Representation (IR): `opt IR.ll -scalar-evolution -analyze` where `IR.ll` is the LLVM IR of the corresponding C/C++ function. The LLVM IR can be obtained running: `clang -emit-llvm -O1 -S SOURCE.c -o IR`, where the `O1` flag is needed in order to apply some basic optimizations to the code that allow to improve the ability to infer loop trip count bounds with scalar evolution.

3.3.2 Candidate applications

Despite the relatively large set of supported C/C++ codes, the Master/Slave architectural template is not be beneficial for all kinds of applications from a performance perspective. In particular, the efficiency of memory transfers usually have a large impact on the overall execution time of the accelerated top function. Hence, as we will discuss in the next section, the Master/Slave architectural template applies caching optimizations so that the data is transfered in burst mode from the DDR and stored locally on device before the actual computation. While such an approach improves memory transfer, it also splits the computation in three sequential stages: read input data from DDR, compute, store output data to DDR. This solution is viable for applications that are compute-intensive and are structured so that the dataset can be split in blocks that fit within the device local memory. An ideal test case is the N-Body application discussed in 3.6. Indeed, for an input of size $\mathcal{O}(N)$, the computation requires $\mathcal{O}(N^2)$ operations. Furthermore, the computation can also be tiled so that it can execute B^2 runs on an input of size $\mathcal{O}(k)$, with $k = \frac{N}{B}$ that require $\mathcal{O}(k^2)$ steps to compute.

Overall, other examples of compute-intensive applications that can benefit from the Master/Slave architectural template are: alignment algorithms, such as Smith-Waterman [30] and PairHMM [90] from the computational biology domain, Quantum Monte Carlo simulations [14] in the computa-

tional chemistry domain, as well as dense linear algebra kernels.

It is also worth mentioning that, compared to other architectures such as Graphics Processing Units (GPUs), FPGAs deliver the best advantages in terms of performance and energy efficiency when dealing with reduced integer data types, as for the Smith-Waterman algorithm where 2 bits are sufficient to encode elements from a DNA sequence. Indeed, compared to general purpose architectures, the FPGA can make more efficient use of the available hardware resources by implementing specialized operators for reduced bitwidth.

3.4 Design flow

The design flow for the Master/Slave architectural template can be divided in 3 main steps. In the first step, we validate whether the target function meets the requirements discussed in Section 3.3 and generates an initial implementation of the accelerator. Afterwards, the flow moves to the actual design space exploration in which we iteratively explore and evaluate multiple optimizations. Finally, the last phase takes care of generating the necessary files to implement the system, as well as the updated host code for the application.

3.4.1 Validation and initial design generation

Once having validated that the target *top function* meets the requirements defined in Section 3.3, we need to generate a first version of the code that can be synthesized through Vivado HLS. This translates into defining the Master/Slave interfaces for the function by means of HLS pragma, since our requirements already ensure that the code can be synthesized by Vivado HLS. More specifically, we map each array argument to an AXI4 Master interface and create an additional AXI4-Lite slave interface to configure the offset address at which the data should be read or written. On the other hand, we only define an AXI4-Lite interface for each scalar argument. Finally, we also create an AXI4-Lite interface to control the execution of the kernel and its status. An example of the Master/Slave interfaces defined for a simple vector sum code is provided in Listing 3.1.

The proposed scheme for mapping arguments to AXI4 and AXI4-Lite interfaces allow to support the accelerator also within the SDAccel flow and hence to target Amazon EC2 F1 instances.

Listing 3.1: *Simple C code with pragma defining the Master/Slave interfaces.*

```
void vector_add_kernel(int c[SIZE], int a[SIZE], int b[SIZE]) {
  #pragma HLS INTERFACE m_axi port=c bundle=gmem
  #pragma HLS INTERFACE s_axilite port=c bundle=control
  #pragma HLS INTERFACE m_axi port=a bundle=gmem
  #pragma HLS INTERFACE s_axilite port=a bundle=control
  #pragma HLS INTERFACE m_axi port=b bundle=gmem
  #pragma HLS INTERFACE s_axilite port=b bundle=control
  #pragma HLS INTERFACE s_axilite port=return bundle=control

  for (int i = 0; i < SIZE; i++) {
    c[i] = a[i] + b[i];
  }
}
```

3.4.2 Design optimization

The proposed design space exploration operates as shown in Algorithm 1. The process iterates over the set of available optimizations, and, for each optimization, generates a set of new candidate implementations starting from the current one. Every implementation is then evaluated in terms of resources and expected latency by running Vivado HLS and parsing the generated resource and performance reports. Then, for each optimization, the best implementation that meet the resources available on the target FPGA are considered as next candidates.

After having populated the set of candidate implementations C, the user has the possibility to select the preferred implementation by considering both estimated performance and resource occupation (`selectImplementation` procedure). Additionally, the user can also decide whether to repeat the optimization process to search for further optimizations or stop the exploration and proceed with the implementation phase. An important aspect of the design process that does not emerge directly from Algorithm 1, is that the set of candidate implementations generated for an optimization can be very dependent on the current implementation. Furthermore, the application of an optimization o_1, can affect the generation of the candidates for a different optimization o_2 in the next iteration. As an example, after having optimized the data transfer and cached the data on the on-chip memory, it might be beneficial to partition the newly created local memories to optimize parallel accesses.

If, at every iteration the designer selects the best implementation in terms of performance, Algorithm 1 can be regarded as a greedy exploration that, at each step, considers the optimization that provides the highest performance benefit. Nevertheless, while such approach might not lead to the

Algorithm 1 Design space exploration for the Master/Slave architectural template

```
1: do
2:     C ← ∅
3:     for each o ∈ Optimizations do
4:         C_o ← ∅
5:         I ← o.generateCandidates(currentImplementation)
6:         for each i ∈ I do
7:             i.perf← evaluatePerformance(i)
8:             i.occupancy← evaluateOccupancy(i)
9:             if i.occupancy< 100% then C_o ← C_o ∪ i
10:        if C_o ≠ ∅ then C ← C ∪ {i ∈ C_o | i.perf ≥ j.perf ∀j ∈ C_o}
11:    currentImplementation, repeat←selectImplementation(C, currentImplementation)
12: while repeat
```

optimal implementation, it is worth noting that many paths in the design space lead to the same solution. Indeed, if we assume that the best design is the result of the application of a set of optimizations O, the order followed by the design space exploration to identify the set O does not affect the quality of the solution. Furthermore, the considered optimizations are very likely to locally improve performance, hence, it is unlikely that a path from an implementation i_0 to a better implementation i_n is not monotonic in terms of performance improvement. Said in other words, it is unlikely that in order to reach an optimal solution we locally need to select optimizations that worsen the current performance, hence justifying the applicability of a greedy approach to achieve good solutions.

The reason why we prefer to let the user select the next implementation, instead of automatically selecting the best local optimization, is to give more control over the design exploration process. Indeed, in most cases, optimized designed tend to require longer synthesis time, but the user might be satisfied with a sub-optimal solution as long as it meets the target performance. Furthermore, more experienced designers might want to guide the order in which the optimizations are applied to converge faster to an optimal solution. In addition, we also allow the user to restart the design process from an intermediate implementation in order to explore different paths if required.

In the next sections we give more details on the specific optimizations that are currently supported within the Master/Slave architectural template and on how new candidate implementations are generated.

On-chip caching

In most cases, trying to synthesize directly a C/C++ code leads to poor memory transfer performance. Indeed, if memory accesses are not strictly consecutive in time, Vivado HLS treats such accesses as single small transfers with high overhead. One way to improve memory transfers and minimize the overhead is to copy consecutive data to local on-chip memories in burst mode. Vivado HLS, unless instructed differently, automatically implements local arrays to on-chip Block RAM (BRAM). Hence, in order to support memory bursts, we declare a local array for each corresponding array argument and make use of `memcpy` calls to copy input/output data from/to the array arguments to/from the local arrays. The size of the memory transfers for the `memcpy` calls can be inferred from the size of the array arguments provided by the user and from their base types. Finally, we rewrite the original accesses to the array arguments into accesses to the local arrays. With this approach, we effectively fully cache the input and output data of the function and make use of `memcpy` for which Vivado HLS is able to infer memory bursts. Notice that it might not be feasible to copy the entire working set on the on-chip memory since BRAM are usually limited to a few MBs per device. Recent devices however also provide larger on-chip memories such as the Ultra RAM (URAM) from Xilinx UltraScale+ devices. Such memories currently allow to store 10s of MBs and their usage can be enforced via HLS pragma. If there are enough memory resources to perform caching and if the optimization is not already applied, a candidate implementation is generated. On the other side, if, even by relying on URAM, there is not enough storage, we do not generate any candidate optimization. Future versions of the optimization might look into finer grained approaches, such as leveraging more complex caches [57] or perform partial caching.

Pipelining / unrolling

One among the most critical optimizations when dealing with HLS for C/C++ functions is loop pipelining. Indeed, in most cases, the highest amount of time is spent in loops, whose optimization is hence crucial. Loop pipelining [117] allows to improve the overall execution time of a loop by pipelining the execution of subsequent loop iterations. The number of cycles between an iteration and the next one is referred as Initiation Interval (II). If we consider a loop that needs L cycles to complete one iteration and iterates N times, the overall number of cycles t of the pipelined version of the loop reduces from $L \cdot N$ to [19]:

$$t = II \cdot (N - 1) + L \tag{3.1}$$

When the ideal $II = 1$ is achieved, we obtain the maximum pipelining performance with an approximate speedup of L times (if we assume $L << N$). Depending on the specific code of the loop, it might not be possible to achieve an ideal II of 1 cycle. Common reasons for this are loop carried dependencies, which require data from previous loop iterations to be computed before starting the next iteration, or multiple read/write accesses to the same BRAM in parallel, that might not be performed in the same cycle due to limited memory ports.

Vivado HLS allows to apply the pipelining optimization to a given loop simply by specifying the `HLS PIPELINE` pragma within the target loop body. While it is relatively simple to test the pipeline optimization on a single loop, handling loop nests with high nesting level or non-perfect loop nests, such as the one that occurs in the image processing domain, requires additional care. Indeed, when pipelining is applied to a loop nest at level i, all the inner loops at level $i' > i$ will be automatically unrolled in order to support pipelining at level i. This, in general, leads to higher resource utilization and potentially higher performance. The performance improvement depends on whether the unrolled operations can be scheduled in parallel running on multiple modules and effectively pipelined across the iterations of the loop. Furthermore, pipelining can be applied even if the loop contains a call to another function. In this case, a simple approach is to inline the called functions and treat the resulting code as in the previous case. Examples of Vivado HLS codes featuring loop pipelining and loop unrolling are shown in Listing 3.2, while Figure 3.4 shows the scheduling of the operations according to the specified optimization.

Within the Master/Slave architectural template we explore pipelining opportunities for all the available loop nests within the *function tree*. More specifically, we first logically collapse the *function tree* by inlining all the *sub-functions* within the *top function*. Afterwards, we identify the available loop nests with a direct search in the source code performed with the support of *clang* [55]. Then, for each loop within each loop nest, we generate a candidate pipelined implementation. Notice that loops for which pipelining was already applied are skipped together with their nested loops. Finally, in order to select the best pipelining candidate, we evaluate the optimizations starting from innermost loops towards outermost loops. Indeed, if we verify that pipelining a loop at level i produces solutions that exceed resource requirements or for which Vivado HLS is not able to provide a schedule, we can avoid exploring pipelining optimization at lower nesting levels (i.e.

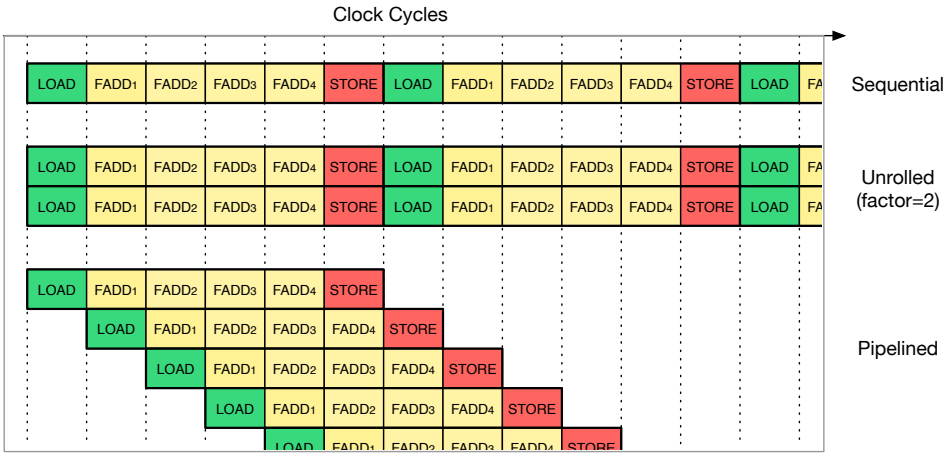


Figure 3.4: *Scheduling for the sequential, pipelined and partially unrolled loops presented in Listing 3.2. Here we assume that a pipelined floating-point adder with 4 stages is used and that load and store operations to local memories require 1 clock cycle. Note that a single floating-point adder is needed for the sequential and pipelined loops, while the partially unrolled loop requires 2 floating-point adders as well as enough memory ports to perform the read and store operations in parallel.*

Listing 3.2: *Vivado HLS C code snippet performing a floating-point vector sum with different implementations: (a) sequential loop, (b) partial loop unrolling, and, (c) loop pipelining.*

```
#define N 256

...
float A[N], B[N], C[N];
...

// (a) Sequential loop
for (int i = 0; i < N; i++) {
  A[i] = B[i] + C[i]
}

// (b) Loop unrolled by a factor of 2
for (int i = 0; i < N; i++) {
  #pragma HLS UNROLL factor=2
  A[i] = B[i] + C[i]
}

// (c) Pipelined loop
for (int i = 0; i < N; i++) {
  #pragma HLS PIPELINE
  A[i] = B[i] + C[i]
}
```

$i' < i$), since such solutions would require higher resource requirements while being also more complex to schedule.

Memory partitioning

As previously discussed, when applying the pipelining optimization, it might not be possible to achieve the ideal II of 1 cycle. With the memory partitioning optimization, we specifically target the case in which the bottleneck for pipelining performance is due to conflicting accesses to the same memory ports.

In order to better understand how conflicting accesses occur it is useful to consider how the on-chip FPGA memories are used in HLS. Each on-chip BRAM provides a small storage capacity and a fixed number of memory ports. In Xilinx devices, each BRAM can host 18Kbits of data and provides two independent memory ports that can be used in different configurations. Larger memories can be created by cascading multiple BRAMs, this effectively increases the storage capacity, but the number of available memory ports remain fixed. When a local array a of size S is implemented in Vivado HLS, the tool automatically allocates a large enough memory to store all the S elements by cascading multiple BRAMs. Hence, it is clear that, the maximum number of parallel accesses that can be performed to array a in a single cycle corresponds, at most, to the number of memory ports available. As an example, consider the pipelined matrix-vector product shown in Listing 3.3. The input matrix A of size $M \times N$ as well as the input vector b and the result vector c are declared as local arrays that will be implemented as on-chip memories. The code perform the computation $c = A \cdot b$, the N elements in vector c are computed in pipeline performing all the M multiply-accumulate operations in parallel. Notice that this code cannot achieve an II of 1, as this would require to access all the elements of b and each element in a column of matrix A in parallel at every cycle. However, only two memory ports are available to read data from A and b.

To overcome this limitation, we can apply memory partitioning to split the content of a large memory into smaller memories, so that data in the resulting smaller memories can be accessed in parallel. Vivado HLS provides different partitioning schemes for each array dimension, namely: *block partitioning*, *cyclic partitioning* and *complete partitioning*. The number of memory N into which the data should be stored, is referred as *partitioning factor* and can be specified along with block or cyclic partitioning. *Block partitioning* stores contiguous block of data in each of the N memories, while *cyclic partitioning* splits the data in a round-robin fashion over

Listing 3.3: *Vivado HLS C code snippet performing a pipelined matrix-vector product.*

```
#define N 256
#define M 16

...
float A[N][M], b[M], c[N];
...

for (int i = 0; i < N; i++) {
  #pragma HLS PIPELINE
  float prod = 0;
  for (int j = 0; j < M; j++) {
    prod += A[i][j]*b[j];
  }
  c[i] = prod;
}
```

the N memories. Finally, *complete partitioning* stores each element of the memory into a single memory or register, depending on its size. The described partitioning schemes can then be applied to each dimension of a local multi-dimensional array independently, providing a rich set of possible memory partitioning strategies. Notice that while memory partitioning increases the available memory ports, it also increases resource usage as the data tends to be fragmented in each target memory.

Within the Master/Slave architectural template we added the capability to automatically detect memory partitioning optimizations. In particular, we first identify all the local arrays together with their sizes in the *function tree* by means of the LLVM IR, and, subsequently, we run Vivado HLS on the current implementation. When Vivado HLS is not able to achieve the ideal II due to limited memory ports, it reports a message in the synthesis log with information on the involved arrays and the line of code at which the conflict arises. Hence, we parse the HLS log and, among the arrays identified at the first step, we consider only those for which memory partitioning can be beneficial. For each candidate array and dimension, we generate a candidate implementation in which the dimension of the array is partitioned completely. Finally, the best implementation is selected according to Algorithm 1.

3.4.3 System implementation

The artifacts generated by the proposed design flow depends on the target architecture. Currently the template supports Amazon EC2 F1 instances and SoCs from the Xilinx Zynq-7000 family.

When targeting the F1 instances, the original code is modified to support both host compilation and HLS synthesis within the SDAccel toolchain. This is done with the insertion of C preprocessor directives that hides or enable parts of the source code as necessary. The `main` function of the application is modified so that, before performing the actual execution, it makes use of the OpenCL Application Programming Interfaces (APIs) provided by SDAccel to detect the FPGA board and load the accelerator bitstream from the AWSXCLBIN file. On the other hand, the original *top function* is wrapped in a function that, by means of the OpenCL APIs, takes care of creating the buffers, copying the input data, starting the accelerator, and copying back the results from the board memory to the host memory. Finally the template also provides a *Makefile* that can be used to generate the XCLBIN file (a checkpoint of the accelerator design after place and route) through the SDAccel toolchain and to compile the host executable. In order to run the FPGA code on the hardware accelerators available on Amazon EC2 F1 instances, the user has to perform a few extra steps to generate the AWSXCLBIN from the XCLBIN binary generated by SDAccel, which involve running predefined scripts and commands provided by Amazon.

With regards to the Xilinx Zynq platform, the changes applied to the original code goes in the same direction of the Amazon EC2 F1 case. The flow modifies the code to support HLS synthesis of the *top function* and integrate the needed device initialization within the `main` function. However, in this case, the flow also runs Vivado HLS and integrates the resulting HDL implementation within a Vivado design. The design is created automatically through *TCL* scripts and includes the AXI memory interconnect, the Zynq subsystem and the Memory Interface Generator (MIG) module as well as the accelerator resulting from the high-level synthesis of the *top function*. Finally, the user can generate the FPGA bitstream through Vivado and compile the modified host code on the ARM processor available on the Zynq SoC.

3.5 Integration in CAOS

The design flow presented in Section 3.4 assumes that the designer already decided which function to accelerate in hardware, but does not provide any guidance on how to select it. On the other hand, the CAOS framework presented in Chapter 2 supports all the steps needed to accelerate a software application on reconfigurable hardware, including the profiling of the application and support for HW / SW partitioning. In order to combine the benefits of the CAOS platform with the proposed design flow, we inte-

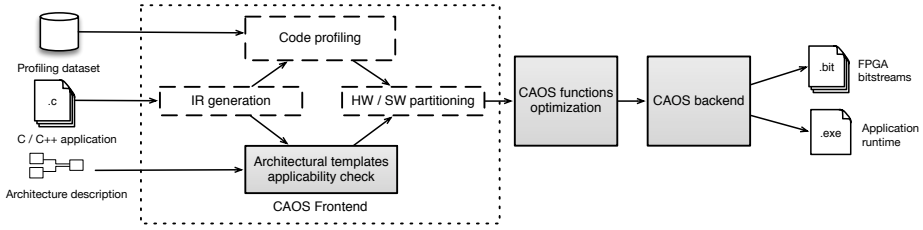


Figure 3.5: *Integration of the Master/Slave architectural template within the CAOS platform. The shaded boxes with solid lines represent modules and components of the CAOS framework that have been modified in order to perform the integration.*

grated the Master/Slave architectural template in the CAOS platform. More specifically, Figure 3.5 shows the modules and flows of the CAOS platform that has been impacted by the integration.

CAOS offers two solutions for extending the support to a new architectural template. The first, is to create brand new modules and specify in the CAOS User Interface (UI) the proper IP addresses, the second, consists in extending the default modules to perform dedicated tasks depending on the architectural template being considered. The first option might be preferred for external researchers willing to create their own modules, nevertheless, for our scenario, we decided to integrate the functionality directly in the default CAOS modules, so that the user can benefit from the architectural template without the need to specify any custom IP address in the CAOS UI. In the next sections we provide more details on how the different stages of the Master/Slave design flow have been cleanly mapped to modules of the CAOS platform.

3.5.1 CAOS frontend

In order to add a new architectural template within the CAOS frontend it is only necessary to implement or modify the *architectural template applicability check* module. Indeed, the other frontend modules are mostly independent from the architectural template being considered. In our scenario, we extended the *architectural template applicability check* module to validate whether the architecture description provided by the user matches one of the Amazon EC2 F1 or Zynq-7000 systems described in Section 3.2. If the check is successful, we then scan all the functions of the application, by leveraging on the CAOS IR, and verify the constraints described in Section 3.3 with the help of LLVM. Finally, for the functions that satisfy all the requirements, we also run Vivado HLS to make sure that the tool is able to generate an HDL implementation for the initial version of the function.

As a result, the module returns if the Master/Slave architectural template is supported for the current application and target hardware description, and, if so, returns the list of functions that can be accelerated in hardware as well as diagnostic information for those functions that do not meet all the requirements.

3.5.2 CAOS functions optimization

The resulting *function tree* and corresponding *top function* identified in the CAOS frontend, with the help of the profiling module and the HW / SW partitioning module, are provided to the CAOS function optimization flow. The Master/Slave design space exploration presented in Algorithm 1 can be easily factorized and mapped to the different CAOS modules discussed in Section 2.3.2. Hence, we extended the default CAOS modules to perform the steps needed by the Master/Slave architectural template, provided that the CAOS frontend validates its applicability.

Static code analysis

Within the static code analysis, we analyze the code in order to extract all the information required to generate the candidate optimizations in the subsequent modules. More specifically, we recover the following information:

- **Read/write analysis:** for each argument of the *top function*, we leverage on the LLVM IR to identify whether an argument is read, written or both read and written. This information can then be leveraged by the onchip-memory caching optimization to save copy time and storage space for parameters that are only read or written.
- **Local arrays identification:** we identify all the local arrays in the *function tree* as well as their sizes and number of dimensions, this information will be used to identify memory partitioning opportunities.
- **Loop nests identification:** the module identifies all the loop nests in the *function tree* and determines the trip count for each loop. Notice that the trip count information is guaranteed to be available due to the code requirements from the *architectural template applicability check* module. Such data can then by leveraged to explore loop pipelining optimizations.
- **Latency estimates:** the module generates an initial latency estimate by running Vivado HLS on the *top function*. Such data is useful in order to compare the performance of the initial implementation with respect to the ones achieved after applying a given optimization.

Resource estimation

The *resource estimation* module simply executes Vivado HLS on the *top function* and retrieve an estimates of the amount of resources required on the target FPGA. The estimation is done assuming a default frequency of 200 MHz which is reasonable for designs targeting Xilinx UltraScale+ devices. Notice that the implementation of this module can be easily replaced with more fast or accurate estimation techniques that to not necessarily rely on commercial tools.

Performance estimation

The *performance estimation* module has been extended to implement Algorithm 1. The generation of the new candidate optimizations leverages the data generated by the *static code analysis* and *resource estimation* modules previously discussed. After having generated the set of candidates, Vivado HLS is leveraged to evaluate the estimated latency and performance. Finally, the module returns the best candidate for each optimization type with their associated resource estimation, performance estimation and optimization parameters.

Code optimization

The code optimization module has been extended to implement the `select Implementation` method described in Algorithm 1. In particular, the user is given the opportunity to select the candidate optimization among the ones generated by the previous module. After having selected the optimization, the module inserts the necessary code pragmas or modify the code according to the optimization type and its parameters. After having applied the optimization the user can decide whether to iterate the optimization process or move to the backend flow.

3.5.3 CAOS backend

The CAOS backend has been extended to support the two implementation flows described in Section 3.4.3, targeting either Amazon EC2 F1 instances of Xilinx Zynq-7000 SoC. When the Master/Slave architectural template is used for the generation of the final system, the module takes care of generating the host code with the needed API calls to interface with the accelerator and either a Vivado project or a Makefile that can be used to generate the accelerator bitstream or XCLBIN file.

Overall, the integration of the Master/Slave architectural template within CAOS allows an application designer to easily go through all the needed steps for accelerating a relatively large set of C/C++ codes on two widespread systems featuring reconfigurable hardware. This allow the application design to focus more on the specific functionality to implement rather than manually optimize the implementation and rewriting the code to target a specific system.

3.6 Experimental results

Within this section we evaluate the Master/Slave architectural template integrated into the CAOS platform on the N-body physical simulation case study. We accelerate the algorithm by means of CAOS targeting a Xilinx UltraScale+ VU9P FPGA on an Amazon EC2 F1 instance. Finally, we compare the achieved results against the ones obtained by the bespoke implementation proposed in [25] and later enhanced in [26].

3.6.1 The N-Body simulation problem

The N-Body simulation is a well known problem in physics and has applications in many fields ([49,58,69]), such as fluid and molecular dynamics, plasma physics, electromagnetism and astrophysics. The problem consists in simulating the evolution of a system composed of N bodies interacting by means of pairwise forces, such as the ones due to the gravity. Each body is characterized by at least three properties: a position, a velocity and a mass. The simulation is performed by solving the motion equations for the bodies involved in the system at subsequent time steps.

In the literature, there exists different approaches to solve the N-Body simulation problem, nevertheless the *all-pair* method [27] is among the most generic ones, yet most compute intensive to perform. In our case study we target the acceleration of the *all-pair* method. The algorithm consists of two alternating functions executed at every timestep of the simulation: `forceComputation`, which computes the force applied to each body and updates the corresponding accelerations, and `updatePosition`, which updates the velocity of the bodies and their positions. The core of the reference software implementation is shown in Listing 3.4, in which the number of bodies N is a statically defined constant.

Listing 3.4: *Core of the considered N-Body simulation C application. This implementation operates on a system of N bodies, where N is a statically defined constant. Notice that smaller systems can be simulated by padding the* `mass` *array to 0.*

```
typedef struct {
  float x;
  float y;
  float z;
} coord3d_t;

void updatePosition(coord3d_t position[N], coord3d_t velocity[N],
    coord3d_t accel[N]){
  update_loop_1: for (int i = 0; i < N; i++) {
    // implicit delta_time = 1 for integrating position and velocity
    position[i].x += velocity[i].x;
    position[i].y += velocity[i].y;
    position[i].z += velocity[i].z;
    velocity[i].x += accel[i].x;
    velocity[i].y += accel[i].y;
    velocity[i].z += accel[i].z;
  }
}

void forceComputation(const float mass[N], coord3d_t position[N],
    coord3d_t accel[N], float eps){
  force_loop_1: for (int i = 0; i < N; i++) {
    force_loop_2: for (int j = 0; j < N; j++) {
      float rx = position[j].x - position[i].x;
      float ry = position[j].y - position[i].y;
      float rz = position[j].z - position[i].z;
      float dd = rx * rx + ry * ry + rz * rz + eps;
      float d = 1.0f / (dd * sqrtf(dd));
      float s = mass[j] * d;
      accel[i].x += rx * s;
      accel[i].y += ry * s;
      accel[i].z += rz * s;
    }
  }
}

void NBodySimulation(const float mass[N], coord3d_t position[N], coord3d_t
     velocity[N], float eps, int time_steps)
{
  coord3d_t accel[N];

  time_loop:for (int t = 0; t < time_steps; t++) {
    memset(accel, 0, N * sizeof(coord3d_t));
    forceComputation(mass, position, accel, eps);
    updatePosition(position, velocity, accel);
  }
}
```

3.6.2 Application acceleration trough CAOS

In order to process the application through CAOS, we prepared the following input resources:

- An archive containing the C source files of the application.
- The JSON program description file that specifies the programming language and the supported compilers with their flags.
- The dataset needed to profile the application.
- The JSON architecture description file targeting an Amazon EC2 F1 instance.

After having provided the necessary files to CAOS, we started the frontend flow. During the *architectural template applicability check* phase, CAOS validated the applicability of the Master/Slave architectural template and identified the `forceComutation` and `updatePosition` as candidate functions for hardware acceleration. Indeed, both functions abide by the constraints described in Section 3.3, while the provided architecture description is supported by the architectural template.

The *code profiling* module showed that the `forceComputation` takes the highest amount of time of the whole computation, this can also be noted by analyzing the complexity of the code in Listing 3.4. Indeed, given N bodies, the computational complexity of `forceComputation` and `updatePosition` is $\mathcal{O}(N^2)$ and $\mathcal{O}(N)$ respectively. Due to the highly unbalanced execution time between the two candidate functions, the *HW / SW partitioning* module suggested to accelerate `forceComputation` in hardware and this concluded the CAOS frontend flow.

Subsequently, during the CAOS functions optimization flow, the platform suggested to apply two optimizations to the `forceComputation` function, namely on-chip caching and pipelining loop `force_loop_1`, which also triggers full unrolling of the innermost loop `force_loop_2`. Notice that such optimizations achieved for values of N in the order of hundreds of bodies (such as 160 and 320 bodies). Indeed, very high number of N (e.g. more than 50000 bodies) might not allow to fully cache the data due to lack of local BRAMs and URAMs, while values of N in the order of 1000s would prevent the tool from pipelining the outermost loop, since it also requires to fully unroll the N iterations of the innermost loop, leading to unsustainable Digital Signal Processing (DSP), Look-Up Table (LUT) and Flip Flop (FF) resource requirements.

Finally, the CAOS backend flow produced the optimized Vivado HLS code for the `forceComputation` (shown in Listing 3.5), the host code for the application and the Makefile to generate the XCLBIN file and the executable for the Amazon EC2 F1 instance.

3.6.3 Achieved results and comparison

Table 3.1 and Table 3.2 show a comparison of the implementations achieved by CAOS against the ones from [25] and [26] that target a Xilinx VC707 board and an Amazon EC2 F1 instance respectively. The performance measure is given in terms of million of pairwise forces (MPairs) computed per second, which is a standard performance metric for the N-Body simulation problem and allows to compare the performance of implementations running on datasets with different number of bodies. We consider two versions of the application described in Section 3.6.2 providing support for a maximum of $N = 160$ and $N = 320$ bodies. It is important to mention that despite larger values of N were supported by CAOS, the final implementation resulted in poor performance. The reason is that pipelining the outer loop becomes unfeasible and the platform resorts to pipeline the innermost loop only, leaving most of the FPGA resources unutilized.

As we can see from Table 3.1, the implementation provided by CAOS with $N = 320$ bodies on a Amazon F1 instance outperforms, both in terms of pure performance and energy efficiency, the software implementation running in parallel on 40 threads on an Intel Xeon E5-2680 v2. Furthermore, the CAOS implementations achieve performance comparable to the implementation in [25] targeting a Xilinx VC707 board. Nevertheless, if we consider the latest implementation from [26], which is also targeting an Amazon F1 instance, it is clear that the bespoke implementation outperform the CAOS one by a factor of about 5x. The performance gap is due to additional optimizations which involve loop interchange, loop splitting, partial accumulations as well as pipelining. Overall, this allowed a more efficient use of the FPGA resources.

Despite the generated implementation is not optimal, it is worth noting that the version of the code developed with the CAOS platform has not been manually modified, whereas the versions presented in [26] and [25] were developed by expert users. Additionally, the CAOS implementation has been produced in less than a day, including the time to build the application for the hardware accelerator.

As a final remark, we want to stress that the results shown in Table 3.1 are meant to compare the CAOS solutions using the Master/Slave architec-

Listing 3.5: *Optimized Vivado HLS implementation of the* `forceComputation` *function from the N-Body simulation described in Listing 3.4. The Master/Slave interfaces are defined through HLS pragma, local arrays have been added to cache the data from the external DDR memory, the outermost loop* `force_loop_1` *has been pipelined and the innermost loop* `force_loop_2` *completely unrolled.*

```
void forceComputation_kernel(const float host_mass[N], coord3d_t
    host_position[N], coord3d_t host_accel[N], float eps)
{
  #pragma HLS INTERFACE m_axi port=host_mass bundle=gmem
  #pragma HLS INTERFACE s_axilite port=host_mass bundle=control
  #pragma HLS INTERFACE m_axi port=host_position bundle=gmem
  #pragma HLS INTERFACE s_axilite port=host_position bundle=control
  #pragma HLS INTERFACE m_axi port=host_accel bundle=gmem
  #pragma HLS INTERFACE s_axilite port=host_accel bundle=control
  #pragma HLS INTERFACE s_axilite port=eps bundle=control
  #pragma HLS INTERFACE s_axilite port=return bundle=control

  float mass[N];
  coord3d_t position[N];
  coord3d_t accel[N];

  memcpy(mass, host_mass, sizeof(float) * N);
  memcpy(position, host_position, sizeof(coord3d_t) * N);
  memcpy(accel, host_accel, sizeof(coord3d_t) * N);

  force_loop_1: for (int i = 0; i < N; i++) {
    #pragma HLS PIPELINE II=1
    force_loop_2: for (int j = 0; j < N; j++) {
      #pragma HLS UNROLL
      float rx = position[j].x - position[i].x;
      float ry = position[j].y - position[i].y;
      float rz = position[j].z - position[i].z;
      float dd = rx * rx + ry * ry + rz * rz + EPS;
      float d = 1.0f / (dd * sqrtf(dd));
      float s = mass[j] * d;
      accel[i].x += rx * s;
      accel[i].y += ry * s;
      accel[i].z += rz * s;
    }
  }

  memcpy(host_accel, accel, sizeof(coord3d_t) * N);
}
```

tural template against other FPGA-based implementations within the state-of-the-art on similar target platforms. Nevertheless, the nature of the N-Body case study makes it particularly suited for GPU-based acceleration. Indeed, thanks to the high amount of floating-point operations and the high degree of parallelism, the *n-body* benchmark from NVIDIA [75] is capable of achieving performance in the order of 100,000 MPairs/s on the most recent devices.

3.7 Final remarks

Within this chapter we presented the Master/Slave architectural template and its integration into the CAOS platform. The template allows to target either Xilinx Zynq-7000 SoC and Amazon EC2 F1 cloud instances, while providing acceleration support for a relatively large subset of the C/C++ language. The integration of the Master/Slave architectural template in CAOS enables application developers with low experience in FPGA-based acceleration to quickly identify the functions of the application to map on the reconfigurable hardware, optimize them and generate the system runtime to test and deploy the final implementation.

We have shown the benefits of the approach on the N-Body simulation case study, demonstrating the ability of the CAOS platform to quickly produce a working FPGA-accelerated implementation providing reasonable performance thanks to semi-automatic code optimizations. Currently, bespoke solutions produced by expert designers are expected to beat the performance of the implementations produced by the template. Nevertheless, the Master/Slave architectural template provides an interesting trade-off in terms of design time and performance.

Currently, we are working on a new design space exploration that alleviates the necessity to rely on Vivado HLS reports and logs. The overall idea is to estimate the performance of HLS implementations by performing approximate scheduling and resource mapping directly within the LLVM compiler framework. Furthermore, we are also including additional optimizations such as partial loop unrolling and support for cyclic as well as block array partitioning. Preliminary results show a 10x improvement for the design space exploration time and, in addition, we managed to automatically identify an optimized N-Body implementation that achieves 12,072 MPair/s, which is close in performance to the bespoke solution presented in [26]. As next steps, we plan to integrate this new methodology within the CAOS Master/Slave architectural template. Furthermore, we also plan to support more sophisticated caching mechanism, such as the one described

Table 3.1: *Performance and energy efficiency of the all-pairs N-Body algorithm accelerated via CAOS and the results achieved by bespoke designs proposed in [25] and [26].*

Ref.	Platform	Type	max # bodies	Frequency [MHz]	Performance [MPairs/s]	Performance/Power [MPairs/s/W]
[26]	Intel Xeon E5-2680 v2	CPU	inf.	-	2642	22.98
[25]	Xilinx VC707	FPGA	32000	100	2327	116.36
[26]	Xilinx VU9P	FPGA	inf.	154	13441	672.06
CAOS	Xilinx VU9P	FPGA	160	200	1781	89.05
CAOS	Xilinx VU9P	FPGA	320	200	2725	136.25

Table 3.2: *Resources utilization of the all-pairs N-Body algorithm accelerated via CAOS and the results achieved by bespoke designs proposed in [25] and [26].*

Ref.	Platform	Type	max # bodies	Resource utilization			
				BRAM 18K	DSP	FF	LUT
[26]	Intel Xeon E5-2680 v2	CPU	inf.	-	-	-	-
[25]	Xilinx VC707	FPGA	32000	509 (24.7%)	678 (24.2%)	88817 (14.6%)	134010 (44.1%)
[26]	Xilinx VU9P	FPGA	inf.	2714 (84.0%)	3762 (55.1%)	541222 (27.3%)	411562 (46.3%)
CAOS	Xilinx VU9P	FPGA	160	110 (3.41%)	3360 (49.21%)	589801 (29.71%)	504303 (56.58%)
CAOS	Xilinx VU9P	FPGA	320	616 (19.07%)	4482 (65.64%)	906479 (45.66%)	721175 (80.91%)

in [57], that could be used as alternatives to full chip caching in case the available local memory does not suffice. This would provide an additional design space point in order to smoothly degrade the memory transfer performance in case full caching is not feasible.

Overall, despite its generality and applicability, the Master/Slave architectural template does not provide optimal implementations for functions that could be implemented with more specific architectures and communication models such as the ones described in Chapter 4 and Chapter 5.

CHAPTER 4

Dataflow architectural template

This chapter presents a new methodology for accelerating applications featuring dataflow-like computations via Field Programmable Gate Array (FPGA). In particular, in the first part of the chapter, we focus on the OXiGen toolchain [79] that allows to perform an automated translation of a specific set of C/C++ functions into dataflow kernels. The second part of the chapter delves into the integration of the OXiGen toolchain as an architectural template within the CAOS platform presented in Chapter 2. Finally, we validate the approach on applications from the image processing and finance domain.

4.1 Introduction

This thesis focuses on the definition and implementation of an open-research platform to help and guide application designers towards the acceleration of applications defined in high-level C/C++ codes by means of FPGA-based architectures. Within this chapter, we discuss on how it is possible to automatize the process for the acceleration of a specific set of computations that can be framed and restructured as pure dataflow computations. Indeed, compared to other computing models, the dataflow one has proven to be very effective when implemented on FPGA [20]. In this context, we trade off the set of computations that we support in order to achieve higher performance, compared with more general approaches such as the Master/Slave architectural template discussed in Chapter 3. The proposed solution for automating the acceleration process revolves around the OXiGen toolchain and its integration within the CAD as an Adaptive Open-platform Service (CAOS) presented in Chapter 2.

The remaining content of the chapter is organized as follows: in Section 4.2 we provide more details behind the dataflow computing model and show how FPGA can benefit from its applicability. In Section 4.3 we describe existing tools and approaches for simplifying the acceleration process of applications on FPGA with a focus on the approaches that explicitly consider the dataflow computing paradigm. Furthermore, we also motivate and discuss the methodology proposed in this chapter. Section 4.4 presents the OXiGen toolchain and its components, while Section 4.5 introduces the performance and resource estimation model that are key for exploring and comparing several design points for the implementation of the dataflow accelerator. Section 4.6 discusses the integration of the OXiGen toolchain within the CAOS platform highlighting the set of modules that have been modified and the benefits brought by the integration. Finally, Section 4.7 analyzes the performance and productivity gain achieved by accelerating applications by means of the dataflow architectural template resulting from the integration of OXiGen within CAOS, while Section 4.8 draws the conclusions.

4.2 Dataflow computing and target systems

The dataflow computing paradigm represents a completely different view on how to perform a computation compared to the classical Von Neumann control-flow computational model [29]. Within the classical control-flow computational model, an algorithm is described as a sequence of instruc-

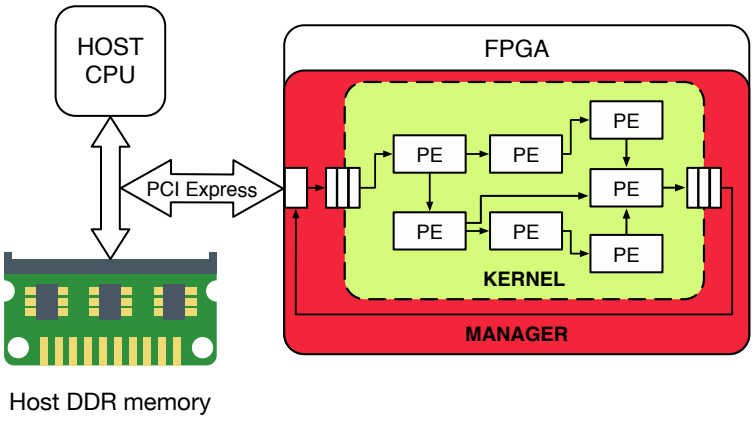


Figure 4.1: *Representation of a system targeted by the proposed dataflow architectural template. The system consists of a host CPU connected to an FPGA via PCIe. The FPGA resources are logically subdivided in a manger and a kernel portion. The manager handles the data transfer to and from the host Double Data Rate (DDR) memory via PCIe and the kernel, while the kernel performs the actual dataflow computation by means of a set of interconnected PEs.*

tions which are executed on a particular processor. Once the instructions are loaded into memory, the computation is performed by moving instructions from the memory to the processor, decoding the instructions to identify the type of operations to perform, reading the necessary data from memory, computing the required operation on the data and, finally, writing back the computed results to memory. Even though nowadays processors employ branch predictors, multiple caching levels and forwarding logic, the computing paradigm itself is inherently sequential and temporal in nature [72].

Within a dataflow computing model instead, the data is streamed from the memory directly to the chip containing an array of Processing Elements (PEs) each of which is responsible for performing a single operation of the algorithm. The data flow from a PE to the next one within a statically defined directed graph, without the need for any kind of control mechanism. In such a model, each PE performs its operation as soon as the input data is available and forwards the result to the next element in the network as soon as it is computed. This approach allows to layout the computation spatially [20], so that at regime, each PE performs useful work in parallel, as opposed to the control-flow paradigm in which operations are executed sequentially on the same shared computational unit. Furthermore, the overall dataflow graph can be deeply pipelined in order to achieve a throughput of one result per clock cycle.

Thanks to the optimized data movement, efficient memory accesses and the elimination of the control-flow logic, the dataflow paradigm allows to improve both performance and energy efficiency by several orders of magnitude for large-scale computations with a static compute graph [37]. Within this context, the FPGA has proved to be a very convenient yet effective technology for the implementation of dataflow computing system. In Figure 4.1 we show the dataflow system that we target in our context. The system consists in a host CPU and the dataflow accelerator deployed on a FPGA connected via PCIe to the host. Both the host CPU and the FPGA logic have access to the host DDR memory containing the input/output data. The FPGA accelerator is organized internally as a Globally Asynchronous Locally Synchronous (GALS) architecture divided into the actual accelerated kernel function and a manger. The manager handles the asynchronous communication between the host and the accelerator, whereas the kernel is internally organized as a set of synchronous PEs that perform the actual computation in parallel.

4.3 Related work and motivations

The most common way to develop designs for FPGAs is to use Hardware Description Languages (HDLs), like Verilog and VHDL. However, it may be definitely complex to master such low level languages. For this reason, in the last years, High-Level Synthesis (HLS) tools has gained more and more interest. Such tools allow the designer to develop an IP using high level languages instead of HDL.

In the state-of-the-art, there exist many and different HLS tools or frameworks that, starting from one or multiple Domain Specific Languages (DSLs), target HLS tools. Xilinx Vivado HLS [111] is a directives-driven architecture-aware HLS tool for Xilinx FPGAs. It supports C/C++ or System C languages, and provides multiple hardware optimized libraries and Application Programming Interfaces (APIs) to support the designer in the development of an IP. At the end of the HLS process, Vivado HLS provides the designer with an exhaustive report about the produced IP, in terms of circuit latency, resource usage, and so on. Eventually, the Vivado HLS exports the IP, which can then be integrated in a system design.

The LegUp [36] HLS framework, with a different approach, guides the developer through an incremental hardware acceleration of parts of an application. In this case, a MIPS soft processor mounted on the FPGA contains an *hardware profiler*, used to identify which parts of the program should be accelerated. The actual synthesis of the functions is performed

starting from an Intermediate Representation (IR) extracted from the initial C code through the LLVM compiler framework, and an iterative process applies different compiler passes and evaluates their effect on the resulting design, selecting only the combination of passes which reduce the number of clock cycles needed for execution. The framework aims at creating an iterative development model which allows a gradual transition from software to hardware implementation.

With respect to the DSL context, FROST [24] is a common backend for the acceleration of DSLs on FPGA. FROST provides an IR that can be targeted by DSLs, and a high-level scheduling co-language to describe the optimizations to enforce. FROST takes as input one or multiple functions described using one of the supported DSLs, translates the computation in its IR, and then applies optimizations according to the scheduling commands. The result of this process is a C/C++ implementation of the original computation suitable for Vivado HLS.

The aforementioned tools, as many other HLS tools, are designed to support different types of computational models, hence it is up the designer the development of an efficient implementation according to the computational model. On the other hand, other approaches focus on specific domains in order to better optimize the resulting design. In particular, since FPGAs fit well the dataflow computational model, there are different tools and languages tailored to this model. Darkroom [45] is a DSL and compiler for image processing pipelines. Darkroom compiles the input application into Verilog line-buffered pipelines, which are then synthesized for multiple architectures, namely ASIC, FPGA, or CPU. Another example of DSL for image processing is RIPL [99]. RIPL first compiles the input program to dataflow graphs, then exploits an open source dataflow compiler [11] to generates the RTL.

Maxeler Technologies implements the dataflow model through *MaxCompiler*, a compilation tool for the development of dataflow applications on FPGA-based platform. Each application is composed of the CPU host code comprehending the most control intensive code and the Dataflow Engine (DFE) code modeling the dataflow execution logic. On the other hand, each DFE is composed of several dataflow kernels performing simple but data-intensive computations and a manager responsible for handling the flow of data between kernels, memory and external interfaces. The CPU host code is written in a high level language, like C, whereas both kernels and manager are written in the *MaxJ* proprietary language.

Another work that targets Maxeler DFE is the FAST/LARA [42] compilation approach, which uses FAST, a programming language based on

C99 syntax, to provide a dataflow specification of the designs, and an *Aspect Oriented Programming* section specified in LARA to implement the necessary transformations to target the MaxCompiler architecture.

As described so far, there are many tools and languages to target FPGAs, and, even if we tighten the context to a specific type of computation, namely dataflow, the designers still have to adapt themselves to a specific technology in terms of both toolchain and language. Differently from the approaches available in the state-of-the-art, OXiGen enables designers to easily design dataflow applications for FPGA from one of the high level languages supported by LLVM. Moreover, OXiGen automatizes the synthesis process; hence, no knowledge of such process is required to the designer. Finally, OXiGen leverages MaxCompiler to implement the dataflow computation on Maxeler's DFEs.

By means of the *Dataflow architectural template* resulting from the integration of OXiGen within CAOS, we provide a comprehensive approach for accelerating dataflow-like applications on FPGA. More specifically, CAOS let the user identify which portions of the applications are amenable to be accelerated as dataflow kernels, allows the user to identify which, among such functions, are the ones that benefit the most from hardware acceleration and, finally, after having optimized the implementation of the candidate functions, it generates the required artifacts for running the accelerated application on the target system.

4.4 OXiGen infrastructure

4.4.1 Overview

The frontend, backend and overall design flow for OXiGen are shown in Figure 4.2. The tool takes as input a LLVM IR source file generated from a high level language frontend, the name of a target function defined within that file, and a set of parameters that determine the translation configuration. The IR of the target function is analyzed and processed in a series of stages in order to produce a dataflow representation of the computation. Once the dataflow graph for the function has been generated, the tool produces a conservative estimate of the hardware resources required to synthesize the circuit on FPGA, based on a simple model that allows to predict how a change in the configuration parameters would affect the estimated result. This information along with the resources constraints for the target board is sent to the configuration optimizer, which produces the optimal configuration for the design. The translation module produces the target

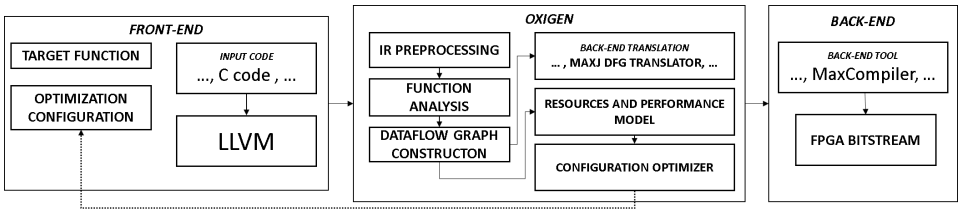


Figure 4.2: *The infrastructure surrounding OXiGen. The frontend contains the input, which consists of a* `.ll` *file of the source code generated by Clang, the name of the target function and a set of optimization options. The input is then processed by OXiGen, which outputs an optimal optimization configuration for the design and a kernel code which can be compiler with the selected backend synthesis tool.*

code by translating the intermediate representation into the target language of choice. When the translation is completed, the tool produces a file containing the code of the kernel, which can then be compiled by the beckend synthesis tool, in this case MaxCompiler, to generate the bitstream.

4.4.2 Frontend support

Presently, the variety of functions which can be translated by OXiGen have some restrictions. An exemplary structure for the input function is illustrated in the snippet of code shown in Listing 4.1. The function to be translated must have one or more outermost loops that iterate over an iteration variable with unitary stride and a number of iterations that is either known at compile time or directly dependent on a scalar input. Every input structure accessed in its outermost dimension through that iteration variable is considered a stream. The code can have any combination of further nesting levels and the corresponding iteration variables can be used to index local variables or other dimensions of the input and output streams. The restriction on the access patterns dictates that all the streams must be accessed linearly through an iteration variable with a constant offset. The further nested loops must also have a number of iterations known at compile time, and the local variables or the further dimensions of input and output streams can be accessed either through constant access or through a nested iteration variable with an optional offset, also known at compile time. It is worth nothing, that user-defined function calls are allowed and complex acyclic callgraphs are also supported (i.e. there should be no direct or indirect recursive calls). OXiGen automatically inlines sub-functions within the top function and treats it as a single software description of the dataflow kernel to implement. Furthermore, OXiGen also provides a library of pre-optimized functions that the user can call within the kernel. Such functions

are defined in a custom header file, which overrides the actual library function calls within the code. The library includes many basic mathematical functions, such as `min()`, `abs()`, `ceil()` and can be easily extended by providing the C-prototype as well as the description of a custom hardware module specified using the backend specific language (such as MaxJ for the MaxCompiler backend).

Despite the constraints on the code, it is worth noting the many applications can be expressed in a form similar to the one presented in Listing 4.1. From a high-level perspective, the computations supported by OXiGen are of the form $y_i = f(\psi(\bar{x}, i), i)$, where $\bar{x}$ is the input vector and $\bar{y}$ is the output vector. The function $\psi(\bar{x}, i)$ returns a set of inputs relative to index i. As an example, if we consider $\bar{x}$ and $\bar{y}$ as vectors containing the pixels of two images stored row-wise, a 1x3 filter f that applied to the input image $\bar{x}$ generates the output image $\bar{y}$ can be expressed as $y_i = f(\{x_{i-1}, x_i, x_{i+1}\}, i)$ in which we used $\psi(\bar{x}, i) = \{x_{i-1}, x_i, x_{i+1}\}$.

4.4.3 IR preprocessing

In the first phase of the translation process the IR is optimized through a series of LLVM analysis and transformation passes. The `-mem2reg` pass promotes memory references to register references and transforms the IR into pruned Static Single Assignment (SSA) form. Then, the `-loops` analysis pass is used to extract information on the natural loops of the function and their structure. Finally, the `-scalar-evolution` analysis pass provides information on the scalar expressions within loops. This allows to compute the number of iterations performed by a loop and to identify the iteration index. The preprocessing phase puts the IR into a form which can be more easily translated into a dataflow representation and extracts all the information required to perform the next translation step. At this stage of the flow, the *vectorization* optimization can be applied if requested. *Vectorization* changes the data type of the input and output streams of the target function into vector types. The user can select a vectorization factor to determine the size of the input and output vectors. A vectorized version of the function is created where an additional dimension is added to the existing data types and the structure is changed to allow iteration over the added vector dimension. This optimization is target specific, since in MaxCompiler this structural change allows to improve the parallelism of the computation, as the hardware resources for each element of the vectorized dimension are replicated and the elements are processed in parallel. This also allows to fully utilize the available bandwidth for data transfer.

Listing 4.1: *An exemplary code for OXiGen. The function takes as input a combination of array types and scalar types. The outer loops iterate over the outer dimension of the array types which will be translated as streams. The accesses to the streams are linear with constant offsets. The function can have a combination of nesting levels iterating over the inputs or local variables*

```
#define INPUT_SIZE 100

void foo( data_type_1* in_1,  data_type_2* in_2, data_type_1* out_1,
    scalar_type_1 v_1 ) {

  data_type_1 tmp_vect[15];

  L1: for( int i = const_offs_1; i < INPUT_SIZE - const_offs_2; i++ ) {

    S1: ...statements...

    L1.1: for( int j = const_offs_3; j < 15; j++ ) {

      S2: tmp_vect[j] = ...expression...
    }

    S3: data_type_1 tmp_scalar = ...expression...;

    L1.2: for( int j = const_offs_3; j < 15; j++ ) {

      S4: tmp_scalar = tmp_scalar + tmp_vect[j];
    }

  }

  L2: for( int i = const_offs_4; i < INPUT_SIZE - const_offs_5; i++ ) {

    S5: ...statements...
  }
}
```

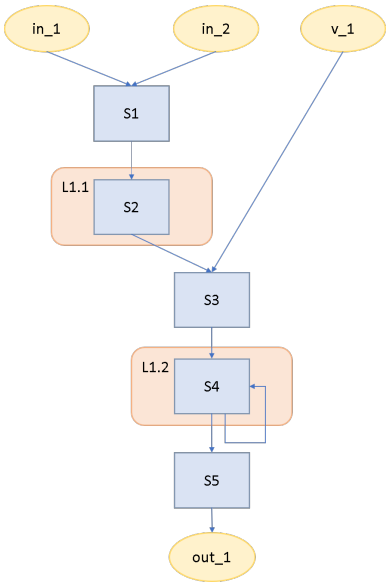


Figure 4.3: *Dataflow graph generated by OXiGen on the exemplary code.*

4.4.4 Function analysis

In the second phase of the translation process a custom pass is used to generate a dataflow graph for the target function. This representation of the computation is not bound to a specific language and is designed to generate translations for different backends. This graph representation is constructed by parsing the LLVM IR and makes use of several LLVM analysis and APIs, but is distinct from the LLVM IR of the function since it is designed specifically to represent dataflow computations. The pass is organized into several components which perform specific functionalities and are instantiated as needed, depending on the characteristics of the target function. The first two, the *AnalysisManager* and the *StreamsAnalyzer*, are always scheduled and perform a series of analysis which inform the subsequent components. In particular, the *AnalysisManager* examines the structure of the function to identify the outermost loops, which become implicit in the dataflow graph representation, and classifies the memory accesses. This is done to enforce restrictions on the patterns which can be used to access memory data structures. In particular, we have restricted the possible memory access patterns to linear accesses with constant offsets. Once these constraints have been verified, the *StreamsAnalyzer* identifies the data structures which will be translated as streams and the inputs and outputs for the function. This information is used to initialize the construction of the

dataflow graph.

4.4.5 Dataflow graph construction

The graph is constructed iteratively by alternating phases in which instructions are parsed to generate new nodes of the graph and phases in which the generated nodes are linked to one another by analyzing the data dependencies within the function. These two processes are performed by the *DFGConstructor* and *DFGLinker* components respectively. During the graph construction, the different offsets used in the indexes to access data structures are translated into offset nodes in the graph, in order to maintain the function semantics. The *DFGSubloopHandler* takes care of analyzing and translating the nesting structure of the function, which is represented as different nesting levels of the graph. A *DFGLoopNode* is a graph node which represents a loop and contains a dataflow graph representing the computation performed by the body of the loop, as well as additional information such as the trip count for the loop and the loop iteration variable. These nodes are translated in a target-specific fashion in the next translation phase.

4.4.6 Dataflow graph translation

The fourth and final translation phase takes as input the dataflow graph of the target function and the optimization options selected by the user to generate the final code for the kernel. This final phase is target specific, since the nodes of the graph need to be parsed and translated into instructions for the target language. At present, the only supported target language is MaxJ, but we are looking to expand this pool in the future. After the dataflow graph has been fully constructed, the loop rerolling optimization is applied if requested. This optimization is also target specific and allows the user to control the amount of hardware replication for the computation within nested loops, provided that said computation does not have data dependencies across loop iterations. The loops are translated in MaxJ as java-like loops which act as macros that describe hardware replication. Each MaxJ loop body iteration is implemented using dedicated dataflow nodes and the dependencies among iterations are resolved statically. A previously conducted analysis in the dataflow graph provides a representation of the nested loops within the function and their data dependencies and labels each loop depending on the kind of computation it performs. The loops which write in a local data structure at each iteration are labeled as *ExpansionLoops* and produce several outputs in parallel, proportionally to the number of iterations they perform. Conversely, those loops which

read more than one element in parallel and write an aggregated result are labeled as *AccumulationLoops*. Presently, if a loop performs both an expansion and an accumulation is not considered eligible for the optimization. A possible future development for this analysis would be the ability to automatically separate the expansion and accumulation computations in two distinct loops to enable further optimization. If the rerolling optimization is selected, the user chooses a rerolling factor which determines how many elements are produced in parallel by the expansion loops. The rerolling factor effectively reduces the number of iterations for a loop (i.e. hardware replication) and the elements of the computation which were previously produced fully in parallel are distributed across several cycles. The accumulation loops use different stream offsets to access the elements needed for the aggregate computation they need to perform, and appropriate delays are introduced to maintain data dependencies. This optimization reduces the speed at which the input is consumed by the kernel, effectively slowing down the computation, but is useful for those computations which cannot by fully parallelized due to limited FPGA resources. Selecting an appropriate rerolling factor can reduce the required hardware resources by sharing them across time. The actual translation phase is relatively straightforward, as most of the computation transforms the information already contained in the graph into statements. The resulting code is in a quasi-SSA form and the instructions relate to the graph nodes in an almost one-to-one mapping. In this phase the function calls within the target functions are resolved by either mapping them to already existing library functions for the target language or including in the kernel a custom implementation for the function. The necessary information to guide this process are provided by including in the original source code a custom header file with all the functions supported by the tool, which overrides the original implementations for those functions. Once the code of the kernel has been generated, it is produced as output in the form of a MaxJ file.

4.5 Resources and performance model

In order to guide the optimization of the function, OXiGen generates a performance and resource model specifically tailored for each of the supported optimizations, such as rerolling and vectorization.

Each model considers two sets of variables that can be tuned in order to modify the expected final performance and resource consumption of the implementation. The first is a set of variables f_i for $i \in I$ related to the optimization to perform, while the second set consists of a single variable

$\theta \in \mathbb{N}$ that specifies which configuration parameters to use for the backend tool when implementing the final system. Currently, in our models, the choice of the value of θ is expected to impact mainly on resource consumption and, in particular, on the resource mix used for implementing the different nodes within the dataflow system. In our experimental setting described in Section 4.7, we used θ to select among different values of the *DSP push* parameter provided by MaxCompiler that allows to balance the usage of Digital Signal Processings (DSPs), Look-Up Tables (LUTs) and Flip Flops (FFs) by using a different technology mapping.

Overall, each model provides a function $p(f_0, \cdots, f_i)$ that expresses the throughput of the system (bits / second) and the functions $q_n(f_0, \cdots, f_i, \theta)$ that specify the number of instantiations of dataflow node $n \in N$ in the final system (e.g. number of 32-bits floating-point multipliers, 8-bits adders, ...).

The overall estimation r_t of the resource consumption of resource type $t \in T$ (e.g.: $T = \{DSP, BRAM, FF, LUT\}$) is given by the following simple model:

$$r_t = \sum_{n \in N} c_{n,t,\theta} \cdot q(f_0, \cdots, f_i) \tag{4.1}$$

where, $c_{n,t,\theta}$ is the number of resources of type t used by node n under the configuration θ. The characterization of the compute nodes given by $c_{n,t,\theta}$ is performed only once by implementing each node separately as a single kernel function and retrieving the final resource utilization reported by the backend tool after place-and-route at a given target frequency on the target FPGA, across the possible configurations θ. It is worth noting that even if this is a time consuming task, once the characterization is performed, the achieved results are independent from the application and can be reused. Furthermore, the range of possible configurations given by θ is usually restricted to a small range (e.g. 10 different configurations).

It is worth mentioning that the resource model only takes into account the resources occupied by the kernel and does not consider the resources needed by the communication subsystem. Nevertheless, as shown in Section 4.7, the resource consumption of the most constrained resource not related to the kernel function is well below 10%. During the design space exploration OXiGen takes into account a 15% slack of the total available resources in order to avoid to overconstrain the design and leaves enough space for the communication subsystem.

In the following sections, we provide the expression of the functions p and q_n of the performance and resource estimation model for the rerolling and vectorization optimizations.

4.5.1 Rerolling model

The rerolling optimization only requires one variable $f_0 \in \mathbb{N}_0$ that specifies the global rerolling factor for the supported nested loops (i.e.: loops without carried dependencies). Since the amount of rerolling is inversely proportional to the peak throughput, the performance $p(f_0)$ can be estimated as:

$$p(f_0) = min\left\{\frac{R_{out}}{f_0},\ B_{out},\ \frac{R_{out}}{R_{in}} \cdot B_{in}\right\} \tag{4.2}$$

Where R_{out} and R_{in} are the overall output produce rate and input consume rate, while B_{out} and B_{in} are the maximum output and input bandwidth respectively. On the other hand, the number of occurrences of the nodes within the final implementation can be computed as:

$$q_n(f_0) = k_0 + \sum_{l \in L} k_{l,n} \cdot \left\lceil \frac{i_l}{f_0} \right\rceil \tag{4.3}$$

Where L is the set of nested loops that supports rerolling, i_l is the original number of iterations of nested loop l, k_0 is the number of occurrences of node n outside the nested loops in L, while k_l is the number of occurrences of node n within nested loop l.

4.5.2 Vectorization model

As for rerolling, the vectorization optimization requires a single variable $f_0 \in \mathbb{N}_0$ that specifies the vectorization factor. Since vectorization increases the overall output produce rate (R_{out}) and input produce rate (R_{in}) of the kernel function, we can estimate the performance as:

$$p(f_0) = min\left\{f_0 \cdot R_{out},\ B_{out},\ \frac{R_{out}}{R_{in}} \cdot B_{in}\right\} \tag{4.4}$$

On the other side, the number of instances of a node in the resulting implementation is simply multiplied by the vectorization factor since the optimization replicates the hardware in order to support the increased input and output consume and produce rates:

$$q_n(f_0) = f_0 \cdot k_0 \tag{4.5}$$

Where k_0 is the overall number of occurrences of node $n \in N$ in the original not optimized implementation.

4.5.3 Design space exploration

Overall, in order to get the optimal performance for a given optimization, we have the following objective:

$$argmax_{f_0,\cdots,f_i,\theta}\ p(f_0,\cdots,f_i) \tag{4.6}$$

under the constraint on the available FPGA resources:

$$r_t \leq M_t\ \ \forall t \in T \tag{4.7}$$

Where M_t is the amount of resource of type t available on the FPGA.

By substituting the equations derived for either the vectorization or rerolling model into equations 4.6 and 4.7 and by applying standard linearization techniques, we can derive a Mixed-Integer Linear Programming (MILP) model that can be solved in order to find the optimal parameters for the given optimization.

The design space exploration starts from the initial dataflow graph description and examines all the available optimizations. For each optimization, the optimal parameters $f_0,\cdots,f_i,\theta$ are identified by solving the corresponding MILP model and the feasible optimization that provides the best performance is selected. Finally, the exploration can be repeated to check if the application might benefits from additional optimizations. It is worth noting that the MILP solver might not be able to find any feasible set of parameters due to the hardware resource constraints. In this case, the corresponding optimization is simply excluded from the available alternatives.

As a final remark, notice that rerolling and vectorization are opposing optimizations that either improve hardware utilization against performance or increase performance at the cost of higher resource utilization respectively. Hence, the design space exploration would apply at most one among rerolling and vectorization.

4.6 OXiGen integration in CAOS

The support for the OXiGen toolchain within CAOS is implemented as an architectural template named as *dataflow architectural template*. This allows CAOS to check, within the *architectural template applicability check* module, if the application is suitable or not for being implemented with OXiGen using the dataflow computational model. This approach also allows the designer to simultaneously verify which, among the different architectural templates, is available for accelerating the application. Finally, the

designer can start multiple design flows testing with different architectural templates to achieve best performance.

Similarly to the integration of the Master/Slave architectural template in CAOS discussed in Section 3.5, we decided to modify the default CAOS modules instead of creating new modules accessible at different IP addresses. The next sections provides more details on the changes performed at the different steps of the CAOS design flow.

4.6.1 CAOS frontend

With regards to the CAOS frontend, we only needed to modify the *architectural template applicability check* module. Adding new architectural template to such module is a relatively easy task, since the module is internally organized in a modular fashion and allows to extend the check for additional templates simply by extending a common interface.

First of all, in order to ensure that the dataflow template is supported, we verify if the architecture description JSON file matches one of the board supported by he Maxeler toolchain, which is currently leveraged for the generation of the final system. As a second step, we scan all the functions of the application and verify whether there is any function which complies to the code structure described in Section 4.4.2. This is done by running the OXiGen toolchain and early stop the execution at the function analysis stage to make sure that the source code is suitable for being converted to a dataflow kernel.

4.6.2 CAOS functions optimization

The CAOS functions optimization flow is internally organized into 4 different modules: static code analysis, hardware resource estimation, performance estimation and code optimization. In order to integrate OXiGen within the functions optimization flow, we have developed specific extensions leveraging on the common interfaces offered by each default CAOS module. Notice that OXiGen does not directly modify the original source code of the software implementations, but instead tailors the overall translation process to the MaxJ dataflow implementation using a set of configuration parameters. In order to keep track of the specific optimizations that OXiGen should apply to the code during the translation process, we add an optimization configuration file together with the source code of the software implementation. Such configuration file can than be parsed by each of the modules within the functions optimization flow as needed.

Static code analysis

The static code analysis module is responsible for extracting specific information from the function code that can then be leveraged by the performance estimation module in order to explore different optimization opportunities and estimate their quality. With respect to the dataflow architectural template, the module leverages OXiGen in order to extract the input and output streams of the function, the bitwidth of such streams, the rate at which input and output data are read and written under the assumption of unlimited bandwidth, the instruction count for each of the node and loop node in the OXiGen dataflow graph and the trip count for each loop node.

Resource estimation

The resource estimation is performed by running OXiGen until the resource model stage. The estimation of the resources takes into account the current set of optimization applied to the code defined within the optimization configuration files as well as the target architecture. The estimation process is performed as described in Section 2.3.2.

Performance estimation

This module is responsible for performing two different tasks. The first task is to collect the information derived from the static code analysis and resource estimation modules to produce a comprehensive report of the hardware resource utilization and an estimation of the final kernel throughput that takes into account the available bandwidth for input and output data transfer. The second task instead, is to provide one or more optimization opportunities to apply during the OXiGen translation process. The potential optimizations are generated by means of the design space exploration discussed in Section 2.3.2.3 using the data derived from the static code analysis for the construction of the performance and resource models.

Code optimization

Once the user selects one among the optimizations produced by the performance estimation module, the code optimization module is responsible for applying the desired changes. Since OXiGen does not directly operates on the original source code of the application, the code optimization is simply performed by annotating the requested optimization within the optimizations configuration file that is taken into account by all the other modules when dealing with the dataflow architectural template.

4.6.3 CAOS backend

After having performed the function optimization flow, the CAOS backend runs OXiGen until completion by taking into account all the selected optimizations. As a result, OXiGen generates the MaxJ files describing both the dataflow kernel and the manager subsystem. Then, the *runtime generation* phase has been extended to modify the original software source code and integrate the Maxeler headers and APIs calls to run the accelerator. Overall, the end result of the CAOS backend flow targeting the dataflow architectural template consists in a Maxeler project ready to be build and compiled on the target system.

4.7 Experimental evaluation

In order to validate the proposed approach, we have tested the dataflow architectural template powered by OXiGen on two case studies belonging to two different application domains. The first application is a financial algorithm used to calculate the pricing of Asian options, which has already been accelerated on FPGA with a dataflow approach in [72], and has yielded remarkable results. The second application consists of a series of filters used in the context of image processing to sharpen images, increasing the contrast between bright and dark regions to bring out features [80]. During the experimental campaign, we considered a C implementation of the algorithm that operates on full High-Definition images (1920x1080 pixels) with 3 8-bit color channels. These two applications have different critical resources and are very appropriate to test the effects of the different configuration options on the resulting design.

The testing system consists of an host machine featuring an Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz connected via PCIe gen1 x8 to a MAX4 Galava board equipped with an Altera Stratix V FPGA. The CPU baselines were compiled with gcc 4.4.7 using -O3 level optimizations, while the target frequency for the dataflow designs is 200 MHz. The characterization of the resource consumption of the base dataflow nodes was performed considering 11 different values for the DSP push parameter (0.0, 0.1, $\cdots$, 0.9, 1.0). Finally, Gurobi Optimizer 7.5 [44] has been used to identify the optimization parameters during the design space exploration. It is worth noting that in both test cases the design space exploration time, including the time for solving the MILP models, was well below a second, which is negligible compared to the overall system synthesis time.

4.7.1 Asian Option Pricing

The Asian Option Pricing algorithm accelerated on Maxeler's DFE in [72] is the Curran's approximation algorithm [74]. This algorithm is definitely computationally expensive, and provides different opportunities for parallelization. Figure 4.4 shows the overall structure of the Asian Option Pricing algorithm. Such application is mainly composed by five asynchronous kernels, reported in Figure 4.4 as K1, K2, K3, K4, and K5. Each kernel performs part of the Curran's algorithm, communicates with the other kernels by means of FIFOs, and leverages fixed-point data types to reduce resource usage, while satisfying the accuracy constraint typical of financial applications. Finally, kernel K4 exploits the normal cumulative distribution function (NCDF) to easily compute the Asian put and call options. Maxeler's library provides `functionHART` to efficiently compute an accurate approximation of the NCDF. `functionHART` is implemented with a fixed-point piece-wise polynomial approximation generated at hardware compile time using the Remez algorithm [65, 72].

Starting from the C version of the Curran's approximation algorithm with 30 averaging points, OXiGen generates an efficient implementation for the Maxeler's DFEs. Just like in the implementation presented in [72], OXiGen organizes the computation in five kernels. In particular, OXiGen can act on K2 and K4 kernels, which are dependence free loops, and apply reroll in order to reduce the usage of hardware resources, and, consequently, the performance. On the other hand, OXiGen currently cannot act on K3 and K5 kernels, since they contain carried dependencies. Finally, OXiGen makes `functionHART` available within its library, which allows users to call it from the C side.

Without the reroll optimization, the application would not fit into the target FPGA. OXiGen detects this issue and finds an implementation that fits within the FPGA and maximizes the performance. The final solution generated by OXiGen uses a rerolling factor of 5 and a DSP push factor of 0.1 achieving a 88.1x speedup over the single-threaded CPU execution. The identified solution together with other implementations using different combinations of the rerolling factor and DSP push factor are shown in Table 4.1 and Table 4.2. The Table reports the speedup with respect to the single-threaded execution on the host machine, the input output bandwidth achieved by the system, the resource utilization as well as the resource estimation error for the kernel function and the total resource consumption including the communication subsystem generated by MaxCompiler. From the table it is possible to see that different values of the DSP Push factor

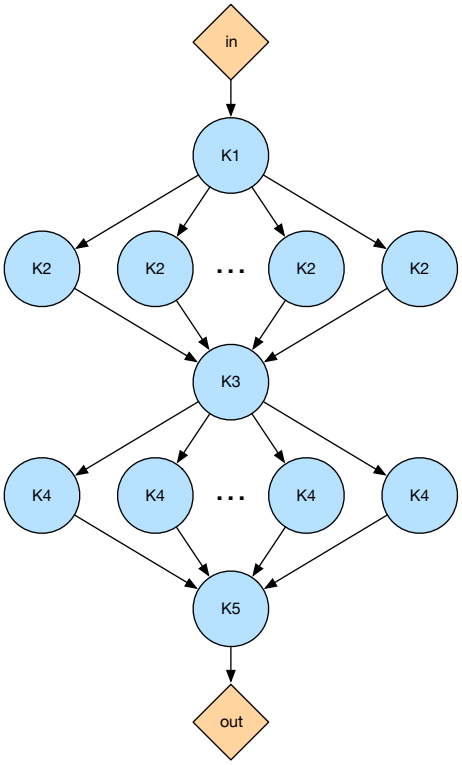


Figure 4.4: *The overall structure of the Asian Option Pricing application, as implemented in [72].*

generates implementations with a different balance of DSPs, LUTs and FF. It is worth noting that an implementation with a default DSP Push factor of 1.0 would have exceeded the number of available DSPs for a rerolling factor of 5, nevertheless OXiGen was able to identify a feasible solution using a DSP push factor of 0.1.

The absolute error of the resource estimation model for the estimation of DSP, FF and LUT is below 5% of the total amount of available FPGA resources, while the Block RAM (BRAM) resources are generally underestimated (at most -9.23% for the largest design). This is due to the fact that the estimation model currently does not consider the impact of the FIFOs, which we plan to support in future works. Despite the estimation error, the model is able to successfully perform the design space exploration since both the reroll and vectorization are coarse grain optimizations that do not require high accuracy.

Finally, compared to the implementation obtained in [74] we needed a

Table 4.1: *Achieved bandwidth and speedup of several implementations of the Asian Option Pricing application.*

DSP Push	Rerol. Factor	Speedup	Bandwidth [MBit/s] Input	Output
1.0	30	15.83	1,892	210
1.0	15	31.22	3,729	414
0.1	10	45.95	5,489	610
1.0	8	56.91	6,799	755
0.1	6	74.35	8,883	987
0.1	5	88.10	10,525	1,169

Table 4.2: *Resource consumption and early resource estimation of several implementations of the Asian Option Pricing application.*

DSP Push	Rerol. Factor	Kernel Resources (estimation error) [%] DSP	BRAM	FF	LUT	Total Resources [%] DSP	BRAM	FF	LUT
1.0	30	39.06 (-0.39)	25.68 (-1.27)	18.48 (-0.93)	31.33 (-3.31)	39.06	37.65	21.86	37.15
1.0	15	55.08 (-1.17)	33.64 (-5.03)	20.77 (-0.51)	34.89 (-3.47)	55.08	42.09	24.47	40.71
0.1	10	29.30 (+0.39)	34.52 (-1.71)	27.92 (-0.63)	47.06 (-3.77)	29.30	41.50	31.55	52.32
1.0	8	87.11 (-2.73)	38.87 (-1.86)	26.98 (-1.28)	41.91 (-3.70)	87.11	54.30	29.60	47.73
0.1	6	41.02 (+0.39)	50.29 (-9.08)	35.81 (-1.03)	58.51 (-4.38)	41.02	59.13	39.51	64.31
0.1	5	46.88 (+0.39)	54.64 (-9.23)	39.59 (-1.07)	64.28 (-4.74)	46.88	63.96	43.31	70.09

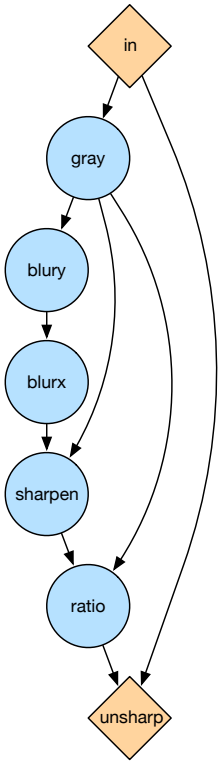


Figure 4.5: *The sequence of filters applied by the sharpen filter on the input image. The figure shows the filters and their data dependencies.*

higher rerolling factor (5 instead of 2) at a similar target frequency. Despite a completely fair comparison of the two implementations is not possible since the authors in [74] targeted a different system, i.e. a Maxeler MAX4 MAIA DFE connected to an Intel Xeon CPU E5-2697 v2 CPU, it is still interesting to compare the execution time of the two solutions on the same dataset. In particular, for a problem consisting of 10000 Asian Options over 5000 market scenarios and using 30 averaging points, the implementation from [74] required 1.25 seconds, while our implementations completed in 1.78 seconds. Hence, the bespoke implementation achieved a 1.4x speedup over the solution generated with OXiGen. Such performance different is rather small if we compare it to the performance gain achieved against a CPU-based implementation. Moreover, it is worth mentioning that the implementation of the full system, starting from the original floating-point version of the code written in C, was performed in about half a day by a non expert user through OXiGen.

One of the main reasons for the performance gap between the OXiGen solution and the one proposed in [74], is the usage of fixed-point data types which allow to reduce the impact on resource consumption and hence to perform a less aggressive rerolling optimization.

4.7.2 Sharpen Filter

A schematic representation of the filters applied within the sharpening algorithm and the corresponding kernels is shown in Figure 4.5. The C implementation of the algorithm does not harness the potential for parallel computation of the FPGA, and can be vectorized until the maximum bandwidth for the target board is reached. Indeed, implementing the application without any optimization leads to an overall resource consumption of 10% of the most constrained resource and a bandwidth utilization of about 26%. In this case, OXiGen identified a solution using a vectorization factor of 8 which achieves a bandwidth utilization close to the maximum PCIe gen1 x8 peak bandwidth. Table 4.3 and Table 4.4 show different implementations of the sharpening algorithm using different values of the vectorization factor. As can be noted, using a vectorization factor higher than 8 does not bring any benefit due to the data transfer bottleneck and unnecessarily increase resource utilization, while smaller vectorization factor produce sub-optimal implementations. Overall, we achieved a speedup of 15.85x compared to the CPU-based single-threaded implementation simply by leveraging the dataflow architectural template within CAOS and implementing the final system via MaxCompiler.

4.8 Final remarks

Within this chapter we presented the CAOS dataflow architectural template powered by the OXiGen toolchain, that allow to translate high-level code functions into optimized FPGA-based dataflow kernels. We evaluated the capabilities of the approach on two applications from the finance and image processing domains and achieved a speedup of 88.1x and 15.85x over the initial single-threaded software implementations respectively. The implementations were obtained in less than a day by non expert users and OXiGen automatically identified the optimization opportunities by leveraging the proposed resources and performance estimation model.

Future works will investigate the possibility of enhancing the accuracy of the resource estimation model in order to enable finer grain optimizations and to extend the set of available optimizations. Moreover, even though we consider a fixed clock frequency for the implementation of the designs, an

Table 4.3: *Achieved bandwidth and speedup of several implementations of the Sharpen Filter.*

DSP Push	Vector. Factor	Speedup	Bandwidth [MBit/s] Input	Output
1.0	1	4.58	4,403	4,403
1.0	2	8.86	8,513	8,513
1.0	4	15.37	14,772	14,772
1.0	8	15.85	15,238	15,238
1.0	16	15.62	15,017	15,017

Table 4.4: *Resource consumption and early resource estimation of several implementations of the Sharpen Filter.*

DSP Push	Vector. Factor	Kernel Resources (estimation error) [%] DSP	BRAM	FF	LUT	Total Resources [%] DSP	BRAM	FF	LUT
1.0	1	5.86 (0)	4.05 (-0.59)	2.37 (-0.13)	3.63 (-0.18)	5.86	9.81	4.85	7.28
1.0	2	11.72 (0)	8.20 (-1.27)	4.60 (-0.11)	7.03 (-0.14)	11.72	13.57	7.07	10.75
1.0	4	23.44 (0)	15.77 (-1.90)	9.05 (-0.07)	13.9 (-0.12)	23.44	21.14	11.52	17.55
1.0	8	46.88 (0)	31.05 (-3.32)	17.97 (-0.02)	27.52 (+0.04)	46.88	36.72	20.43	31.17
1.0	16	93.75 (0)	61.23 (-5.76)	35.78 (+0.13)	54.96 (+0.16)	93.75	67.29	38.16	58.51

additional design space exploration step could be considered to identify the maximum achievable clock frequency in order to improve the performance of compute bound designs. Notice also that despite the Maxeler toolchain is used for implementing the final system, it is not a requirement of the proposed approach. Indeed, future work might consider generating code for Vivado HLS or targeting directly HDL generation.

CHAPTER 5

Streaming architectural template

Stencil computations represent a highly recurrent class of algorithms in various high performance computing application scenarios, such as differential equation solving or numerical and scientific simulations. The Streaming Stencil Time-step (SST) architecture represents a very promising state-of-the-art Field Programmable Gate Array (FPGA) implementation of stencil computations, which exploits the highly regular and repetitive structure of such algorithms to achieve a high performance/power consumption trade-off. In this chapter, we present a design space exploration methodology for SST-based architectures which leverages on floorplanning in order to identify the maximum number of modules that can be instantiated for a given device. Furthermore, we describe the integration of the SST architecture and the proposed design exploration as an architectural template in CAOS. The resulting streaming architectural template allows programmer to efficiently accelerate stencil code written in C on FPGA.

5.1 Introduction

Iterative Stencil Loops (ISLs) represent a class of algorithm that are highly recurrent in many High Performance Computing (HPC) applications such as differential equation solving (e.g. for weather or ocean modeling [59]), scientific simulations (e.g. quantum dynamic [67] or seismic ones [51]). The fundamental structure of such algorithms consists of a main loop iteratively processing an n-dimensional input vector many times; at each iteration each value in the vector is updated by applying a *stencil*, being a weighted sum of the elements in the neighbor positions.

The implementation of ISLs has been widely investigated in the literature, and their optimization has been approached in a variety of ways and targeting various architectures such as multi-core CPU [6, 56], Graphics Processing Units (GPUs) [47] and FPGA devices [66, 70, 91]. Indeed, the ISL regular structure makes them an excellent candidate for automatic compile-time analysis and optimization. For instance, such regularity allows ISLs to be modeled via a powerful mathematical framework, called *polyhedral model* [33]. This model has been employed to either optimize ISL implementation on the target architecture, such as in [6], or design custom HW architecture tailored to this class of algorithms, as proposed in [70].

Among the available solutions, FPGA-based implementations (such as [66, 70, 91]) currently represent a very promising candidate for ISL acceleration, as they offer a compelling trade-off between performance and power consumption thanks to direct hardware acceleration, while retaining flexibility achieved by means of their reconfigurability. One of the most interesting approaches in this scenario has been presented in [70]. The work proposes a design methodology for the automated generation of HW accelerators targeting FPGA devices starting from the C algorithmic description and exploiting commercial High-Level Synthesis (HLS) tools. The obtained design consists of a highly repetitive and modular pipeline of a basic module, called SST, that results in an efficient resource usage and scalability. However, such methodology lacks a backend devoted to the implementation and optimization of the pipeline design onto the FPGA resource grid. In fact, since the SST-based architecture usually contains up to 100 modules, the absence of specific actions in the implementation flow leads to suboptimal performance and long design times.

Given these motivations, in this chapter we propose a fully automated design flow to support the designer in the acceleration of stencil codes as streaming accelerators powered by an SST-based architecture on FPGA.

Additionally, we integrate such design flow within CAD as an Adaptive Open-platform Service (CAOS) as the *streaming architectural template*, so that the designer can also benefit from the modules of the CAOS frontend to profile the application and verify which architectural templates can be used for hardware acceleration. The proposed approach, starts by generating an initial version of the system consisting of a single SST by following the methodology in [70], then, it maximizes the throughput of the initial design by 1) preliminarily estimating the number of SST modules that can be instantiated on the target FPGA device, and 2) floorplanning the design by means of a custom strategy to maximize the achievable clock frequency. As experimentally demonstrated, the main advantages of the proposed methodology are 1) a drastic reduction of the design time, since the approach is not based on iterative improvements, and 2) the possibility to increase the performance of the final design thanks to the exploitation of its characteristic regularity.

The rest of the chapter is organized as follows. Section 5.2 briefly discusses the background on the SST architecture and highlights the motivations of this work. Then, Section 5.3 presents the proposed design flow, while its integration in CAOS is discussed in Section 5.4. Finally, the proposed design flow is evaluated in Section 5.5, while Section 5.6 draws the conclusions and presents the future work.

5.2 Background and motivations

This section briefly describes the SST architecture and, later, analyzes its performance characteristics in order to identify the possible knobs to be acted in the proposed design flow.

SST architecture The basic structure of ISLs is depicted in Algorithm 2; the outer loop iterates for a given number of times, so called *time-steps*, while, at each time-step, the inner loop updates each value of the n-dimensional input vector by means of the *stencil* function, computing a weighted sum of the neighbor values in the vector.

Algorithm 2 Generic ISL Algorithm.

```
for t ≤ TimeSteps do
    for all points p in matrix M do
        p ← stencil(p)
```

The architectural template of the SST-based accelerator [70] is depicted in Figure 5.1. The basic SST module performs the computation of a single

time-step and is conceptually separated in a *memory system*, responsible for storage and data movement, and a *computing system*, that performs the actual computation. The SST module is designed in order to operate in a streaming mode on the various elements of the input vector; this internal structure is derived by means of the *polyhedral analysis* that allows to refactor the algorithm to optimize on-chip memory resource consumption and implement a dataflow model of computation. Then, the complete SST-based architecture is obtained by replicating N times the basic module to implement a pipeline, where each module computes in streaming a single time-step of the outer loop of the algorithm. Such a pipeline is finally connected with a fixed communication subsystem interfacing with the host machine.

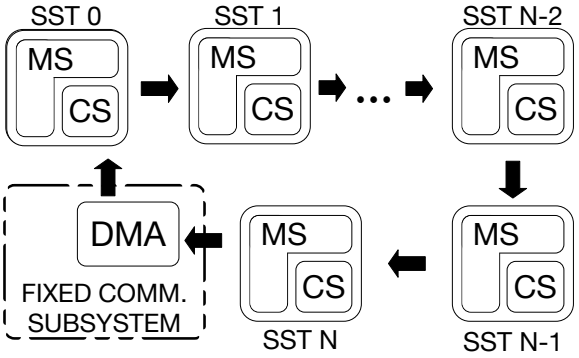
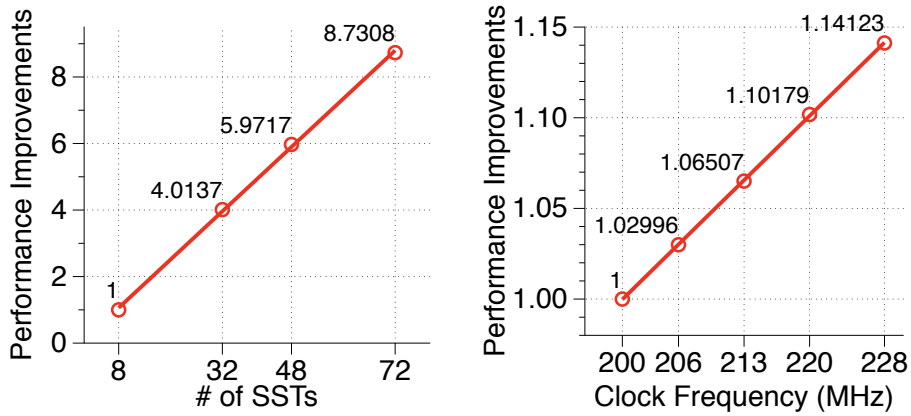


Figure 5.1: *Architecture of an SST-based accelerator for ISL.*

SST performance evaluation As discussed in [70], the performance of such architecture scales almost linearly with the increase in the number of SSTs (Figure 5.2(a)). Indeed, as the overall number of iterations of an ISL algorithm is usually very large, performance can be improved by maximizing the number of modules in the pipeline within the limits on the available device resources. The result is that the accelerator implements a higher fraction of the total number of time-steps of the ISL, reducing the number of needed sweeps through the accelerator. From an analytical point of view, the performance of this architecture can be estimated as:

$$T_q = \frac{T_0}{q} \tag{5.1}$$

where T_q is the completion time of the system with q instantiated SSTs and T_0 is the one of a single module.



(a) # instantiated SSTs

(b) Frequency

Figure 5.2: *Experimentally-measured performance scaling for Jacobi2D algorithm.*

Within this work we also analyzed the impact of the clock frequency targeted in the implementation on the overall system performance. As an example, Figure 5.2(b) reports the performance speedup of a Jacobi2D algorithm featuring a single SST module, synthesized on a Xilinx Virtex XC7VX485T device by means of Xilinx Vivado. We notice that also frequency improvements result in a linear increase in performance. For this reason, we can enhance the speedup model taking into account both the number of SSTs and the clock frequency:

$$T_q = \frac{T_0}{q} \cdot \frac{f_0}{f_q} \tag{5.2}$$

where f_q and f_0 are respectively the improved clock frequency and the original one.

As an additional experiment, we used the analytical approach of resource occupation provided in the original methodology [70] to identify the maximum number of SSTs instantiated in the pipeline. Unfortunately, this is an iterative approach that overestimates the number of SSTs that can be implemented in the final design, and, performs several synthesis to find the actual number with a trial-and-error flow, thus requiring a considerable design time. In the Jacobi2D example, the approach started from 127 modules and converged to 88 instances at 206 MHz in a design phase lasting for 406 hours. This highlights that such an approach is not accurate and efficient; as we will demonstrate in this work a more accurate estimation of the resources based on a preliminarily floorplanning of the design on the target

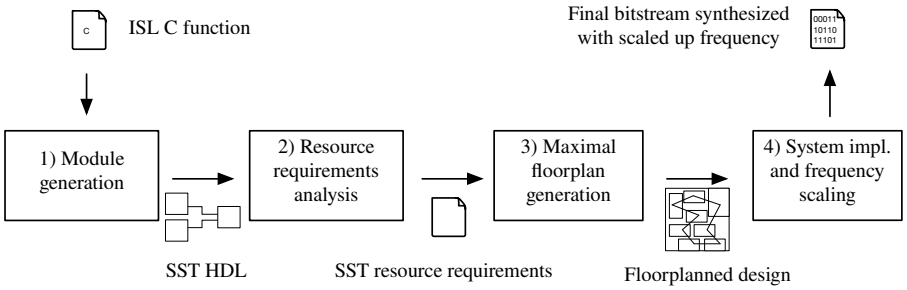


Figure 5.3: *The proposed design flow for the streaming architectural template, its main phases, inputs and outputs.*

device grid offers the possibility to directly identify the actual number of SST modules that can be instantiated. Nevertheless, a specific floorplanning activity may also allow to reach final implementations guaranteeing the timing closure at clock frequencies not reachable by an unconstrained implementation. In our running example, we were able to implement up to 90 SSTs at 228 MHz in a design phase lasting 25 hours.

We may conclude that there is room for a large improvement both in the performance of the SST-based architecture under consideration and in the duration of the design process. This issue has been addressed in this work by defining a novel design flow to optimize the result achieved by the methodology in [70].

5.3 Design flow

This section provides a high level overview of the proposed design flow for the streaming architectural template and, subsequently, discusses the details of its phases.

5.3.1 High level overview

The objective of the design flow is twofold: first, maximize the performance of the overall system by determining the maximum number of SSTs that can be instantiated onto the FPGA and by allowing to increase the operational frequency; second, reduce the overall synthesis and implementation time that would be incurred with simple iterative Domain Space Exploration (DSE) approaches. In order to achieve both objectives, we leverage floorplanning. On one side, by optimizing the floorplan of both the fixed communication subsystem and each SST, we are able to increase the fre-

quency of the system and the number of SSTs that can be instantiated while gaining timing closure. On the other hand, thanks to floorplanning, we can directly determine the number of SSTs that can be instantiated without running multiple implementation attempts thus requiring significantly long design times.

The overall design flow, depicted in Figure 5.3, is internally organized in four phases, and is interfaced with a commercial synthesis and implementation tool (Xilinx Vivado has been here used). Phase 1 starts by generating the Hardware Description Language (HDL) of an SST module from a C language description of the ISL computation. This phase is well described in [70] and does not constitute a contribution of the proposed streaming architectural template. In Phase 2, we consider a base design consisting in a single SST accelerator and the fixed communication subsystem generated from Phase 1. Such design is synthesized in order to obtain a first SST resource estimation; the module is further refined by floorplanning the design and validating the place and route results against several possible choices in the size and shape of the placement region. Once the minimal SST region size is determined, Phase 3 leverages an Integer Linear Programming (ILP) model to maximize the number of SST regions that can be floorplanned and solves an euclidean Traveling Salesman Problem (TSP) to find an optimal interconnection order for the SST accelerators. Finally, Phase 4 implements the full design with the identified floorplanning constraints and runs multiple synthesis to determine the maximum design frequency.

Phase 2 and 3 require a preliminary generation of a set of placements covering a given amount of FPGA resources. Hence, we first present this preliminary placements generation process and the FPGA characterization with respect to the proposed design flow. Then, the details of the four phases are discussed.

5.3.2 FPGA model and placements generation

Similarly to most of the floorplanning algorithms available in the literature [73, 85], we model an FPGA device as a matrix of tiles, each one containing a single type of resource such as Control Logic Blocks (CLBs), Digital Signal Processings (DSPs) and Block RAMs (BRAMs).

Generally, the tile granularity is dependent on the type of addressed problem; for a better description of the tile constraints that are needed when considering Partial Dynamic Reconfiguration (PDR), we refer the reader to Chapter 6 that specifically targets this problem. Nevertheless, since in our scenario we are addressing only static designs, in order to maximize the

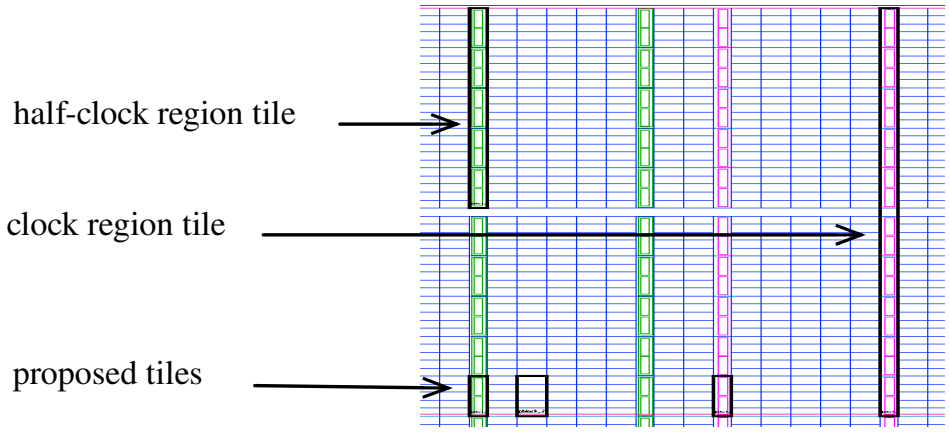


Figure 5.4: *Different tile sizes for a Xilinx Virtex7 FPGA device.*

overall utilization of the device, we take into account a small tile granularity, as shown in Figure 5.4. The considered tile spans a single resource wide, while its height covers the minimum integral number of resources for each resource type (e.g. either 2 DSPs, 2 BRAMs or 5 CLBs for Virtex7 devices).

We denote with W and H the number of tiles on the horizontal and vertical directions respectively. A placement $p = (x, y, w, h)$ on the tiles grid, is defined as a rectangular shape starting in the bottom-left corner (x, y) and spanning w tiles wide and h tiles height. Furthermore, we denote with $r_p = (n_{CLB}, n_{DSP}, n_{BRAM})$ the resource vector of placement p that specifies the number of CLB, DSP and BRAM covered by the placement.

Upon the presented FPGA model, we here define an utility routine leveraged by Phase 2 and 3 of our design flow that is in charge of the generation of the set of all the possible placements for a single SST. In our formulation, a valid placement is defined by specifying a width w such that: 1) the region has the minimum width to form a bounding box covering all the resources required by the SST module, 2) it presents a given aspect ratio (i.e. the width/height ratio). Algorithm 3 shows the procedure for SST placements generation; the function *searchWidth* uses binary search to find the minimum width in the range $[minW, maxW]$ compatible with the aspect ratio constraint. If the *searchWidth* is unable to find such value, or the identified placement overlap with a hardware macro or forbidden user-defined area, the function returns 0.

After several experimental tests, we noted that a maximum aspect ratio of 2.5 provides the best performance in terms of capability to maximize the

Algorithm 3 SST placements generation

```
1: a ← maximum aspect ratio (≤ 1)
2: r ← resource requirement vector
3: P ← ∅
4: for x ← 0 to W − 1 do
5:     for y ← 0 to H − 1 do
6:         for h ← 1 to H − y do
7:             minW ← a · h
8:             maxW ← (1/a) · h
9:             w ← searchWidth(x, y, h, minW, maxW, r)
10:            if w > 0 then
11:                P ← P ∪ (x, y, w, h)
```

number of SST while still allowing to meet timing closure. Hence in the rest of the chapter, such value is used for the SST placements generation.

Additionally, since all the SST designs share the same fixed communication subsystem, the floorplanning of such part is only dependent on the chosen FPGA device. Even though we do not directly target the placement of this subsystem, we noted that a simple approach to find a good placement is to synthesize a design without any SST and note the area in which the place and route tool places the components. Secondly, a floorplan constrain can be directly derived starting from the place and route suggestion. For the rest of the discussion we assume that the placement for the fixed subsystem is already defined and no placements overlapping with such region are generated by Algorithm 3.

5.3.3 Phase 1: module generation

Phase 1 follows the steps discussed in the methodology in [70]. In particular, we first verify whether the C function features an ISL computation by extracting its polyhedral representation via Clan[1]. If the function abide by such requirements, Candl[2] is used to construct the data dependency graph on which several transformations are applied in order to generate the implementation of an SST module defined in Vivado HLS C++ code. Finally, Vivado HLS is used to generate the HDL code for the SST module.

5.3.4 Phase 2: resource requirements analysis

Phase 2 consists in determining the minimum resource requirements for a placement hosting an SST module. Indeed, we empirically observed that, in most cases, Xilinx Vivado returns a conservative estimation of the required

[1] http://icps.u-strasbg.fr/~bastoul/development/clan/

[2] http://icps.u-strasbg.fr/people/bastoul/public_html/development/candl/

CLB resources; hence, if we floorplan an SST module by considering such an estimation without providing any additional resource margin, the subsequent place and route phase tends to fail.

The first performed step is the synthesis of a simple design containing only a single SST and the communication subsystem. Once the synthesis is performed, we collect the estimated SST resource vector $\hat{r}_{SST} = (\hat{n}_{CLB}, \hat{n}_{DSP}, \hat{n}_{BRAM})$ and generate the set of placements $\hat{P}$ that covers the resource vector $\hat{r}$. Among the identified placements, we select the one that covers the least amount of CLB resources and do not overlap with the predefined area for the fixed components. Finally, we perform the place and route of the design using a runtime optimized strategy (such as the *Flow Quick* implementation strategy available in Xilinx Vivado) that limits the overall implementation time. Depending on the success or failure of the place and route, we repeat the entire procedure by respectively decreasing or increasing $\hat{n}_{CLB}$ by $\Delta = 5$ CLB tiles (corresponding to half a CLB tile on Virtex7 devices). The algorithm terminates once the minimal placement size for which the place and route succeeds is determined.

It is worth noting that, even if we are running multiple place and route iterations, they are performed on a small design and hence the required time is often negligible with respect to the full design synthesis and implementation. As we will show in the experimental section, each place and route run takes up to a few minutes, while the final synthesis of the whole design lasts several hours.

5.3.5 Phase 3: maximal floorplan generation

In Phase 3, based on the resource requirements derived from the previous phase, we simultaneously identify the maximum number of SSTs that can be placed into the device and their floorplanning constraints. More formally, given a resource vector r for a single SST and the set of placements P generated using Algorithm 3, the objective is to find a maximal subset of placements $F \subseteq P$ such that no two distinct placements $p_1, p_2 \in F$ overlap.

It is worth noting that this problem is quite different from classical FPGA floorplanning problems, such as the one discussed in Chapter 6; indeed, in this scenario we are not given a specific set of regions to floorplan, but we need to maximize their number. Nevertheless, even if classical floorplanners [73, 85, 105] can be adapted to iteratively increase the number of regions to determine the highest feasible number, they might require a considerable number of attempts. Furthermore, as we will show in the experimental section, solutions obtained by re-adapted recent approaches

such as [73] are in general suboptimal in our context.

In order to identify such a maximal floorplan, we propose an ILP-based strategy that allows to find the optimal number of placements within a limited amount of time. For every placement $p \in P$, we associate a binary variable x_p that is set to 1 if and only if the corresponding placement p is selected in the final solution. Hence the objective of the ILP can be simply written as:

$$max \sum_{p \in P} x_p \tag{5.3}$$

Moreover, to guarantee the feasibility of the ILP solution, the selected placements must not overlap. As further detailed in Chapter 6, such constraint can be translated into requiring that at most one placement can be selected out of those covering a specific FPGA tile:

$$\forall\, xt \in \{0, 1, \ldots, W-1\},\ yt \in \{0, 1, \ldots, H-1\} : \sum_{\substack{p=(x,y,w,h) \in P \mid \\ x \leq xt < x+w, \\ y \leq yt < y+h}} x_p \leq 1 \tag{5.4}$$

Once the ILP model is solved, the maximal subset of placements is $F = \{p \in P \mid x_p = 1\}$. It is worth noting that this ILP formulation corresponds to solve the maximum independent set problem on a graph $G(P, E)$ where P is the set of nodes and for each pair of overlapping placements p_1, p_2 there exists an edge $(p_1, p_2) \in E$. In our formulation, Equation (5.4) represents the clique constraints that ensure that in the final solution F no two placements p_1 and p_2 are connected by an edge (i.e. they do not overlap). Since finding the maximum independent set is $\mathcal{NP}$-complete [48], it is unlikely that there exists an efficient incremental floorplanner that, starting from a partial solution $F' \subseteq P$ in which some of the placements are already selected, guarantees to find the optimal solution.

At this stage of the phase we have identified the number $|F|$ of SSTs and their floorplanning constraints, but we have not yet taken into account the modules interconnections. To optimize the interconnections among the SSTs, we exploit the regular design structure shown in Figure 5.1. Indeed, it is easy to note that the interconnection topology of the SSTs and the fixed subsystem form a ring. Furthermore, since the SSTs are all replicas of the same basic module, we are allowed to interchange the connections of two elements in the ring topology without affecting the functionality of the design. Thanks to these properties, the problem of optimizing the SST interconnections translates into finding a minimum TSP tour that vis-

its the centers of each SST placement and the fixed part of the design exactly once. For the TSP optimization, we consider the euclidean distance among the centroids of the placements, as it ensures no overlap among the interconnections, and we leverage an exact TSP solver to find the optimal solution. Notice that the number of modules in the final design is usually below 150/200 units, hence in practice, the time to find an optimal euclidean TSP is very small. An example of a final result achieved after this phase is shown in Section 5.5.

5.3.6 Phase 4: system implementation and frequency scaling

In Phase 4, the design flow explores different target frequencies to determine the maximum one that allows timing closure during the system place and route. The exploration is performed within an interval $I = [f_{min}, f_{max}]$ by first implementing a design at frequency f_{min} to verify the existence of a feasible design, and subsequently exploring with binary search other target frequencies in the interval. Due to technological constraints, the actual frequency values that can be selected in the interval I consist in a small set of frequency choices $f_0, f_1, \cdots, f_n$. Hence, the binary search terminates returning frequency f_i as soon as it verifies that timing closure is met at frequency f_i but not at f_{i+1} or if $f_i = f_{max}$. Notice that compared to the SST maximization, frequency exploration is a time consuming process since each binary search iteration requires to re-synthesize the overall design. Nevertheless the same design time overhead would also apply for the approach discussed in [70] since no explicit frequency estimation approach is defined. After several tests, we noted that a suitable frequency range for the frequency exploration process on a Xilinx Virtex7 devices is $[160MHz, 250MHz]$. Such interval will be also used within the experimental section.

5.4 Integration in CAOS

In this section, we highlight the changes performed to the default CAOS modules to add support for the streaming architectural template and its underlining design flow.

5.4.1 CAOS frontend

Similarly to the Master/Slave and dataflow architectural templates discussed in Chapter 3 and Chapter 4, we only modified the *architectural template applicability check* module within the CAOS frontend. For a given func-

tion, we leverage on the Clan tool to verify whether the polyhedral model can be used to represent the function and if it describes an ISL computation. Subsequently, the module also checks if the architecture description matches one of the boards supported by [70], namely the Xilinx VC707 (XC7VX485T) and the Xilinx Zynq Zybo (XC7Z010). The reason for this restriction is that we rely on the TCL scripts and templates from [70] for the generation of the final system implementation.

5.4.2 CAOS functions optimization

The optimization of the number and the placement of the SST modules is performed within the CAOS functions optimization flow. In this flow, Phase 1, Phase 2 and Phase 3 of the streaming architectural template are implemented within distinct CAOS modules.

Static code analysis

The *static code analysis* module has been extended to implement Phase 1. In particular, it applies the methodology in [70], which leverages on polyhedral analysis, to generate the HDL implementation of a single SST. Additionally, the module also reports the number of clock cycles needed to complete the processing of a single time-step by assuming a default frequency of 200 MHz.

Resource estimation

Phase 2 of the streaming architectural template cleanly maps to the CAOS *resource estimation* module. Here, the SST implementation is synthesized using quick synthesis options to obtain a more accurate estimation of the resource requirements of the module than the one that would be achieved via HLS.

Performance estimation

Phase 3, which is at the core of the design space exploration of the streaming architectural template, is implemented within the CAOS *performance estimation* module. The resource estimations from the previous module are used to generate the set of feasible placements for an SST on the target FPGA fabric. Subsequently, the module generates a maximal floorplan of the SST modules following the approach described in Section 5.3.5. If a feasible solution is found, the module outputs a single CAOS optimization dubbed as *SST queuing*. The optimization specifies the number of SST

modules that can be placed, their floorplanning constraints and their connectivity. Finally, the module estimates the performance of the provided optimization in terms of overall execution time of the accelerated FPGA computation. The estimation is done by considering the overall number of time-steps of the computation, the number of allocated SSTs and the number of cycles that an SST takes to compute a single time-step as provided by the *static code analysis* module.

Code optimization

The modification applied to the default *code optimization* CAOS module are relatively simple. Indeed, since the user can only select the *SST queuing* optimization, the module simple stores the parameters of the optimization in a dedicated file within the code archive that is later exploited by the CAOS backend for generating the final system.

5.4.3 CAOS backend

The CAOS backend flow tailored for the streaming architectural template mainly performs the steps of Phase 4 discussed in Section 5.3.5. The backend generates a Vivado project starting from a base template and instantiates as many SSTs as dictated by the *SST queuing* optimization. Subsequently, the flooprlanning constraints are used to specify the exact regions in which each SST should be implemented. Finally the design is synthesized, placed and routed multiple times using different target frequencies in order to identify the maximal one.

5.5 Experimental evaluation

The proposed streaming architectural template has been evaluated on three representative ISL computations (Jacobi2D, Heat3D and Seidel2D) targeting a Xilinx Virtex XC7VX485T device. The synthesis and implementation process has been performed using Xilinx Vivado 2015.4, the ILP models have been solved by means of Gurobi 7.0.2 [44], and the TSP problem has been solved by means of the exact Concorde TSP solver [3]. Experiments have been performed on a Intel Core i7-6700 CPU at 3.40 GHz with 32 GBs of RAM.

In our experimental campaign we compared the Proposed Approach (PA) against the baseline design methodology defined in [70], that for the sake of fairness in the comparison, has been enhanced with the final frequency exploration strategy defined in Section 5.3. Table 5.1 reports the

Table 5.1: *Analysis of the achieved system performance.*

Algorithm	# SSTs		design frequency (MHz)		PA performance improvement with respect to [70]
	PA	[70]	PA	[70]	
Jacobi2D	90	88	228	206	13.20%
Heat3D	25	25	228	206	10.68%
Seidel2D	19	19	183	183	0%

performance improvement achieved by the proposed approach over the baseline [70]. As can be noted, thanks to FPGA floorplanning we are able to increase the target frequency for the Jacobi2D and Heat3D algorithms of approximately 11%. Additionally, for the Jacobi2D case, the ILP model is also able to allocate two additional SSTs achieving a final floorplan further improving the performance up to 13%; the final floorplan of such system is reported in Figure 5.5. Nevertheless, the Seidel2D algorithm does not provide the same improvement figure. Indeed, since the total number of SSTs that can be placed into the design is small for this benchmark due to the internal complexity and resource requirement of the single SST module, the floorplanning reduces its impact on the overall design by leaving more room to the place and route algorithm. Additionally, the Seidel2D base SST is intrinsically more complex, as this ISL has *spatial dependencies* between point updates, and thus updates within a time-step are inherently sequential [70]. Due to this characteristics, the resulting SST design is more convoluted and more eager of logic resources.

The execution time as well as the number of synthesis required by the two approaches is reported in Table 5.2. As can be noted, the maximal floorplanning algorithm allows to greatly reduce the number of required synthesis within the SSTs maximization stage, thus leading to an execution time saving of 15.84x for Jacobi2D. Instead, the execution time of the frequency exploration phase does not change significantly among the two approaches as it only depends on the binary search iterations required to identify the maximum frequency.

Finally, we aimed at demonstrating also the higher efficiency of our strategy specifically targeted to the SST design with respect to the employment of one of the standard FPGA floorplanners proposed in the literature. To this purpose we adopted *PRFloor*, i.e. the most recent full-fledged strategy presented in [73][3] and used it in an iterative fashion by floorplanning an

[3]It is worth mentioning that we reimplemented the algorithm in [73] since it is not publicly available.

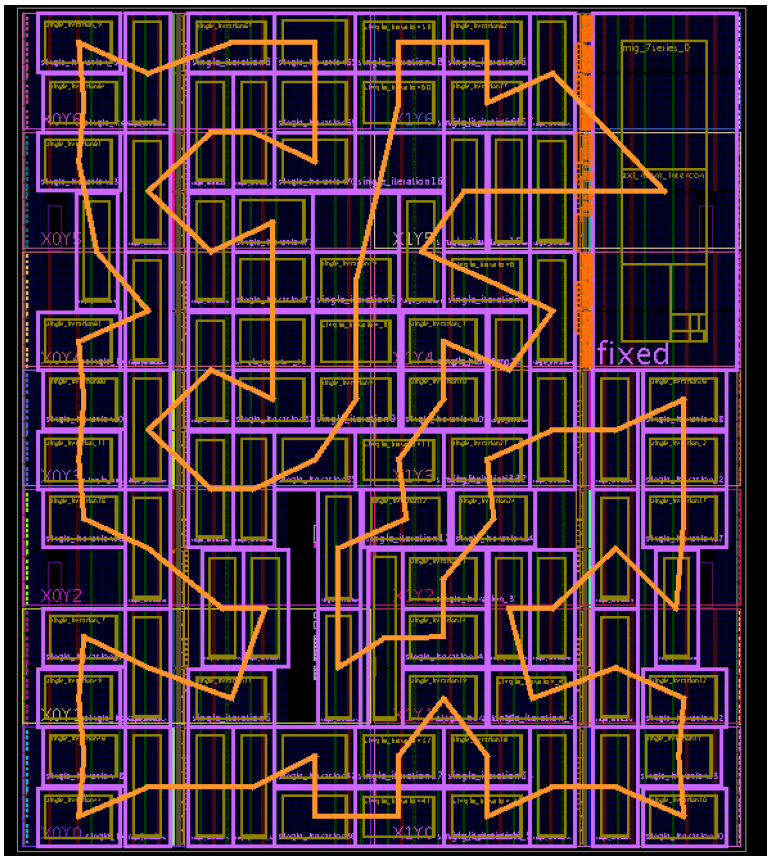


Figure 5.5: *Floorplan achieved by the proposed approach on the Jacobi2D algorithm. The fixed communication components are placed on the top-right corner, while the other regions constrain the position of 90 SST modules.*

Table 5.2: *analysis of the DSE execution time.*

Algorithm	SSTs maximization time (# synthesis runs)		frequency exploration time (# synthesis runs)		PA time reduction with respect to [70]
	PA	[70]	PA	[70]	
Jacobi2D	6.36h (1)	382.00h (40)	19.30h (3)	24.30h (4)	15.84x
Heat3D	4.19h (1)	13.91h (2)	14.66h (3)	17.94h (4)	1.69x
Seidel2D	4.95h (1)	8.60h (2)	17.31h (4)	15.42h (4)	1.08x

Table 5.3: *Comparison of the proposed approach and the iterative version of PRFloor [73] using half-clock region tiles.*

Algorithm	# SSTs		execution time	
	PA	PRFloor	PA	PRFloor
Jacobi2D	90	64	0.18s	4207.39s
Heat3D	20	18	1.77s	450.41s
Seidel2D	18	15	0.78s	209.97s

increasing number of SST modules until no more feasible solution is found. In order to align to the PRFloor algorithm, we here consider tiles spanning half a clock region. Furthermore, since the final trial-and-error placement procedure of PRFloor is not completely defined in [73], we adopted instead an exact ILP model equivalent to the one discussed in [85], but targeting only the subset of placements that are close to the anchor points identified by the PRFloor bipartitioning phase. As a consequence, the modified PRFloor is guaranteed to find a floorplan solution, if it exists, even if it might incur in a higher running time. Table 5.3 compares the number of SSTs identified by the algorithms as well as their running time. Notice that since we increased the granularity of the tiles, the numbers of SSTs found by our approach can be smaller than the ones presented in the previous experimental session. As it can be noticed from the table, PRFloor is not able to identify the maximum number of SST even if it uses a considerable amount of time. The cause is that the bipartitioner does not fit well the regular ring topology of the SST-based accelerator. Indeed, a ring topology leads to several optimal partitioning solutions, however, since PRFloor considers the horizontal and vertical cuts independently, it has a low probability of spreading the modules evenly on the FPGA.

5.6 Final remarks

In this chapter, we presented the streaming architectural template that allows to efficiently accelerate ISL computations written in C on FPGA. Starting from the initial design flow proposed in [70], we devised an automated design space exploration, capable of achieving the maximum performance level for a given FPGA through 1) the maximization of basic modules instantiated in the design and 2) optimization of the design floorplanning. The architectural template has been integrated into the CAOS platform, hence allowing users to quickly evaluate whether the streaming architectural template can be applied to one or more critical functions of the application.

Experimental results have demonstrated the capability of the proposed approach to reduce the design time up to 15x with respect to naive design space exploration approaches, and, at the same time, to improve the performance of the 13%. Such benefits have been achieved thanks to careful floorplanning of the base modules of the design. In Chapter 6, we will continue the discussion on automated floorplanning algorithms shifting the focus towards partially reconfigurable designs. Indeed, while floorplanning can be regarded as a commodity for the streaming architectural template, we foresee the possibility to integrate in CAOS architectural templates the rely on partial dynamic reconfiguration which require floorplanning as a key ingredient.

CHAPTER 6

CAOS backend: floorplanning for partially-reconfigurable designs

When dealing with Partially-Reconfigurable designs on Field Programmable Gate Arrays (FPGAs), floorplanning represents a required yet critical step that highly impacts system's performance and reconfiguration overhead. However, current vendor design tools still require the floorplan to be manually defined by the designer. In this chapter, we present a novel floorplanning automation framework, targeting the Xilinx toolchain, which is based on an explicit enumeration of the possible placements of each region. The proposed approach is suitable for being integrated within the CAOS backend flow discussed in Chapter 2 in order to automatize the implementation process for architectural templates that require partial-dynamic reconfiguration support.

6.1 Introduction

FPGA devices are nowadays widely employed in commercial and industrial appliances in many scenarios (e.g. telecommunication, automotive, high performance computing, video and image processing), due to their reduced costs, good computational power, and high flexibility since they can be reconfigured in order to change their functionality. Moreover, Partial Reconfiguration (PR) [112] has received a considerable attention in the recent years since it even more enhances such flexibility, by enabling the possibility to dynamically change only part of the modules at runtime while the rest of the system keeps working. Indeed, PR offers new opportunities such as the possibility to execute at different times more functionalities than the ones physically placeable on the device or the possibility to update or vary their implementations. In order to enable PR two necessary conditions must hold: 1) the FPGA device has to physically support the change of only a part of the configuration at runtime, and 2) the companion design tools have to support the implementation of such reconfigurable systems. In this scenario, Xilinx [110] is the vendor presenting the most mature solution.

The role of floorplanning [18] in PR-based system design is even more prominent than to the standard FPGA design flow. In fact, while in the latter, this activity is mainly of interest for expert designers aiming at achieving advanced performance optimization, in the former the implementation of a partially reconfigurable system forces to define the specific regions on the device fabric that will host the interchangeable functionalities. Therefore, floorplanning directly affects the feasibility and the performance of the final solution. However, it is a quite complex activity since the area constraints for the reconfigurable regions have to meet specific placement requirements (reported in [112]), while covering a minimum amount of configurable resources that are needed by the modules reconfigured over time in each of the regions. Nevertheless the internal architecture of FPGAs is becoming more and more advanced, exacerbating the floorplanning complexity. In fact, the homogeneous grid of Control Logic Blocks (CLBs) is alternated, most of the time in an irregular way, with columns of dedicated elements such as Block RAMs (BRAMs) and Digital Signal Processings (DSPs).

Commercial tool-chains still support floorplanning through visual instruments, such as Xilinx Vivado [110] (which integrates the previous PlanAhead tool). However, the designer still has to manually define the shape and the position of the reconfigurable regions, since the tool provides a limited automation on the regions definition that generally leads to unfeasible so-

lutions. Nevertheless, also other design flows for Xilinx FPGA devices proposed by the academia (e.g., [10, 114]) suffer from the same lack. On the other hand, several academic solutions have been presented in literature to automate floorplanning ([7, 13, 18, 63, 85, 98, 105, 115]). However, only few ones ([13, 85, 105]) take into account the requirements for PR, and, at the same time, accurately consider an arbitrary distribution of heterogeneous resources within the device. Indeed, most of the algorithms consider only one of the two aspects, i.e. the PR requirements (e.g. [63, 115]) or the resource distribution (e.g. [7, 18, 35, 98]). Finally, as it will be shown in this work, such comprehensive approaches generate suboptimal solutions. Another relevant consideration that can be drawn on most of such automation solutions is the fact that they are actually unconnected from the real design flow; in fact, only few of these engines ([10, 54, 71, 114]) are tested on real circuits and synthesize their final outcome on a real board to check for feasibility.

In this work we present a novel floorplanning automation framework fully integrated with the Xilinx design flow. The framework exploits a direct representation of the problem based on the enumeration of the feasible placements that is able to abstract the computational complexity of floorplanning exploration while taking into account all the relevant constraints for PR on recent devices and metrics such as area consumption and aspect ratio. We show that, differently from the classical problem for VLSI design, enumerating a suitable subset of the feasible placements for each reconfigurable region is a viable approach and can be also efficiently automated by means of classical optimization algorithms. In a previous work [82] we proposed a preliminary version of the framework where the design space exploration was automated by means of a Mixed-Integer Linear Programming (MILP) formulation. We here propose a more complete and mature framework featuring a new automation engine providing higher performance. In conclusion, we summarize our contributions as follow:

- We accurately model all the constraints in the current PR guidelines [112].
- We propose a direct formulation of the floorplanning problem based on a conflict graph of the feasible regions placements described in terms of the actual coordinates on the fabric grid.
- We propose a genetic algorithm extended with a local search strategy exploiting the defined problem representation. The algorithm is able to speed up the identification of near-optimal solutions in a limited elaboration time.

- Finally, we experimentally show the effectiveness of the proposed approach by comparing against various state-of-the-art solutions and alternative engines exploiting the same problem representation on both synthetic benchmarks and real case studies.

The remainder of the chapter is organized as follows. Section 6.2 discusses the related work in the area, while Section 6.3 presents a formal description of the problem. Then, Section 6.4 shows the proposed design flow, whose details are discussed in Sections 6.5 and 6.6 in which the feasible placements generation process and the floorplanning automation algorithm are presented respectively. Finally, Section 6.7 evaluates our approach on different problem instances, and Section 6.8 draws the conclusions. In addition, we report in Appendix 6.A a revisited presentation of the MILP formulation [86] that has been used as a baseline for the experimental evaluations.

6.2 Related work

Several floorplanners for FPGAs have been proposed in literature; however, most of them produce solutions that are either not compliant with PR requirements and guidelines (e.g. [7, 35, 98]), or only focus on a simplified device model, not capable of representing modern FPGAs lacking a uniform distribution of heterogeneous resources (e.g. [63, 115]).

One of the first algorithms that considers the heterogeneity of FPGA resources has been presented in [18]. The algorithm exploits simulated annealing over a slicing-tree representation, and, subsequently, performs a compaction step to recover from unfeasible solutions and to improve the shapes of the modules. However, the resulting floorplan unlikely produces shapes that meet the PR requirements. Furthermore, the approach assumes the FPGA to have a homogeneous resource distribution, i.e., BRAM and DSP columns are homogeneously spaced within the device fabric. Based on this assumption, the algorithm divides the fabric grid in a set of homogeneous blocks having the same size and containing the same amount of resources for each resource type (DSP, BRAM and CLB). However, this organization, which characterizes obsolete device families (such as Xilinx Virtex-II and Spartan 3), does not hold for the recent devices (e.g. Xilinx Virtex 6). Interesting aspects of such formulation are the Irreducible Realization List (IRL) and the dominance relation, that have been successfully borrowed in our problem representation as described in Section 6.5.

A similar approach considering a heterogeneous FPGA device has been proposed in [35]; it consists in 1) a simulated annealing algorithm explor-

ing a sequence-pair representation of the solution, and 2) a subsequent refinement of such solution by means of a Min-Cost Max-Flow formulation which alters the rectangular shapes of the reconfigurable regions. Due to this second phase, the approach in general does not satisfy PR requirements.

Another class of approaches ([63, 115]) introduces the time domain in the problem by handling the definition of reconfiguration operations together with the design of the floorplan. In [115] only logic blocks are taken into account while ignoring other types of resources available in the FPGA device. The work proposed in [63] considers both the partitioning of modules into reconfigurable regions and their floorplanning. During the partitioning phase, the algorithm assigns each of the modules to a reconfigurable region to minimize the wastage of resources over time. After partitioning, the resource requirements of the regions are known and the algorithm computes a floorplan by means of simulated annealing using moves that preserve the PR constraints. Even though the approach considers heterogeneous resources, similarly to [18] it assumes their regular and uniform distribution.

Differently from [63], other approaches ([7, 98]), called *multi-layer* floorplanners, analyze together the various circuit configurations the system assumes in different instants of time. Their aim is to identify a floorplan such that the common modules used in all the configurations are placed at the same position in all the circuit configurations. Such modules will represent the static area of the device, while the rest of the device is reconfigured as a whole. As a consequence, the reconfigurable part does not follow the Xilinx PR flow. Nevertheless, in [7] the device is assumed to have a homogeneous resource distribution as in [18].

A last class of floorplanners ([13], [105], [85]) considers both the PR constraints and an accurate description of the heterogeneous resource distribution. The work proposed in [13] stems from Parquet [2], the state-of-the-art fixed-outline floorplanner for VLSI design, and presents a non-trivial adaptation of the methodology to deal with partially reconfigurable FPGAs. The algorithm uses simulated annealing to perturb a floorplan representation that consists of a sequence pair augmented with a vector characterizing the aspect-ratio of the modules. Moreover, to increase the probability to detect feasible floorplans, it implements smart moves to recover from solutions in which the resource requirements are not satisfied.

The approach devised in [105] characterizes the FPGA device in terms of minimal reconfigurable units [112] called tiles. Each tile spans multiple configurable frames on the horizontal direction and contains a specific type and number of resources. Thus, the resource requirements of the re-

configurable regions are translated in terms of tile requirements and a technique called Columnar Kernel Tessellation is applied to search for floorplans that minimize the overall estimated bitstream size. A post processing step moves the obtained areas on the vertical direction trying to locally improve the wire length without affecting the occupation of resources.

Even though [13] and [105] give better results than [63] in terms of wire length and area occupancy respectively, [85] shows that the quality of their solutions can still be improved by means of analytic methods at the cost of a longer execution time. Specifically, [85] proposes two algorithms both based on a compact MILP formulation. The first algorithm is meant to locally improve the quality of an initial feasible solution with a relatively small computational effort. Instead, the second algorithm is able, in principle, to explore the full solution space and to find provably optimal solutions. Unfortunately both the algorithms require, to some extent, an initial feasible floorplan to achieve good final solutions. In our previous publication [82] we have demonstrated that solutions achieved by the approaches proposed in [85] (and consequently in [105] and [13]) can be further optimized without additional time penalties.

A final aspect to be considered is the experimental validation of the proposed solutions. Actually, only in few approaches ([10, 54, 71, 114]) the produced results are synthesized on the target device to check their feasibility and the achieved performance in terms of maximum clock frequency. In particular, in [54] an in-depth analysis of the effects of modules aspect ratio on the maximum achievable clock frequency is performed, while no automation strategies are presented. In [71] a similar analysis is performed by concluding that squared aspect ratios are preferable, and a very simple semi-automated floorplanner for pipeline designs based on a single chain of components is proposed. In [10], the floorplanning problem is tackled from a different perspective: the system is first synthesized without any constraint, and, then, an automated engine tries to identify a suitable set of placement constraints around the area used for placing and routing each module; unfortunately the approach is tested with a single reconfigurable region and it is unlikely to work with a larger number of regions. Finally, in [114] a simulated annealing is directly integrated with the synthesis tool to implement each explored solution; even though such a strategy presents a huge cost in terms of elaboration time.

Table 6.1 recaps the characteristics of the existing approaches showing the supported features. It is worth noting that the most efficient approach has been proposed in [86]; in fact, it outperforms (possibly in a indirect way) most of the relevant previous solutions supporting PR (i.e. [13, 35,

Table 6.1: *Comparative analysis of past approaches*

Approach	FPGA Model*	Reconfig. aware	PR support	Experimental comparison**	Exp. verified
Cheng et al. [18]	Homo.			[2]	
Feng et al. [35]	Heter.				
Yuh et al. [115]	CLB	✓			
Montone et al. [63]	Homo.	✓	✓		
Singhal et al. [98]	CLB	✓			
Banerjee et al. [7]	Homo.	✓			
Bolchini et al. [13]	Heter.	✓	✓	[35, 63]	
Vipin et al. [105]	Heter.	✓	✓	[63]	
Rabozzi et al. [85]	Heter.	✓	✓	[13, 105]	
Lamprecht et al. [54]	Heter.				✓
Neely et al. [71]	CLB	✓	✓		✓
Beckhoff et al. [10]	Heter.	✓	✓		✓
Yousuf et al. [114]	Heter.	✓	✓		✓
Rabozzi et al. [86]	Heter.	✓	✓	[13, 85]	
PA	Heter.	✓	✓	[86]	✓

(*) Device models with a homogeneous resource distribution, heterogeneous one and considering only CLBs

(**) The cell lists the approaches that have been tested and outperformed by the one of the current line

63, 85, 105]). However, the weakness it presents is the lack of an experimental validation of the achieved solutions while not all the current PR constraints are taken into account. In this chapter, we aim at proposing a novel floorplanning automation framework that, starting from the preliminary idea presented in [86], supports the peculiarities of modern FPGA devices and PR design flow, and features an even more efficient automation engine in terms of quality of the achieved solutions and elaboration time. Moreover, we also present an experimental validation of the approach by implementing real designs on a FPGA device.

6.3 Floorplanning problem description

This section provides some relevant background on the floorplanning problem, in particular focusing on the Xilinx FPGA devices and the design rules of the related PR flow.

As shown in Figure 6.1a, the reconfigurable fabric of an FPGA device is organized in a set of columns of resources of various types, that is

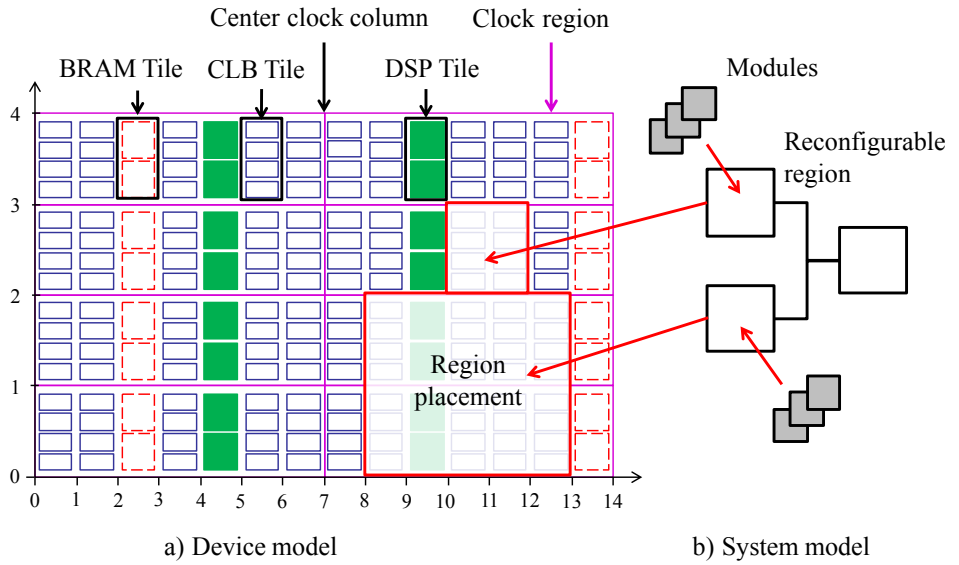


Figure 6.1: *Problem representation in terms of a) the reconfigurable FPGA device and b) the top level structural description of the system.*

$T = \{CLB, BRAM, DSP\}$. The grid is also divided in quadrants, called *clock regions* according to the structure of the clock tree and the organization of the configuration memory. Based on the memory organization, the basic reconfiguration portion of the device grid, that we call *tile*, spans one clock region height and one resource width. Each tile contains a single type of resource depending on the position of the tile, and the amount of units depends on the type of resource. Thus, as in [105], we consider a more abstract model of the FPGA organization in terms of a grid of tiles. Finally, we also define a coordinate system on the grid of tiles, starting from the bottom-left corner. We denote with W and H the maximum values on the X and Y axis respectively.

According to the PR design guidelines, as shown in Figure 6.1b, the reconfigurable system is specified in terms of a structural description of interconnected N top components called *reconfigurable regions*[1]. Each region implements a partially-reconfigurable unit in which it will be possible to load in a mutually-exclusive fashion a set of *modules* implementing different functionalities. Thus, the reconfigurable region n presents resource requirements that depends on the hosted modules; for each resource type t we denote the required amount as $r_{n,t}$. Moreover, the region is connected with the others and with the static part of the design (another component

[1] When clear from the context, we also refer to them simply as *regions*.

or set of components not featuring reconfiguration capabilities) by means of a set of interconnection buses, each one characterized by a width b in terms of number of wires. Do note that at floorplanning stage, positioning of the connections of the wires among the region boundaries is not handled; therefore, a center-to-center interconnection model is here adopted and the overall wire length is estimated using the classical half-perimeter wire length (HPWL) formula [7].

The goal of the floorplanning is to define a *placement* for each of the reconfigurable regions, in terms of rectangular shape and position on the FPGA resource grid. To this purpose, on the basis of the defined FPGA model, we denote with P the set of all possible placements that may be defined for the floorplanning of a single reconfigurable region:

$$P = \{(x, y, w, h) \mid x, y, w, h \in \mathbb{N}, \\ x + w \leq W, y + h \leq H\} \tag{6.1}$$

where x and y represent the coordinates of the bottom-left corner of the placement, while w and h define its width and height respectively. Thus, the specific placement p can be characterized in terms of the available resource capacity, denoted as $c_{p,t}$ (for each resource type t), depending on the specific position and shape. It is worth noting that in some devices (e.g. the Zynq device) specific placements are forbidden since they overlap with hard processors, static logic or I/O blocks. We represent such placements with the subset $S \subset P$, that will be discarded during the floorplanning exploration. For a formal description of the floorplanning requirements it is convenient to define a relation $\perp$ such that for $p_1, p_2 \in P$: $p_1 \perp p_2$ if and only if the two placements overlap on at least a tile. The non-overlapping relation $\not\perp$ is simply defined as the complement of $\perp$: $\not\perp = P \times P \setminus \perp$.

To be feasible, a floorplan must assign a placement p_n for each region n and satisfy a set of PR requirements:

REQ1: each assigned placement must contain at least the required resources for the corresponding region:

$$\forall n \in N, t \in T : c_{p_n,t} \geq r_{n,t} \tag{6.2}$$

REQ2: each assigned placement must not be forbidden:

$$\forall n \in N : p_n \notin S \tag{6.3}$$

REQ3: the left and right boundaries of a placement p_n must be aligned to specific coordinates that prevent splitting of interconnect resources

[112] (VL and VR enumerate valid left and right coordinates, respectively):

$$\forall p_n = (x, y, w, h) \mid n \in N : x \in VL \wedge x + w \in VR \tag{6.4}$$

REQ4: CLB resources at both sides of the center clock column must lie in the static part of the design, an assigned placement can cross the center column but such resources are not available for the corresponding region:

$$\forall t \in T : c_{(x_{clk}-1,0,2,H),t} = 0 \tag{6.5}$$

REQ5: placements assigned to two different regions cannot overlap:

$$\forall p_{n1}, p_{n2} \mid n1, n2 \in N \wedge n1 \neq n2 : p_{n1} \not\perp p_{n2} \tag{6.6}$$

This list of constraints can be partitioned in two groups: REQ1-REQ4 are specifically related to the placement p_n for a single region n, while REQ5 rules the relative positions between different regions. Moreover, the first set can be summarized in a single definition by introducing a new set P_n, which represents all the feasible placements on the device for a reconfigurable region n.

In conclusion, the floorplanning problem can be stated as follows: *Given the sets P_n of feasible placements, a floorplan is a function f that assigns for each region $n \in N$ a placement $p \in P_n$ such that there is no overlapping among the placements*. More formally:

$$\begin{aligned} &f : n \in N \rightarrow p \in P_n \\ &f(n_1) \not\perp f(n_2) \;\; \forall n_1, n_2 \in N : n_1 \neq n_2 \end{aligned} \tag{6.7}$$

6.4 Proposed floorplanning framework

The structure of the proposed floorplanning automation framework and its integration in the Xilinx design flow is depicted in Figure 6.2. The design flow implemented in Xilinx Vivado consists in three main automated steps:

1. *Synthesis*, which takes the input Hardware Description Language (HDL) structural specification of the system and translates it in an intermediate netlist,

2. *Implementation*, which performs the place and route of the netlist on the selected FPGA device, and,

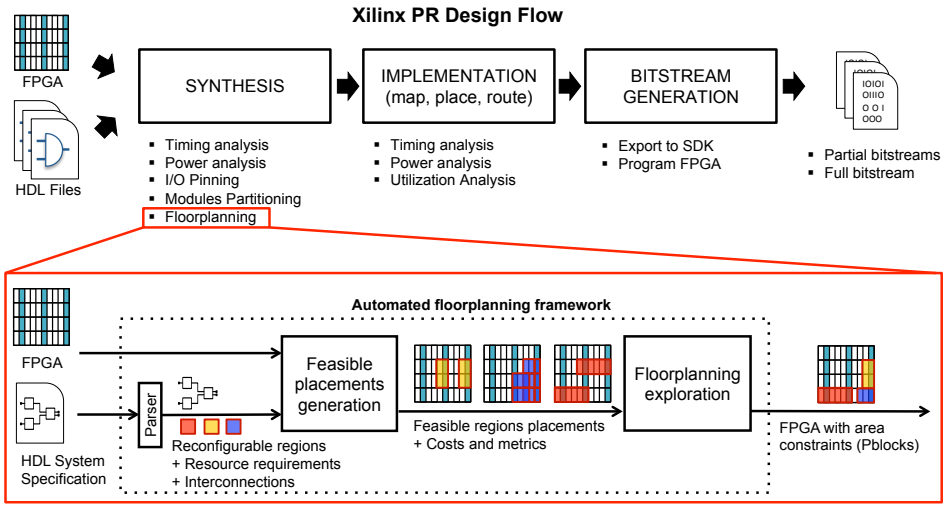


Figure 6.2: *The proposed floorplanning framework integrated within the Xilinx PR design flow.*

3. *Bitstream Generation*, which generates the final partial and complete configuration files.

Moreover, after each phase, a set of manual steps can be performed to define specific aspects and set parameters (such as the selection of the I/O pins or the floorplanning of the modules), while the obtained circuit can be analyzed by means of utility tools (for instance to estimate the power consumption or the timing of the netlist).

In this scenario, the PR design flow is an enhancement of the standard flow able to handle the fact that several modules can be implemented in the same reconfigurable region. During the three phases, partial specifications, related to the modules and the top level of the design, are used for the synthesis and the implementation of sub-circuits and the subsequent generation of partial bitstreams. Within this scenario, the floorplanning is a manual activity executed before the implementation phase, immediately after the definition of the reconfigurable regions within the top level specification.

The proposed floorplanning automation framework replaces the corresponding manual activity in the considered PR design flow. The framework takes in input the HDL structural specification of the system and translates it in an internal agile representation based on a graph. Moreover, it exploits synthesis reports to collect information on the resource requirements, that will be annotated on the system internal representation. The overall resource requirements of each reconfigurable region are com-

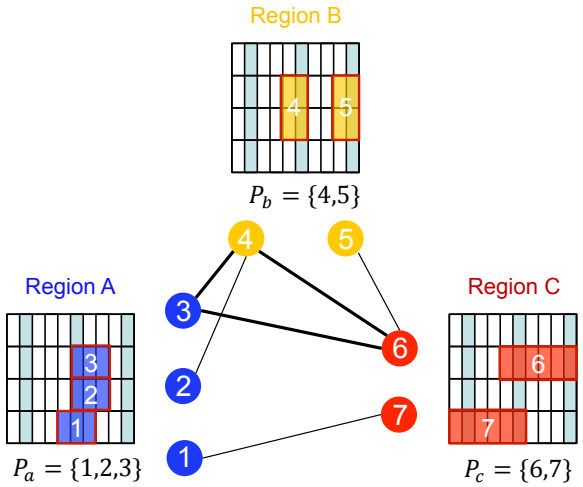


Figure 6.3: *Example of conflict graph.*

puted as the maximum requirements among the various modules that will be hosted within the specific region. Finally, the framework takes in input the description of the considered FPGA device modeled as discussed in Section 6.3. The output of the framework is the set of rules describing the floorplan solution, specified in the Xilinx constraint language to be imported in Vivado in order to continue with the subsequent implementation phase.

According to the formalization of the problem presented in the requirements REQ1-REQ5, the floorplanning automation framework is divided in two different phases:

1. **Feasible placements generation**, that consists in building a conflict graph where nodes represent the union of the P_n sets of possible feasible placements for each reconfigurable region n (REQ1-REQ4), and edges represent the overlapping among pairs of placements of different regions (REQ5).

2. **Floorplanning exploration**, that selects a possible placement in P_n for each reconfigurable region n such that there is no overlap among placements (REQ5) and an objective function specified by the designer is maximized.

We automated the two phases with different strategies according to their peculiarity and computational complexity. In particular, the first phase performs an exhaustive exploration for the definition of the conflict graph,

since, as shown in the next section, the problem has a limited complexity. For the second phase, characterized by a considerably larger design space, the framework features an efficient exploration engine powered by a genetic algorithm extended with a local search strategy. As shown in the experimental session, this strategy provides near-optimal solutions with a very limited execution time. Nevertheless, as shown in Section 6.7, the framework supports the integration of further automation strategies. The two phases are discussed in more details in the following sections.

6.5 Feasible placements generation

The first phase of the proposed framework is devoted to the definition of an abstract model called *conflict graph* that describes all the feasible placements for the various reconfigurable regions and the possible conflicts among pairs of placements. As shown in Figure 6.3, the conflict graph contains a group of nodes for each reconfigurable region n representing the overall enumeration of the feasible placements P_n, computed by fulfilling requirements REQ1-REQ4. Moreover, edges are used to represent conflicts between pairs of placements of two different regions, according to requirement REQ5.

It is worth noting that in the classical floorplanning for VLSI design it is commonly agreed that such a problem representation, based on the direct specification of all the possible region coordinates on the device grid, is extremely inefficient for automated optimization due to the huge solution space it defines. For this reason, past approaches have exploited various indirect representations, such as slicing trees, sequence pairs or other hierarchical tree-based representations [39]. At the opposite, the PR guidelines cause a considerable decrease in the number of feasible placements for a single reconfigurable region, thus allowing to effectively exploit such a direct representation of the placements. As an example, Figure 6.4 reports the number of feasible placements generated for a single reconfigurable region when varying its resource requirement on a specific Xilinx XC7V585TFF6 device; the number of all feasible placements under PR constraints (P_n) is two order of magnitude smaller with respect to the number of all the possible placements that can be generated on the device without constraints (namely *No PR* in the figure). It is worth noting that only CLBs are considered as resource requirements, while taking into account also other resource types would have even more decreased the number of placements.

The number of placements to be explored for the floorplanning can be even more reduced with respect to P_n, if we consider that in most of the

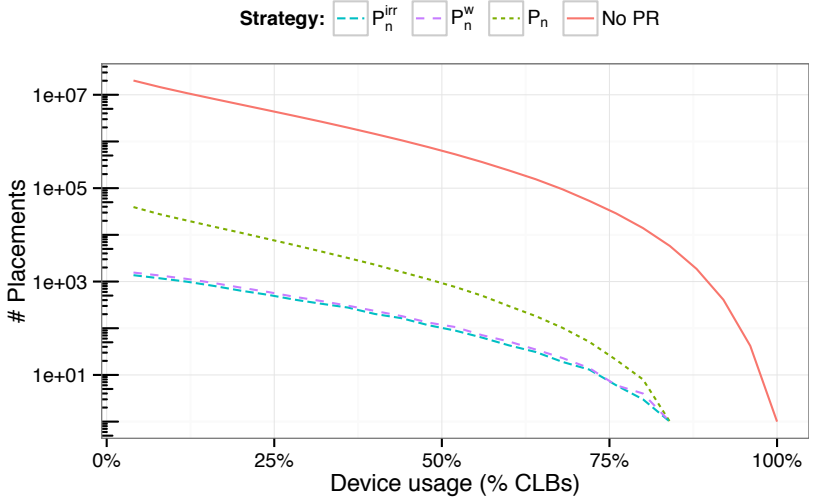


Figure 6.4: *Number of feasible placements for a single reconfigurable region.*

cases, from an optimization point of view it is not efficient to define placements larger than the minimal bounding boxes containing the required resources. In fact, as discussed in [18] and [13], by using the minimal bounding boxes it is possible to reduce resource utilization, thus leaving space for additional functionalities and reducing the reconfiguration time. For this reason, we define a new set P_n^{irr} containing the *irreducible placements* of a region n as:

$$P_n^{irr} = \{p \in P_n \mid \nexists p_2 \in P_n : p_2 \neq p \wedge p_2 \prec p\} \tag{6.8}$$

where $\prec$ represents a containment relation between two different placements of the same region. More formally, given two placements $p_1 = (x_1, y_1, w_1, h_1), p_2 = (x_2, y_2, w_2, h_2) \in P_n$, we have $p_1 \prec p_2$ if and only if $x_1 \geq x_2$, $y_1 \geq y_2$, $x_1 + w_1 \leq x_2 + w_2$ and $y_1 + h_1 \leq y_2 + h_2$. As shown in Figure 6.4, P_n^{irr} allows to reduce the size of the conflict graph of about another order of magnitude compared to P_n.

Even if P_n^{irr} sets are well-suited for optimizing resource occupation, we have experimentally noted that they may produce suboptimal results in terms of global wire length among various regions. For this reason, we have slightly relaxed the definition of P_n^{irr} to consider also placements that are required to be minimal only with respect to the horizontal direction. For a formal definition of this set of placements we consider a weaker containment relation $\prec^w$: given the two above placements p_1 and p_2, we have $p_1 \prec^w p_2$ if and only if $x_1 = x_2$, $y_1 = y_2$, $h_1 = h_2$ and $w_1 \leq w_2$. Thus the

Algorithm 4 Width-reduced placements generation

```
1: for each n ∈ N do
2:     P_n^w ← ∅
3:     for each x ∈ VL do
4:         for y ← 0 to H − 1 do
5:             for h ← 1 to H − y do
6:                 w ← searchMinimalWidth(x, y, h, n)
7:                 v ← validAspectRatio(x, y, h, w)
8:                 if w > 0 ∧ v = true then
9:                     P_n^w ← P_n^w ∪ (x, y, w, h)
```

corresponding width-reduced placements set for region n is defined as:

$$P_n^w = \{p \in P_n \mid \nexists p_2 \in P_n : p_2 \neq p \wedge p_2 \prec^w p\} \tag{6.9}$$

For the three defined sets the following relation holds:

$$P_n^{irr} \subseteq P_n^w \subseteq P_n \tag{6.10}$$

It is worth noting that the choice of reducing the placements only on the horizontal direction is suggested from the structure of current devices. Usually H is much more coarse-grained than W; as an example a Xilinx Virtex-5 XC5VLX110T is described using 8 rows and 62 columns of tiles ($W = 62$ and $H = 8$) [85]. On average, with respect to different CLB resource requirements, this relaxed strategy leads to 14% more placements with respect to P_n^{irr}, as shown in Figure 6.4.

A last relevant issue related to the generation of the feasible placement is the *aspect ratio*, that is the ratio between the width and the height of a placement. Indeed, as discussed in [71] and [54], extreme aspect ratios (e.g. lower than 1:5 or higher than 5:1) often lead to implementations with high routing congestion and low performance. This issue is mainly suffered on the vertical direction since its axis is more coarse-grained. Thus, placements with such elongated shapes can be filtered during the placement generation process; then, among the available ones, higher cost can be attributed to placements with extreme aspect ratios during the exploration phase.

Algorithm 4 automates the computation of the sets P_n^w. The procedure executes an extensive search of the possible placements by scanning all the valid coordinates of the device starting from the bottom left corner of the FPGA. For each point on the coordinate system, the algorithm considers all the possible placement heights that do not exceed the boundaries of the device and search for the minimal width needed to cover the required resources. Notice that depending on the resource requirements and on the

presence of hard processors and static logic, the search for the minimal width can fail; in these situations the corresponding placement is not generated. In case of success, the *searchMinimalWidth* function returns the minimal width for the current starting point and height. If it is not possible to find a feasible placement for the given height and position, the method returns 0. At the same time the *validAspectRatio* function is called to check if the current placement has to be discarded due to extreme aspect ratios.

From an asymptotic complexity point of view, *searchMinimalWidth* function is the most time consuming operation in the innermost loop. By using a binary search and pre-computing the resources occupied by each placement on the device, this function can be implemented with a $\mathcal{O}(logW)$ time complexity. Since the set VL, representing valid left coordinates for the regions, has a size proportional to W, *searchMinimalWidth* function is invoked $\mathcal{O}(|N| \cdot H^2 \cdot W)$ times. In conclusion, Algorithm 4 has an overall time complexity of $\mathcal{O}(|N| \cdot H^2 \cdot W \cdot logW)$. Notice that H and W are usually small numbers and the placements generation takes only a small amount of time compared to the overall optimization of the floorplan. Another observation, deriving directly from the algorithm, is that the number of width-reduced feasible placements for a region cannot exceed $H^2 \cdot W$. As discussed in Section 6.7, in real situations the number of feasible placements is limited up to few thousands items per set, and therefore leads to manageable conflict graphs and to a placements generation time of few seconds.

6.6 Floorplanning exploration

Once the feasible placements are generated, the second phase of the proposed floorplanning framework consists in the choice of the most suitable placement p for each region n among the available ones, such that 1) all selected placements do not overlap, meaning that no conflict edge exists among pairs of such placements, and 2) a specified objective function is optimized. The proposed floorplanning automation framework supports the integration of any automation engine capable of solving such an exploration problem by working on the defined conflict graph. It is worth noting that even if the conflict graph has a small size, the solution space, represented by the Cartesian product of the sets of feasible placements of each reconfigurable region, has a size that grows exponentially with respect to the number of regions, thus motivating the necessity of an efficient exploration engine.

In the preliminary formulation of the framework [82] we adopted an ex-

ploration engine based on an exact MILP model, whereas, within this work we designed and tested various heuristic methods in order to improve the floorplan exploration time. During our experimental sessions, we identified the Genetic Algorithm (GA) engine extended with steepest descent local search to be the most effective approach in finding near-optimal solutions in a reduced amount of time. Therefore, we here present the GA engine in details, while we refer the reader to Appendix 6.A for a description of the MILP model that has been used as a baseline. From experiments conducted with different placement sets, we also noticed that the width-reduced placements P_n^w offer the best trade-off in terms of size of the resulting solution space and quality of the achievable results. Hence, in the following discussion we refer to sets P_n^w, even though the approach is still valid when other sets of placements such as P_n or P_n^{irr} are considered.

The proposed GA engine for automating the floorplanning exploration is based on the classical simple Genetic Algorithm formulation [41]. We defined a solution encoding exploiting the enumeration of the feasible placements identified during the first phase of the floorplanning framework. More precisely, the *chromosome* is a linear vector where each position represents a reconfigurable region n, and the contained value refers to the feasible placement p in the corresponding set P_n^w. Then, the standard *crossover* and *mutation* operators have been employed. The *crossover* operator cuts in a random point the chromosomes of two parent solutions and exchanges the second parts to generate two children, while the *mutation* operator replaces with a given probability the placement of a region with another randomly selected placement in P_n^w.

In order to evaluate the solution we consider two different cost metrics:

- A_{cost}, the cost directly related to placement selection.
- W_{cost}, the cost deriving from inter region wire length.

The first contribution can be easily computed summing the cost $a_{p,n}$ associated to each placement $p \in P_n^w$ that is selected for the current floorplan. As an example, the cost $a_{p,n}$ can refer to the aspect ratio of the placement, amount of wasted resources, or wire length of a connection to a fixed I/O pin. On the other hand, the second metric estimates the inter region wire length using the HPWL formula. HPWL considers the wire connections concentrated in the center of the regions and measures the wire length using the Manhattan distance. In conclusion, the considered *fitness* function is a linear combination of the two defined metrics and an additional parameter

Algorithm 5 GA local search

```
1: function IMPROVESOLUTION(solution)
2:     obj ←solution.evaluate()
3:     repeat
4:         oldObj ← obj
5:         for each n ∈ N do
6:             for each p ∈ P_n do
7:                 obj' ← solution.evaluatePlacement(n, p)
8:                 if obj' < obj then
9:                     solution.setPlacement(n, p)
10:                    obj ← obj'
11:    until obj < oldObj
12:    return solution
```

λ able to handle unfeasible situations:

$$obj = q_a \cdot \frac{A_{cost}}{A_{max}} + q_{wl} \cdot \frac{WL_{cost}}{WL_{max}} + \lambda \tag{6.11}$$

In the formula, A_{max} and WL_{max} represent the maximum values that A_{cost} and WL_{cost} can assume respectively; they are used to normalize the two contributions. Then, q_a and q_{wl} are user-defined weights. In particular, the fitness function first analyzes the feasibility of the solution only in terms of fulfillment of the non-overlapping condition (REQ5), and then computes the cost according to the selected metrics. If the solution is unfeasible, a penalty value λ, defined as the number of pairs of regions that overlap, is summed to the objective value. In Equation (6.11) we force $q_a + q_{wl} = 1$ so that, valid floorplans are represented by $0 < obj \leq 1$, while $obj > 1$ identifies unfeasible solutions. The goal of the λ factor is to enable the *selection* operator of the GA (we use the classical tournament selection) to rank unfeasible solutions in terms of the criticality of the constraint violation.

The choice of a simple solution encoding and operators has been driven by the possibility to directly manage the solution space, as motivated in Section 6.5. However, as a drawback, we have noted during a preliminary experimental evaluation that such GA engine is not able to obtain better performance than the preliminary MILP approach since it generates too many unfeasible solutions. In fact, the direct problem formulation leads feasible solutions to evolve in unfeasible ones with a high probability. The main cause is the crossover operator that, due to its nature, applies "global changes" to each explored solution; at the opposite the mutation operator which performs local moves, has higher possibilities to make a feasible solution to evolve to another feasible one, that is a "neighbor" in the solution space. For this reason, such engine has been enhanced with a *local*

search function, based on a steepest descent heuristic, that improves the current solution with iterated local modifications until no further improvement is possible. The strategy, shown in Algorithm 5, is invoked in the GA fitness function and guarantees to reach a local optimum from the input solution. We empirically demonstrated that the adoption of local search within GA leads to a hybrid approach able to converge faster towards global optima [107].

6.7 Experimental evaluation

The proposed floorplanning automation framework has been implemented in C++; GAlib [106] has been used for the GA engine. Within the following experimental sessions, we also integrated in the framework a Simulated Annealing (SA) engine, implemented in C++ by using the GNU Scientific library [1], and the preliminary MILP formulation [82]. For the other considered state-of-the-art approaches, the original algorithms provided by the related authors have been adopted, while all the MILP models have been solved using Gurobi 6.5.

In the first experimental session, we performed an extensive testing campaign considering a large set of synthetic circuits aimed at demonstrating that the proposed GA engine outperforms the state-of-the-art approaches, whereas in the second section we performed a more in-depth comparison of various engines (such as GA, SA and MILP) exploiting the same direct problem representation proposed in this chapter. Finally, we carried out two real case studies to show that the framework generates feasible floorplanning solutions possibly without any manual action of the designer, and, moreover, it is able to improve the system performance. The three sessions are presented in the following sections. As a final note, all the experiments have been performed on a 2.2GHz Intel Core Duo T6600 processor running a Linux operating system.

6.7.1 Comparison with respect to past approaches

The first experimental session considered the test suite of synthetic circuits from [85] targeted for the Virtex-5 XC5VLX110T device. This suite consists of 20 circuits with different area occupancy and number of reconfigurable regions; specifically there are 4 circuits having a number of reconfigurable regions in the range {5, 10, 15, 20, 25}, while with respect to area utilization there are 5 circuits for each fixed device occupancy in the range {70%, 75%, 80%, 85%}. It is worth noting that the maximum number of regions for the considered circuits has been set by taking into account

that reasonably a reconfigurable system does not feature a larger number of reconfigurable regions. Moreover, we also made comparisons on some circuits from MCNC and GSRC suites, adapted as done in [13]; more precisely, we considered `apte`, `xerox`, `hp`, `ami33` and `ami49` targeted for a more recent Virtex-7 XC7K160T device.

In this first session, we compared our GA engine (dubbed PA-GA[2]) against the most efficient state-of-the-art approaches discussed in Section 6.2, i.e., [13], the HO and O MILP-based algorithms presented in [85] and our preliminary MILP formulation [86] (dubbed as PA-MILP). Nevertheless, in order to perform the comparison it was necessary to remove requirements REQ3 and REQ4 from the placement generation process since previous approaches do not support them and their integration for [13] and [85] is not straightforward. Regarding the objective function, in this section we only considered the overall wire length since it has been noted to be the most challenging optimization goal. For each experiment we executed 10 runs of [13] and considered the best result as its final solution. According to the approach defined in [85], we run the HO approach by starting from some of the best solutions found by [13] (the ones within 10% from the best one), and subsequently O by using the final solution achieved by HO. For all the MILP formulations, the Gurobi solver execution time was limited to 1800 seconds, whereas, for PA-GA engine we used a stopping criterion based on elapsed time and the same time limit was applied. The elaboration was parallelized on all the available cores by using the *Threads* Gurobi setting and by running different processes for PA-GA with different random seeds. Notice however that the time limit does not take into account the time needed by PA-MILP and PA-GA for the generation of the feasible placements and the additional time to generate the initial solution for O [85]. Finally, PA-GA and PA-MILP used P_n^w as input.

Tables 6.2 and 6.3 show the results of this first experimental session both in terms of execution time and quality of the achieved solutions. PA-GA was always able to find equivalent or better solutions than PA-MILP that in turns provided better results than [85] and [13]. Furthermore, the highest improvements were achieved for the most challenging circuits consisting of high number of reconfigurable regions. Specifically, for the test cases having 20 and 25 regions PA-GA reduced the wire length of PA-MILP solutions by 8.2% on average while using the same amount of time. On the other hand, when considering the variation of resource usage (reported in Table 6.3), as expected, the best results for both PA-GA and PA-MILP are

[2]The label PA, standing for Proposed Approach, here identifies all the engines based on the direct problem formulation proposed in this work.

Table 6.2: *Results with different numbers of reconfigurable regions*

# RRs	Average wire length improvement w.r.t. [13]				Average execution time (sec)				
	HO [85]	O [85]	PA-MILP [82]	PA-GA	[13]	HO [85]	O [85]	PA-MILP [82]	PA-GA
5	6.99%	7.48%	7.46%	7.46%	11.0	13.3	84.1	32.5	1801.2
10	7.59%	11.65%	17.98%	19.07%	24.1	48.3	1848.3	1421.8	1801.4
15	8.88%	16.25%	34.03%	36.14%	41.1	74.6	1874.6	1802.1	1801.7
20	5.47%	14.80%	33.29%	39.17%	65.5	82.8	1882.9	1802.6	1801.9
25	5.67%	24.01%	40.72%	45.32%	94.0	119.1	1919.2	1803.0	1802.3

Table 6.3: *Results with different overall device occupancy*

Occupancy	Average wire length improvement w.r.t. [13]				Average execution time (sec)				
	HO [85]	O [85]	PA-MILP [82]	PA-GA	[13]	HO [85]	O [85]	PA-MILP [82]	PA-GA
70%	8.51%	17.40%	30.19%	32.15%	47.4	87.6	1559.0	1454.7	1801.7
75%	5.49%	18.89%	26.21%	28.74%	47.3	67.1	1514.2	1138.9	1801.7
80%	6.20%	12.86%	27.57%	30.73%	47.1	60.0	1508.1	1448.9	1801.7
85%	7.48%	10.20%	22.82%	26.10%	46.7	55.8	1505.9	1447.1	1801.7

Table 6.4: *Approaches comparison on different test cases*

Circuit	# RRs	Wire length					Execution time (sec)				
		[13]	HO [85]	O [85]	PA-MILP [82]	PA-GA	[13]	HO [85]	O [85]	PA-MILP [82]	PA-GA
apte	9	12789	12029	9682	5313	5206	21.75	147.93	1947.93	1801.87	1801.88
xerox	10	26589	25878	22974	12643	12813	25.53	307.97	2107.98	1802.36	1802.37
hp	11	12403	11796	11036	5568	5298	29.42	201.04	2001.05	1802.41	1802.45
ami33	33	172332	157583	130121	128538	104414	164.61	324.54	2124.57	1803.93	1803.98
ami49	49	55819	53178	36930	22669	18583	180.54	192.25	1992.26	1809.35	1809.47

obtained for circuits with lower resource requirements even if there is no pronounced trend.

Table 6.4 reports the results for the re-adapted MCNC and GSRC benchmarks. Results show that PA-MILP gives an improvement with respect to O that varies from 1.2% on `ami33` to 49.5% on `hp` circuits, while with respect to the comparison between PA-MILP and PA-GA, the results are aligned to the synthetic benchmark trend. Indeed, for `apte`, `xerox` and `hp` circuits both PA-MILP and PA-GA provides a similar overall wire length, whereas, when dealing with the bigger `ami33` and `ami49` circuits, PA-GA is able to improve PA-MILP solutions by 18.8% and 18.0% respectively.

It is worth noting that for the synthetic circuits having 5 reconfigurable regions (Table 6.2), PA-MILP is able to certify the optimality of the solutions and thus complete its execution before the given time limit. PA-GA, being a meta-heuristic approach, cannot state if the identified solution is optimal, and, hence, the execution time of the algorithm depends on the time budget assigned. However, in these cases, we noted that PA-GA converge faster to the optimal solution than PA-MILP, while as the problem grows above the 10 regions, no MILP formulation is able to reach the optimal solution in a reasonable time when the inter-region wire length is considered. In fact, in our tests we run the PA-MILP, i.e. the most efficient MILP formulation for several hours; however, after an initial very fast convergence to a near-optimal solution, the engine was not able to improve the solution or certify its optimality. This is related to the weak linear relaxation bounds provided by the MILP formulation with respect to inter-region wire length; the issue was only partially mitigated by using additional cuts to the model (see Appendix 6.A). As an example, we report in Figure 6.5 the graph representing the improvement of the best solution for the considered algorithms. We may note from the figure that PA-GA is the faster to evolve towards near-optimal solutions. This trend is representative for all the performed tests. Moreover, we noted that on average PA-GA and PA-MILP tends to stabilize their solutions in less than 600 seconds, while after that, no relevant improvement is reported.

As a final note, the generation of the definition of the conflict graph model had a negligible impact on the overall execution time of the proposed algorithms. Indeed, in real situations, the size of the conflict graph is manageable; as an example, for the circuit having the highest number of regions (`ami49`), the feasible placements generation process produced 146446 nodes in less than 10 seconds.

Table 6.5: *Proposed approaches comparison with different number of regions*

# RRs	Solutions improvements w.r.t. PA-MILP [82]								
	$q_a = 1.0, q_{wl} = 0.0$			$q_a = 0.5, q_{wl} = 0.5$			$q_a = 0.0, q_{wl} = 1.0$		
	PA-SA	PA-GAn	PA-GA	PA-SA	PA-GAn	PA-GA	PA-SA	PA-GAn	PA-GA
5	0.00%	-3.78%	0.00%	-0.77%	-11.95%	0.00%	-3.11%	-2.77%	0.00%
10	-4.31%	-23.02%	0.00%	-2.32%	-27.09%	0.18%	-8.59%	-29.16%	0.55%
15	-7.25%	-31.62%	-0.27%	-6.19%	-21.16%	5.28%	-12.69%	-78.73%	5.35%
20	-9.97%	-31.53%	-0.94%	-4.44%	-17.45%	7.09%	-15.29%	-30.36%	10.41%
25	-17.02%	-24.34%	-0.35%	-7.38%	-20.66%	9.42%	-3.75%	-8.78%	19.39%
30	-10.19%	-18.97%	-0.60%	-10.74%	-16.80%	8.95%	0.21%	-21.48%	21.86%

Table 6.6: *Proposed approaches comparison with varying resource usage*

Usage	Solutions improvements w.r.t. PA-MILP [82]								
	$q_a = 1.0, q_{wl} = 0.0$			$q_a = 0.5, q_{wl} = 0.5$			$q_a = 0.0, q_{wl} = 1.0$		
	PA-SA	PA-GAn	PA-GA	PA-SA	PA-GAn	PA-GA	PA-SA	PA-GAn	PA-GA
70%	-9.63%	-17.24%	0.00%	-1.02%	-13.23%	7.50%	1.44%	-15.95%	10.67%
75%	-7.72%	-24.94%	-0.32%	-3.77%	-11.13%	5.98%	-3.16%	-21.95%	9.29%
80%	-6.37%	-19.66%	-0.28%	-11.66%	-28.70%	3.21%	-17.54%	-53.37%	7.95%
85%	-8.75%	-26.81%	-0.82%	-4.76%	-25.84%	3.94%	-9.52%	-25.12%	10.49%

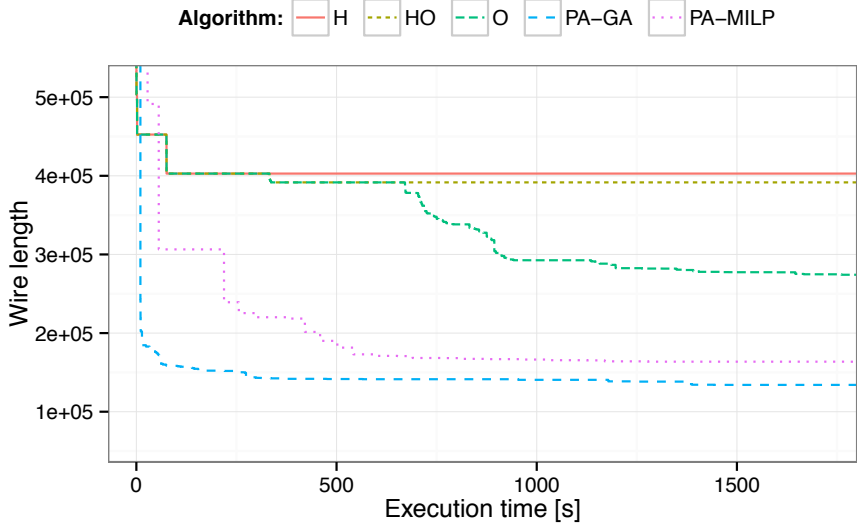


Figure 6.5: *Solution improvement over time for different approaches on the* `ami49` *test case.*

6.7.2 Analysis of engines based on the proposed representation

In a second session, we performed a more challenging comparison of PA-GA against other engines exploiting the same direct problem representations. In particular, we considered 1) PA-MILP, 2) an SA engine (called PA-SA), since it represents the classical approach for floorplanning strategies, and 3) a GA engine exploiting the same solution encoding but without any local search strategy (dubbed as PA-GAn). The SA engine was defined on the basis of the standard SA algorithm and re-using the evaluation and mutation functions of the GA engine. In this experimental session we considered PA-MILP as a baseline, since it has been the preliminary automating solution designed for the proposed framework. For this analysis, we included requirements REQ3 and REQ4, and we re-adapted the suite of the first session to target a Virtex-7 XC7V585T device; we actually modified the resource requirements of the circuits according to the size of the new device and we considered a new set of circuits with 30 regions. It is worth noting that the considered Virtex-7 device presents a higher heterogeneous distribution of resources and requires the introduction of more forbidden placements than the Virtex-5 one used in the first experimental session. Indeed, in some preliminary tests, we found that the considered state-of-the-art engines in [13, 85] failed in finding any feasible solution on this device.

Moreover, the tests have been performed considering different settings of the objective function ranging from an optimization based only on placements cost ($q_a = 1.0$, $q_{wl} = 0.0$), one considering only wire length ($q_a = 0.0$, $q_{wl} = 1.0$) and finally a mixed objective function taking into account both metrics to the same extent ($q_a = 0.5$, $q_{wl} = 0.5$). In order to perform a fair comparison of the approaches in terms of exploration efficiency, according to the discussion in Section 6.7.1 we fixed a limited time budged of 600 seconds that includes the time for the generation of the feasible placements. Except for the new time limit, the run settings for PA-MILP and PA-GA were as in the previous section, whereas, we restarted PA-SA engine several times on the available cores using different random seeds until the available time budget elapsed.

Table 6.5 and Table 6.6 compare the obtained results according to the number of regions and device usage respectively. For the problem instances consisting of 5 regions the MILP approach and the GA were both able to find the optimal solution in all cases, whereas the SA engine found optimal solutions only when the objective function was set to consider uniquely region placements cost. In general, the SA engine was almost never able to achieve better solutions than the MILP based algorithm except for some problem instances consisting of large number of regions. This result is quite interesting since SA is the commonly used approach for automating the floorplanning exploration. On the contrary, we may conclude that it is not well-suited for the defined problem representation due to the fact that many unfeasible solutions can be generated.

PA-GA proved to be an effective approach, leading to almost the same results for an optimization based on placement cost, while greatly outperforming the MILP engine when considering the most difficult problem, that is the wire length optimization. This is especially highlighted when the circuits feature a large number of regions or a low resource usage. This improvement was mainly obtained exploiting local search within GA that allows to quickly explore a solution space consisting of local optimal solutions. Indeed, as it is possible to notice from the two tables, PA-GAn provides results considerably far from its enhanced counterpart since it spends a large amount of time exploring unfeasible regions of the solution space, without the capability to recover to a feasible solution. It is worth noting that in some experiments PA-GAn was not able even to find an initial feasible solution; such situations were discarded from the results reported in the two tables. Thus, we may conclude that the GA engine is the most promising solution for the proposed floorplanning framework.

6.7.3 Case studies

Finally, we validated the proposed approach on two real case studies to be implemented on a Virtex-7 XC7V585T device. The first case study is a re-adaptation of a Xilinx sample design (*project_cpu_virtex7*), consisting of five modules connected using a star topology, whereas the second case study is an in-house design of a an image processing pipeline with reconfigurable components. As already noted in the previous section, the approaches proposed in [13] and [85] do not consider requirements REQ3 and REQ4 thus leading to potentially invalid floorplan solutions with respect to the subsequent place and route phase. Hence, we compared the results achieved by the proposed floorplanner with respect to solutions designed manually by starting from the initial placements provided by the *place pblock* feature available in Vivado. This Vivado feature provides the user with suggestions on where to place the reconfigurable regions, however the identified placements do not meet PR guidelines and require manual modifications.

Similarly to [71], the Xilinx case study has been re-adapted considering each of the five available modules as reconfigurable, thus leading to five distinct reconfigurable regions each containing a single module. Notice however that this does not represent a limitation for the evaluation of the proposed approach since the employed implementation flow is the same. The resource requirements of the reconfigurable regions are derived from the requirements of the corresponding modules (shown in Table 6.7) in which the number of LUTs was augmented by approximately 25% to ensure enough space for the insertion of proxy logic [112]. Furthermore, the number of interconnections among reconfigurable regions together with interconnections to I/O are summarized in Table 6.8.

For the exploration, the objective function was set to consider wire length and resource consumption to the same extent in order to reduce both reconfiguration overhead and improve the possibility to meet timing constraints. The floorplan solution identified by PA-GA is shown in Figure 6.6a together with the placed and routed circuit. Overall the implementation phase was successful and the timing constraint requiring a 100MHz frequency was met. On the other hand, Figure 6.6b presents the initial solution provided by Vivado *place pblock*, and Figure 6.6c the subsequent manually re-adapted floorplan. It is clearly visible that the designer has to perform a considerable and intrusive change of the solution proposed by Vivado; from our experience we may report that such activity requires around 2 hours of time, while our automated engine requires a few minutes. Moreover when considering

Table 6.7: *Resource requirements of modules from the Xilinx case study*

Module	LUTs	Registers	F7 Muxes	F8 Muxes	BRAMs	DSPs
cpuEngine	7440	3892	297	0	21	4
fttEngine	2837	1679	0	0	16	96
usbEngine0	6000	4699	259	81	36	0
usbEngine1	6080	4699	259	81	36	0
wbArbEngine	6800	1044	1959	172	0	0

Table 6.8: *Interconnection matrix of regions for the Xilinx case study*

Region	fttEngine	usbEngine0	usbEngine1	wbArbEngine	I/O
cpuEngine	1	0	0	311	0
fttEngine	-	0	0	106	69
usbEngine0	-	-	0	118	69
usbEngine1	-	-	-	118	0
wbArbEngine	-	-	-	-	0

the quality of the achieved solutions, this second one was not able to meet timing during implementation due to not optimized inter-region interconnections. By lowering the timing constraint it was possible to meet timing at 80MHz, however no place and route solution was found satisfying timing constraints with frequency equal or higher than 85MHz. Furthermore, the floorplan produced by the GA engine was able to reduce the overall size of the partial bitstreams by 25.7% with respect to the manual solution.

As a second case study, we realized a design in the context of image analysis consisting of seven different modules whose interconnections are shown in Figure 6.7. In particular, the design is composed of two main computational pipelines that operate on a gray scaled image. The first pipeline includes the *histogram*, *Otsu filter* and *threshold 1* modules, it binarizes the given image by applying the Otsu separation algorithm [77], while, the second pipeline is configured to perform edge detection by exploiting the *Gauss filter*, *Laplace filter* and *Threshold 2* modules. The modules were generated by using Vivado HLS and AXI stream interfaces were used for communication. Furthermore, we implemented different alternative versions of each module within the design (employing different algorithms or having a different trade-off between results accuracy and execution time) to exploit partial reconfiguration to switch from one to the other one; thus,

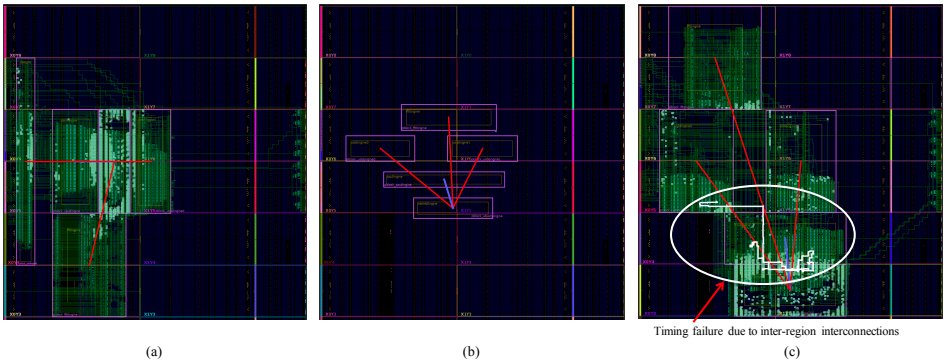


Figure 6.6: *Floorplans for the Xilinx case study: (a) place and route of PA-GA floorplan at 100MHz constraint, (b) initial floorplan obtained with Vivado place pblock, and, (c) place and route of manually adapted Vivado floorplan at 90MHz constraint.*

seven reconfigurable regions were considered (one for each component within the original design). The resource requirements for each version of the various modules are listed in Table 6.9, whereas the requirements for the corresponding reconfigurable regions were obtained as in the previous case study.

Even in this scenario, the objective function was tuned to take into account inter-region wire length, regions aspect ratio and wasted resources to the same extent. Within this case study, an analysis of the post-implementation results showed that the critical paths were represented by the internal interconnections between the computational logic and the local BRAMs of the modules, whereas, inter-region interconnections were easily routed. Due to the peculiar characteristics of this design, both the PA-GA and manually re-adapted floorplans were able to meet timing at 120MHz and failing at a frequency equal or higher than 125MHz. Moreover, the floorplan produced by the GA engine was able to reduce the overall size of partial bitstreams of the manual solution from 9695 KB to 8815 KB, hence leading to a smaller reconfiguration time. Finally, similarly to the previous case, the manual definition of the floorplan required about 2 hours of activity.

6.8 Final remarks

In this chapter, we proposed a novel floorplanning automation framework, compatible with the Xilinx tool-chain and its PR flow. The framework considers a direct problem representation, which consists in an explicit enumeration of the possible placements of each region. The defined model allows to simplify the development of efficient floorplanning algorithms

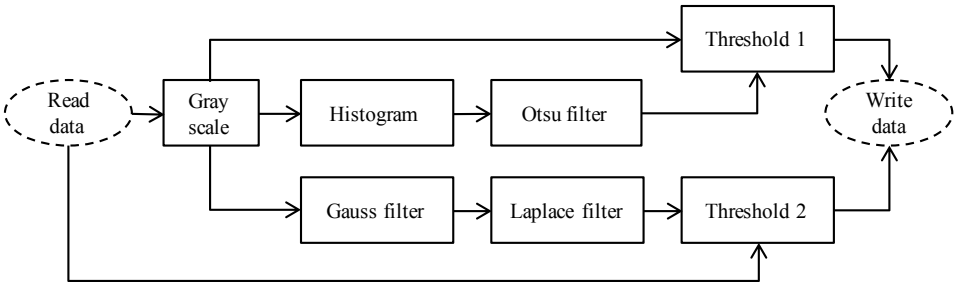


Figure 6.7: *Modules interconnections for the image processing case study.*

Table 6.9: *Resource requirements of modules for the image processing case study*

Region	Module	LUTs	FFs	BRAMs	DSPs
Laplace filter	LF_v1	628	332	64	2
Gauss filter	GF_3x3	807	465	64	0
	GF_3x3 float	881	809	32	5
	GF_5x5 float	815	760	32	5
Gray scale	GS_v1	334	238	64	4
Histogram	Hi_v1	256	180	1	0
	Hi_v2	104	87	1	0
Otsu filter	OF_v1	1205	1164	0	13
	OF_v2	726	517	0	2
Threshold 1	Th_v1	115	71	0	0
Threshold 2	Th_v1	115	71	0	0

devoted to the optimization of different metrics such as aspect ratio, inter-region wire length and resource consumption. Various engines, based on an exact MILP formulation, GA and SA heuristic approach, possibly enhanced with local search strategies, have been designed for automating the floorplanning activity. Such algorithms are experimentally evaluated with a challenging synthetic benchmark suite and real case studies. Experimental results demonstrated the effectiveness of the proposed direct problem representation and superiority of the defined GA engine with respect to the other defined strategies and the state-of-the-art approaches in terms of exploration time and identified solution.

In the context of CAD as an Adaptive Open-platform Service (CAOS), the proposed floorplanning automation framework can be leveraged by several architectural templates within the CAOS backend. Examples of such an architectural template could be inspired on the Dyplo workflow by Topic [103]. Indeed, they consider an architecture in which multiple reconfigurable regions are connected in a ring topology and the application is allowed to swap in and out different accelerator at runtime using Partial Dynamic Reconfiguration (PDR). Deciding the dimensions and placements of the reconfigurable regions, is a design space exploration task suited for our automated floorplanning framework.

6.A MILP formulation

Within this appendix we report the MILP model proposed in the preliminary version of the framework [86] and used within Section 6.7 as a baseline for algorithms comparison. The variables, sets and parameters of the formulation are listed in Table 6.10, whereas the model constraints and objective function are summarized in Table 6.11.

In the proposed model, the binary variables $x_{n,p}$ represent the current solution by stating which placement p is chosen for a given region n; when considering the conflict graph, these variables state which specific node is selected for each region. Thus, a first class of constraints, dubbed as *placements constraints*, guarantees that a given solution is feasible: in particular, exactly one placement for each region must be selected (C1), and no pairs of placements connected by an edge can be selected since they are overlapping (C2). It is worth noting that constraint C2 is defined on maximal cliques (i.e. fully-connected subgraphs) in the conflict graph instead of single edges. This allows to reduce the size of the model while, at the same time, improves the linear relaxation bounds during the MILP solving process [88], so that the overall effect is a speed-up in the MILP solver

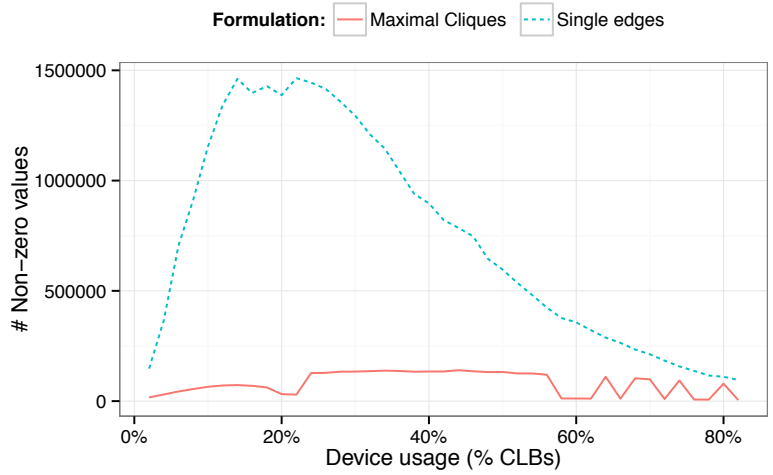


Figure 6.8: *Comparison of model size for the maximal cliques and single edges MILP formulations on a problem consisting of two regions having equal resource requirements.*

performance. Such a reduction in the problem size can be clearly seen in Figure 6.8, by analyzing the number of edges between the placements generated for two regions when varying equally their requirements in terms of CLBs; more precisely, the figure compares the number of non-zero terms within the constraint matrix of the complete MILP model using the single edges approach and the maximal cliques approach.

A second class of constraints, parameters and variables are used to compute the cost OBJ of a given solution, that is equivalent to the one used for the GA engine without the penalty contribution. Constraint C9 computes the A_{cost} metric by summing the cost $a_{p,n}$ of each selected placement, while constraint C10 computes the global HPWL. The specification of C10 requires the introduction of variables cx_n and cy_n to compute the coordinates of the centroid of each region n, and variables $dx_{n1,n2}$ and $dy_{n1,n2}$ to compute the Manhattan distance between the centroids of each couple of regions $n1$ and $n2$. Moreover, in order to guarantee the semantics of these variables constraints C3-C8 are specified; constraints C3-C4 compute the coordinates of the centroids, while constraints C5-C8 ensure that the distances among regions cannot be less than expected.

An in-depth analysis of the formulation has shown that constraints C3-C8 give weak bounds when the linear relaxation of the MILP model is solved. For this reason we introduced the additional cut C11, stating that that the centroid distance of two regions has to be at least the sum of the distances to reach the centroids of the selected placements from their near-

est borders. Indeed a wire connecting the centroids of two placements has to cross at least one border for each of them.

Table 6.10: *MILP variables, sets and parameters*

Sets	
N	set of reconfigurable regions to floorplan
P_n^w	set of width-reduced feasible placements for region n
I	set of interconnections between regions. Each element is a tuple of the form: $(n1, n2, b)$ where $n1$ and $n2$ are the regions involved in the interconnections and b is its width
Parameters	
W	maximum value on the horizontal direction
H	maximum value on the vertical direction
$tileW$	the width of a tile within the FPGA
$tileH$	the height of a tile within the FPGA
$a_{n,p}$	cost associated to placement $p \in P_n^w$ for region $n \in N$
q_a	weight associated to the area cost
q_{wl}	weight associated to the wire length cost
A_{max}	maximum cost due to placements selection
WL_{max}	maximum cost related to global inter-region wire length
Variables	
$x_{n,p}$	binary variable set to 1 if and only if the placement $p \in P_n^w$ is selected for region $n \in N$
cx_n	x coordinate of region $n \in N$ centroid
cy_n	y coordinate of region $n \in N$ centroid
$dx_{n1,n2}$	horizontal distance between centroids of regions $n1, n2 \in N$
$dy_{n1,n2}$	vertical distance between centroids of regions $n1, n2 \in N$
A_{cost}	floorplan cost due to placements selection
WL_{cost}	floorplan cost related to global inter-region wire length

Table 6.11: *MILP model constraints and objective function.*

Placements constraints	
C1	$\sum_{p \in P_n^w} x_{n,p} = 1, \forall n \in N$
C2	$\sum_{n \in N, p \in P_n^w : p \perp (xp,yp,1,1)} x_{n,p} \leq 1,$ $\forall xp \in [0, W-1], yp \in [0, H-1]$
Wire length semantics	
C3	$cx_n = \sum_{p=(xp,yp,wp,hp) \in P_n^w} x_{n,p} \cdot (xp + wp/2), \forall n \in N$
C4	$cy_n = \sum_{p=(xp,yp,wp,hp) \in P_n^w} x_{n,p} \cdot (yp + hp/2), \forall n \in N$
C5	$dx_{n1,n2} \geq cx_{n1} - cx_{n2}, \forall n1, n2 \in N \mid n1 \neq n2$
C6	$dx_{n1,n2} \geq cx_{n2} - cx_{n1}, \forall n1, n2 \in N \mid n1 \neq n2$
C7	$dy_{n1,n2} \geq cy_{n1} - cy_{n2}, \forall n1, n2 \in N \mid n1 \neq n2$
C8	$dy_{n1,n2} \geq cy_{n2} - cy_{n1}, \forall n1, n2 \in N \mid n1 \neq n2$
Cost functions definition	
C9	$A_{cost} = \sum_{n \in N, p \in P_n^w} a_{n,p} \cdot x_{n,p}$
C10	$WL_{cost} = \sum_{(n1,n2,b) \in I} (dx_{n1,n2} \cdot tileW + dy_{n1,n2} \cdot tileH) \cdot b$
Additional cuts	
C11	$dx_{n1,n2} + dy_{n1,n2} \geq$ $\sum_{p=(xp,yp,wp,hp) \in P_{n1}^w} x_{n1,p} \cdot min\{wp/2, hp/2\} +$ $\sum_{p=(xp,yp,wp,hp) \in P_{n2}^w} x_{n2,p} \cdot min\{wp/2, hp/2\},$ $\forall n1, n2 \in N \mid n1 \neq n2$
Objective function	
OBJ	$\min \left\{ q_a \cdot \frac{A_{cost}}{A_{max}} + q_{wl} \cdot \frac{WL_{cost}}{WL_{max}} \right\}$

CHAPTER 7

CAOS backend: mapping and scheduling for partially-reconfigurable designs

The usage of Partial Dynamic Reconfiguration (PDR) broadens the set of design choices and trade-offs to consider for the implementation of accelerated functions on Field Programmable Gate Array (FPGA). Indeed, while PDR allows to overcome the limited availability of FPGA resources with granular time-multiplexing, it also introduces reconfiguration overhead, that, if not carefully considered and hidden, might jeopardize performance. In this chapter we present and evaluate an approach to map and schedule a Directed Acyclic Graph (DAG) of compute tasks onto reconfigurable regions on a target FPGA. The proposed algorithm exploits the notion of resource efficient task implementations in order to reduce the overhead incurred by PDR and increase the number of concurrent tasks that can be hosted on the reconfigurable logic as hardware accelerators. The mapping and scheduling techniques discussed in this Chapter as well as the floorplanning algorithm presented in Chapter 6 are two key steps for adding partial-reconfiguration support within the CAOS backend described in Chapter 2.

7.1 Introduction

In the last few years, System-on-Chip (SoC) architectures consisting of processor cores tightly coupled with reconfigurable logic have become popular [113]. These SoCs allow the designer to realize complex applications in which hardware software co-design solutions are required to achieve the best performance. Furthermore, current FPGAs allow to reconfigure portion of the device dynamically while components not affected by the reconfiguration process can still operate [112]. This feature, dubbed as PDR, allows to virtually increase the resource availability of the FPGA by changing at runtime the set of configured hardware modules. To enable PDR, the reconfigurable logic is partitioned into several *reconfigurable regions* each being able to host different hardware accelerators over time. Each reconfigurable region must be defined large enough to satisfy the resource requirements of the hardware modules that are assigned to it, while the floorplanning of the regions on the FPGA must comply with PDR constraints [86]. The reconfiguration process is then performed by means of a dedicated component, such as the Internal Configuration Access Port (ICAP) available on Xilinx devices, that is exploited to load the partial bitstream for the reconfigurable region to the FPGA configuration memory.

Even though PDR provides a great flexibility for designing the system, the overhead incurred during the reconfiguration process must be carefully taken into account since it can easily jeopardize the performance gain achieved by hardware acceleration [15]. The reconfiguration time is proportional to the amount of resources required by the reconfigurable region in which the new hardware accelerator has to be allocated, it is thus crucial to minimize unused resources across region configurations and mask the reconfiguration time whenever possible.

An additional degree of freedom given to the designer is the selection of the implementation for an application task. Indeed, a task can be either implemented in software and run on the available processor cores, or implemented as a hardware accelerator. Furthermore, multiple hardware implementations with different resource requirements and execution time might also be available. As an example, the designer can provide a software implementation for a given task and leverage High-Level Synthesis (HLS) tools to generate multiple hardware implementations by setting different loop unrolling factors to trade off task performance against resource requirements.

Overall, the resulting task scheduling problem is NP-hard as it represents a more general optimization version of the Resource Constrained

Scheduling Problem (RCSP) that is NP-complete [38]. Thus efficient scheduling heuristics are required to both quickly evaluate the potential performance achievable by the application on a given architecture and provide optimized solutions. In this work, we propose an offline scheduling technique for application taskgraphs on SoCs featuring processor cores and a partial dynamic reconfigurable FPGA. The proposed approach leverages the floorplanning algorithm presented in Chapter 6 to ensure that the final solution admits a feasible floorplan. Within this context, we may summarize our contribution as follows:

- we introduce the notion of resource-efficiency as a mean to guide the scheduler in the generation of a suitable set of reconfigurable regions;
- we present a fast deterministic scheduling heuristic for the optimization of the application execution time;
- we propose a randomized version of the scheduling heuristic that can be exploited to trade off algorithm execution time and quality of the final solution;
- finally, the effectiveness of the proposed algorithms are evaluated on a large set of synthetic benchmark and compared with state-of-the-art approaches.

The remainder of the chapter is organized as follows: Section 7.2 shows the related work in the literature, Section 7.3 gives a formal description of the problem, Section 7.4 presents a high level view of the proposed approach, Section 7.5 discusses the implementation details of the deterministic scheduler, Section 7.6 shows how we derived the randomized version of the algorithm, Section 7.7 evaluates our approach on different problem instances and, finally, Section 7.8 draws the conclusions.

7.2 Related work

In the literature, several approaches that address the task scheduling problem on reconfigurable architectures have been proposed ([8,9,15,23,34,40, 53, 61, 89]). However, only few of them explicitly take into account PDR and consider reconfigurations as separate tasks performed by a dedicated component ([8, 9, 15, 23, 89]).

In [34], the authors propose an exact algorithm for the scheduling and mapping of tasks having a single hardware implementation on FPGAs with

homogeneous resources. However, the approach does not consider contention on the reconfigurator, so that an unlimited number of reconfigurations can be performed concurrently. The same limitation still holds for the optimal algorithm described in [40] where reconfigurations are not explicitly handled as separate tasks. Furthermore the approach considers reconfigurable regions having equal dimensions, thus limiting the size of the solution space and leading to potential suboptimal results. Differently from [34] and [40], [61] gives more flexibility in terms of the type of available implementations for the application tasks. In [61], the authors present a hybrid approach based on genetic algorithm and list based scheduling that allows to partition the tasks of the applications either in software or on the FPGA. However, the bottleneck due to the single reconfigurator is still not taken into account. The work developed in [53] generalizes the partitioning of tasks to an arbitrary set of processing elements and adopt clustering techniques to reduce the communication overhead across different components, nevertheless the algorithm does not consider PDR for FPGAs components.

Among the approaches that take into account PDR and contention on the reconfiguration controller, [9] is worth mentioning. The authors in [9] propose the PARLGRAN heuristic, an improved solution with respect to their previous work [8] for the scheduling and placement of tasks considering physical constraints. The approach uses anti fragmentation techniques for the definition of the areas on the FPGA and exploits time slacks within the reconfigurator to mask the reconfiguration overhead. However, the applicability of the algorithm is limited to applications whose tasks have a single hardware implementation. In [89], the authors present an Integer Linear Programming (ILP) formulation that allows to perform scheduling, mapping and define the reconfigurable regions for the tasks assigned to the FPGA. The model considers the reconfigurations as separate tasks and enables schedules in which the configuration of a task does not need to precede immediately its execution. The latter strategy is dubbed as *reconfiguration prefetching* and allows to improve the final solution thanks to the added flexibility for the reconfigurations scheduling. The formulation also consider *module reuse*, that is the possibility to avoid reconfiguration among subsequent tasks that have the same implementation and are scheduled in the same reconfigurable region. Furthermore, [89] generalizes the task scheduling problem by giving the possibility to consider multiple reconfiguration controllers. Nevertheless, the resulting complexity of the ILP formulation makes the approach not viable even for small problem instances.

Both the approaches presented in [15] and [23] explicitly consider par-

tial dynamic reconfiguration as a mean to optimize the execution time of the application. In [15], the authors tackle the scheduling and the mapping of tasks on the architectural components separately. Ant Colony Optimization (ACO) is used to explore the solution space in terms of tasks mapping, while, at each iteration of the metaheuristic a list-based scheduler is invoked to search for an optimized schedule given a fixed mapping. The separation of the two phases allows to manage the complexity of the problem but, at the same time, increases the probability to find suboptimal solutions. On the other hand, in [23] the authors propose a Mixed-Integer Linear Programming (MILP)-based algorithm that tackle simultaneously the mapping of tasks on the architecture and the scheduling of their execution. Even though the approach could potentially be used to search for the optimal solution to the problem, the execution time of the algorithm grows exponentially with the number of tasks, thus making it impractical for large problem instances. To overcome this difficulty the authors propose IS-k, an iterative approach in which k tasks at a time are optimally scheduled exploiting the MILP model. Even though IS-k showed to achieve better results than [15], the iterative scheme of the algorithm makes it vulnerable to initial wrong decisions as demonstrated in Section 4.4.6.

Within this work we propose two novel scheduling heuristics that are able to improve the quality of the results achieved by [23] in terms of running time and schedule execution time. The approaches take into account PDR, the reconfiguration overhead caused by contention on the reconfiguration controller and allows the designer to provide a set of different implementations for every single task. Furthermore, we exploit the MILP-based floorplanning algorithm described in Chapter 6 in order to identify the constraints for the reconfigurable regions and to validate the final solution.

7.3 Problem description

The task scheduling problem addressed in this work is similar to one presented in [23] and [15]. Given a description of the target architecture in terms of processor cores and the resource availability of the reconfigurable logic, the goal is to schedule the applications tasks either in Hardware (HW) or Software (SW) trying to minimize the overall execution time while possibly exploiting PDR. More formally, the architecture description is given by the following parameters:

$P :=$ set of available processor cores;

$R :=$ set of reconfigurable resources on the FPGA (e.g.: CLB, BRAM, DSP, ...);

$recFreq :=$ the amount of bitstream that can be reconfigured in a second on the reconfigurable logic;

$maxRes_r :=$ number of resources of type $r \in R$ available on the FPGA;

Notice that within the architecture we consider a single reconfiguration component, thus no two separate reconfigurations can occur at the same time due to contention.

On the other hand, the application is given in terms of a taskgraph, that is a DAG $G = (T, E)$ in which a node $t \in T$ represents a specific application task and an arc $(t_1, t_2) \in E$ denotes data dependency between tasks $t1$ and $t2$. Furthermore, each task can have several hardware and software implementations whose details are given by the following parameters:

$I_t^H :=$ set of available HW implementations for task $t \in T$;

$I_t^S :=$ set of available SW implementations for task $t \in T$;

$I_t :=$ set of all implementations for task $t \in T$ ($I_t^H \cup I_t^S$);

$time_i :=$ execution time of implementation $i \in I_t$;

$res_{i,r} :=$ resources of type $r \in R$ required by hardware implementation $i \in I_t^H$;

The communication overhead among different tasks is not explicitly handled by the adopted problem representation, however the time needed to read and write data for a given implementation can be included within its execution time. Furthermore, we assume that at least a software implementation is available for each task.

The output of the scheduling algorithm consists of: (1) the set of reconfigurable regions together with their resource requirements, (2) a mapping function that assign each task to a specific implementation and to a processor core or a reconfigurable region, (3) the time slot allocated for each task, and, (4) the set of required reconfigurations together with their time slots. Furthermore, the scheduler ensures that the dependencies among different tasks are respected, guarantees that reconfigurations are performed between the execution of tasks assigned to the same reconfigurable region and ensures that the resource required by the reconfigurable regions do not exceed the available FPGA resources.

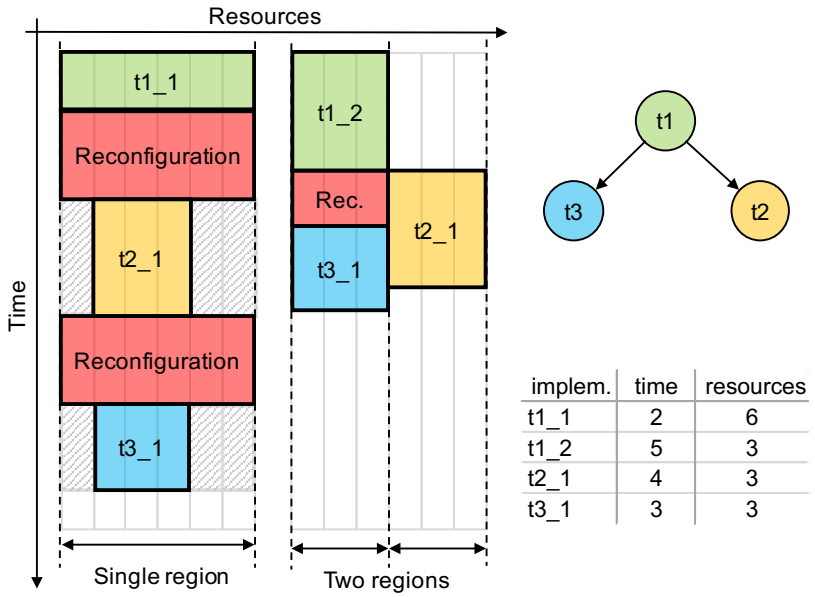


Figure 7.1: *Impact of implementation selection on schedule execution time.*

7.4 Proposed approach

A critical step of the task scheduling problem is the selection of the implementation to be used for a specific task. As an example, in Figure 7.1 we show two possible schedules for a simple application consisting of tasks $T = \{t1, t2, t3\}$ where task $t1$ has two available hardware implementations ($t1\_1$ and $t1\_2$), while tasks $t2$ and $t3$ have a single hardware implementation. Software implementations are not reported, but we assume that their execution time is high enough so that, when selected, they always worsen the overall schedule. The figure also shows the data dependencies between the three tasks and, for the sake of simplicity, considers a single type of FPGA resource. The available implementations for task $t1$ offer a trade-off between execution time and resource requirements as it generally happens for real hardware implementations. As we can see form the schedule on the left, the selection of implementation $t1\_1$ provides the best execution time for task $t1$ but leads to the generation of a single large reconfigurable region. This choice is not efficient in terms of resource utilization and worsen the overall execution time in three ways: (1) limits parallelism since a single task at a time can run on a reconfigurable region, (2) increases the number of required reconfigurations and, (3) leads to higher reconfiguration times since a larger bitstream is required for the region reconfiguration. On the

other hand, the selection of $t1\_2$ locally worsen the execution time for task $t1$ but eventually improves the schedule execution time, as shown in the schedule on the right.

Iterative algorithms such as IS-k perform subsequent greedy optimizations of the schedule considering k tasks at a time. In the extreme case of IS-1, the implementation $t1\_1$ would be selected, thus increasing the overall application execution time. In order to overcome this issue, we propose a deterministic scheduling heuristic that: (1) starts by considering implementations having a suitable trade-off in terms of resource requirements and execution time, (2) assigns the hardware tasks on the reconfigurable logic giving priority to tasks with resource-efficient implementations, (3) creates an initial optimal but generally not feasible schedule and, (4) modifies the initial schedule to achieve a feasible solution by increasing as least as possible the overall execution time. With resource-efficient task implementation we mean those hardware implementations that have a high ratio between execution time and required resources. Such implementations tend to have a high execution time with low area usage and, as shown in Figure 7.1, they allow to distribute more evenly the load on the reconfigurable logic. In addition, we also propose a randomized variant of the scheduler that relaxes the fixed priority based on resource-efficiency and allows to explore a larger solution space.

Both the proposed algorithms are able to generate directly the set of reconfigurable regions S needed by the hardware implementations and the resource requirements $res_{s,r}$ for each region $s \in S$ and resource type $r \in R$. Such information are also exploited internally by the scheduler to estimate the bitstream size for a reconfigurable region $s \in S$ by means of the following equation:

$$bit_s = \sum_{r \in R} res_{s,r} \cdot bit_r \tag{7.1}$$

where bit_r represents the average number of bits required to configure a single resource of type r whose value is derived form the number of configuration frames and resources available in a FPGA tile of type r [105]. Starting from the bitstream size, the estimated reconfiguration time for region $s \in S$ is simply computed as:

$$reconf_s = \frac{bit_s}{recFreq} \tag{7.2}$$

Even though the schedulers guarantee that the resources required by different reconfigurable regions do not exceed the amount of resources available on the FPGA, it is still necessary to verify whether the set of regions

admits a floorplan that complies with the PDR constraints [112]. Similarly to [23] a floorplanning algorithm is executed after the scheduling phase to search for a feasible floorplan. If a solution is not found, the scheduler is re-executed by virtually reducing the available FPGA resources $maxRes_r$ by a constant factor.

7.5 Implementation details

The proposed deterministic scheduling heuristic can be subdivided in a sequence of eight different steps. The first one, dubbed as *implementation selection*, assigns each task to an initial hardware or software implementation based on a cost metric, then, during the subsequent *critical path extraction* step an initial schedule is generated assuming unlimited resources. The third step, named *regions definition*, re-evaluates the initial schedule and assigns each task requiring a hardware implementation to a reconfigurable region by leveraging an efficiency index. After having defined the reconfigurable regions, a post processing step called *software task balancing* tries to improve the schedule by switching tasks implementation from hardware to software. Once the implementations choices have been fixed, step five performs a *start and end time computation* for each task. Then, step *software task mapping* assigns the software tasks to the available processors and step *reconfiguration scheduling* concludes the generation of the schedule by defining the reconfiguration tasks and ensuring that no two reconfigurations overlap in time. Finally, the *feasibility check* step invokes a floorplanning algorithm to verify that the final solution is feasible. The details of the algorithm steps are discussed in the following subsections.

7.5.1 Implementation selection

Within this phase, the scheduler identifies and selects the most suitable implementation for each task of the application. As a first step, for every task $t \in T$ a cost is associated to each hardware implementation $i \in I_t^H$:

$$cost_i = \frac{\sum_{r \in R} weightRes_r \cdot res_{i,r}}{\sum_{r' \in R} weightRes_{r'} \cdot maxRes_{r'}} + \frac{time_i}{maxT} \tag{7.3}$$

where:

$$\begin{aligned} weightRes_r &= 1 - \frac{maxRes_r}{\sum_{r' \in R} maxRes_{r'}} \\ maxT &= \sum_{t \in T} (\min_{i \in I_t} time_i) \end{aligned} \tag{7.4}$$

As shown in equation 7.3, the cost of an implementation takes into account the relative amount of resource required on the FPGA, and, the needed execution time normalized with respect to the schedule in which all the tasks are executed in series using the lowest execution time. This allows to assign a higher cost to those implementations that either use a relatively large number of resources or execution time. Notice also that higher importance is given to resources that are more scarce on the device. After having assigned the implementation costs, the algorithm identifies the HW implementation $i_H \in I_t^H$ having the lowest cost and the software implementation $i_S \in I_t^S$ having the lowest execution time. Finally the implementation with the lowest execution time among the SW and HW implementations i_S and i_H is assigned to task t. In the next steps we refer to hardware and software tasks depending on the type of implementation selected.

7.5.2 Critical path extraction

Once every task $t \in T$ has been assigned an implementation $i \in I_t$, and consequently, an execution time $T_{EXE_t} = time_i$, the taskgraph topological order is computed and its critical path is extracted using the Critical Path Method (CPM). Every task belonging to the critical path is categorized as critical while all the other tasks are labeled as non critical tasks. The CPM generates a time interval for every task $t \in T$ between a minimum and a maximum time instant defined as a time window $w_t = [T_{MIN_t}, T_{MAX_t}]$, where T_{MIN_t} represents the minimum time at which task t can start its execution, while T_{MAX_t} is the last time instant at which the task execution can be completed without propagating delays to the overall schedule. In the following steps, whenever a task is assigned a different implementation, the time windows are recomputed with respect to the current tasks dependencies.

7.5.3 Regions definition

The goal of this phase is to define the set of reconfigurable regions S, specify the resource requirements $res_{s,r}$ for each region $s \in S$ and resource $r \in R$ and assign each hardware task to one of the defined regions. As a preprocessing step, the efficiency index for the each task t for which a hardware implementation i has been selected, is computed as follows:

$$eff_i = \frac{time_i}{\sum_{r \in R}(res_{i,r} * weightRes_r)} \tag{7.5}$$

Then, the phase starts by setting $S = \emptyset$ and subsequently loops through

the hardware tasks trying to assign them to already defined regions, or generating new ones if needed. The order in which hardware tasks are processed determines the set of the reconfigurable regions that are generated and, as a consequence, greatly impacts the quality of the final schedule as discussed in Section 4.4.6. In order to reduce the probability to incur in schedule delays, the region assignment is performed for critical tasks first. Among the set of critical and non critical tasks, precedence is given to those having a hardware implementation with higher efficiency index. This choice tends to increase the number of reconfigurable regions that can be defined on the FPGA giving the possibility to better exploit hardware parallelism.

The region assignment for a critical task t having a hardware implementation i is done as follows:

1. We consider the set of regions $S_t \subseteq S$ such that $s \in S_t$ has the following properties: s has enough resources to host t, the time windows of tasks already assigned to s do not overlap with w_t and with the time window of the reconfiguration required to host t. If $S_t \neq \emptyset$, then task t is assigned the region $s \in S_t$ having the lowest bitstream bit_s.

2. else, if there are enough available resources on the FPGA, a new region s with resource requirements $res_{s,r} = res_{i,r}$ is added to S and assigned to task t.

3. Otherwise, the implementation of task t is switched to the fastest SW one and the time windows are updated.

Area assignment for a non critical task t with hardware implementation i follows a similar procedure, however the goal is shifted towards trying to maximize the FPGA utilization:

1. If there are enough available resources on the FPGA to host implementation i, a new region $s \in S'_t$ with resource requirements $res_{s,r} = res_{i,r}$ is added to S and assigned to task t.

2. else, we consider the set of regions $S'_t \subseteq S$ such that s has enough resources to host t and the time windows of tasks already assigned to s do not overlap with w_t. If $S'_t \neq \emptyset$, then task t is assigned to the region $s \in S'_t$ having the lowest bitstream bit_s.

3. Otherwise, the implementation of task t is switched to the fastest SW one and the time windows are updated.

Notice that in the previous step, whenever a task is assigned to an existing reconfigurable region, new dependencies are inserted into the taskgraphs to guarantee the ordering of tasks inside each reconfigurable region.

7.5.4 Software task balancing

Within the previous phase some of the task implementations could have been switched to SW and, as a consequence, the overall solution could have worsened since SW implementations tend to have a higher execution time than the HW ones. This change in the implementation choice generally produces a solution in which HW tasks are blocked for a long time waiting for the completion of other SW tasks. In order to exploit the underutilized resources on the FPGA, it is possible to check if some SW tasks can be executed in hardware during the unused time interval. The SW task balancing procedure is applied to all the SW tasks $t \in T \mid I_t^H \neq \emptyset$ starting from the ones having lower T_{MIN_t} as follows:

1. An estimation of the total amount of time spent for reconfigurations is calculated using the equation:
$$totRecTime = \sum_{s \in S} reconf_s \cdot (|T_s| - 1) \tag{7.6}$$
where T_s represents the set of hardware tasks assigned to the reconfigurable region s.
2. If $T_{MIN_t} > totRecTime$ and exists a region $s \in S$ whose tasks time windows do not overlap with w_t, then task t is assigned to region s using the hardware implementation with the lowest cost and the time windows are recomputed.

In the above procedure, the total reconfiguration time is used to verify that the task implementation can be moved to hardware without generating contention on the reconfigurator.

7.5.5 Start and end time computation

After having fixed all the implementations and assigned the hardware tasks to reconfigurable regions on the FPGA, this phase computes the start and end time for each task $t \in T$ as follows:

$$\begin{aligned} T_{START_t} &= T_{MIN_t} \\ T_{END_t} &= T_{START_t} + T_{EXE_t} \end{aligned} \tag{7.7}$$

Notice that the execution time T_{EXE_t} is known from the implementation selected in the previous phases.

7.5.6 Software task mapping

The aim of this step is to bind each SW task to one of the available processors in P. The main idea is to consider the software tasks in chronological order and assign them to the processor in which the minimum delay is generated. For each task $t \in T$ the following procedure is applied starting from tasks having the lower T_{MIN_t}:

1. For every processor $p \in P$ the potential delay λ_p is computed as:

$$\lambda_p = min\{0,\ max_{t2 \in T_p}(T_{END_{t2}} - T_{MIN_t})\}, \tag{7.8}$$

   where T_p is the set of current tasks assigned to p.

2. The task is assigned to the processor $p \in P$ having the lowest delay λ_p and new dependencies are added to ensure task ordering within the processor.

3. The task start and end time are recomputed as:

$$\begin{aligned} T_{START_t} &= T_{MIN_t} + \lambda_p \\ T_{END_t} &= T_{START_t} + T_{EXE_t} \end{aligned} \tag{7.9}$$

4. if $T_{END_t} > T_{MAX_t}$ the delay $T_{END_t} - T_{MAX_t}$ is propagated over the taskgraph.

7.5.7 Reconfigurations scheduling

The last step required to obtain a complete application schedule is to generate the set of reconfiguration tasks RT for the regions hosting multiple implementations. A reconfiguration occurs between each couple of subsequent tasks $t_{in}, t_{out} \in T$ assigned to the same reconfigurable region and it is needed to load the partial bitstream for task t_{out}. Tasks t_{in} and t_{out} are dubbed as ingoing and outgoing task respectively, if a reconfiguration has an outgoing critical task, then the reconfiguration task is also considered critical. Similarly to the application tasks, each reconfiguration $t \in RT$ has a time window in which it has to be executed to avoid the generation of delay:

$$\begin{aligned} T_{MIN_t} &= T_{END_{t_{in}}} \\ T_{MAX_t} &= T_{START_{t_{out}}} \end{aligned} \tag{7.10}$$

On the other hand, the execution time of the reconfiguration t depends on the region $s \in S$ in which the ingoing and outgoing tasks are scheduled:

$$T_{EXE_t} = reconf_s \tag{7.11}$$

Critical reconfigurations are scheduled before non critical ones since their delays would be completely propagated on the schedule. For each critical reconfiguration $t \in RT$, the following procedure is applied starting from the reconfiguration having lower T_{MIN_t}:

1. The time window for the reconfiguration task is recomputed since some delays may have modified the time windows of its ingoing and and outgoing tasks.

2. The last scheduled reconfiguration task tl on the reconfiguration component is extracted.

3. If $T_{MIN_t} > T_{END_{tl}}$, then the start time of the reconfiguration is set as $T_{START_t} = T_{MIN_t}$ otherwise the reconfiguration start time is computed as $T_{START_t} = T_{END_{tl}} + 1$.

4. The end time for the reconfiguration is set as usual as $T_{END_t} = T_{START_t} + T_{EXE_t}$

5. Check if the reconfiguration task generates a delay (i.e. $T_{END_t} > T_{MAX_t}$) and, if so, propagate the delay over the taskgrpah.

The scheduling of non critical reconfiguration task is more complex since an existing partial scheduling is already present and we must ensure that no two reconfigurations overlap in time. For each non critical reconfiguration $t \in RT$, the following procedure is applied starting from the reconfigurations having lower T_{MIN_t}:

1. If the time instant T_{MIN_t} lies within the execution of an already scheduled reconfiguration, T_{MIN_t} is set to the first time instead ahead in time in which no other reconfigurations are performed.

2. The start time and end time of the reconfiguration are computed as $T_{START_t} = T_{MIN_t}$ and $T_{END_t} = T_{START_t} + T_{EXE_t}$.

3. If $T_{END_t} > T_{MAX_t}$, the delay of the reconfiguration is propagated to the outgoing task.

4. Finally, if the reconfiguration overlap with the execution of other reconfigurations, those are shifted ahead in time and their delay is propagated over the taskgraph.

7.5.8 Feasibility check

At this stage, a schedule and mapping of all the tasks on the target architecture has been performed, however, there is no guarantee that the identified reconfigurable regions admit a valid floorplan on the FPGA. For this purpose, the set of regions S together with the resource requirements $res_{s,r}$ are given has input to the MILP-based floorplanning algorithm presented in Chapter 6. It is worth noting that no objective function is specified for the floorplanner since we are interested in verifying the existence of a solution in a small amount of time. If it is not possible to find a feasible floorplan, the scheduling algorithm is restarted by virtually reducing the number of resources available on the FPGA by a constant factor.

7.6 Randomized scheduler variant

The order in which non critical tasks are considered during the reconfigurable region definition presented in Section 7.5.3 leads in general to suboptimal schedules, since it does not take into account the impact of tasks dependencies, reconfiguration constraints and the possibility that some task implementations can be switched to SW. However, finding the best ordering is a computationally expensive processes, while sorting the tasks with respect to the efficiency index of theirs selected implementations provides good quality results in a small amount of time. In this section we relax the fixed processing ordering for non critical hardware tasks and we propose a variant of the algorithm that exploits a randomized order. This gives the possibility to explore a larger solution space by executing the scheduler several times, so that the designer can trade off the quality of the solution against the execution time of the algorithm. Furthermore, this randomized approach allows to amortize the computational cost of the floorplanner over different scheduling iterations. A high level view of the scheduler variant is given by Algorithm 6.

The $doSchedule$ function represents the core of the algorithm described in Section 7.5 devoid from the feasibility check phase, where an additional parameter has been added to specify the type of ordering for non critical hardware tasks during the definition of the reconfigurable regions. On the other hand, the $checkFloorplan$ method is used to query the floorplanner and verify that the current schedule is valid. Overall, the algorithm takes as input the problem instance describing the target architecture together with the application taskgraph and a time budget, while it produces as output the best schedule found. As shown in Algorithm 6 the floorplanner is executed

Algorithm 6 Randomized task scheduling algorithm

```
1:  function RSCHEDULER(instance, timeToRun)
2:      deadline = getCurrentTime() + timeToRun
3:      bestSchedule = null
4:      bestExeTime = +∞
5:      tasksOrder = "RANDOM"
6:      while deadline ≥ getCurrentTime() do
7:          schedule = doSchedule(instance, tasksOrder)
8:          if schedule.exeTime < bestExeTime then
9:              feasible = checkFloorplan(solution)
10:             if feasible then
11:                 bestSchedule = schedule
12:                 bestExeTime = schedule.exeTime
13:     return bestSchedule
```

only when a potential improving schedule is found so that the overall time spent in validating the solution on the FPGA is reduced. Furthermore, differently from the deterministic approach, unfeasible solutions in terms of the resulting floorplan are simply discarded without the need to restart the scheduler reducing the amount of available FPGA resources.

7.7 Experimental evaluation

Within this section we evaluate the proposed scheduling algorithm with task ordering based on efficiency index (PA), and, the scheduler variant that exploits randomized tasks ordering (PA-R). The achieved results are also compared to the ones found by IS-k [23], since it is the approach that is more close to our context, being able to exploits PDR and giving the possibility to trade off the execution time of the algorithm against the quality of the solutions. Specifically, the comparisons are performed against IS-1 and IS-5, where IS-1 represents the version of IS-k that has the lowest execution time for small to medium taskgraphs, while IS-5 is able to give better results at the cost of a generally higher but still acceptable execution time.

7.7.1 Testing Environment

The evaluations were performed on a test suite consisting of 100 pseudo-random taskgraphs that are organized in 10 groups containing 10 taskgraphs each. The taskgraphs within a group have a fixed number of tasks that varies in the range $[10, 100]$ across different groups. Each task has one software implementation and 3 hardware implementations with heterogeneous resource requirements (i.e. different requirements of CLBs, DSPs

Table 7.1: *Execution time of the scheduling and floorplanning algorithms.*

# Tasks	PA [s]			IS-1 [s]	PA-R / IS-5 [s]
	scheduling	floorplanning	total		
10	0.070	0.332	0.402	1.211	4.734
20	0.097	0.526	0.623	3.286	68.387
30	0.118	0.979	1.097	13.628	90.304
40	0.139	1.074	1.213	25.786	149.460
50	0.161	1.028	1.189	57.215	135.362
60	0.180	1.005	1.185	120.131	189.140
70	0.197	1.091	1.288	256.967	413.137
80	0.216	1.166	1.382	276.271	288.639
90	0.236	0.981	1.217	328.214	288.528
100	0.276	1.041	1.317	564.855	563.129

and BRAMs). Moreover, we considered that different tasks can share a common implementation so that module reuse can be exploited by IS-k, a feature currently not supported by the proposed approach. The target architectures for the applications is the ZedBoard with Zynq^{tm}-7000 All programmable SoC that provides a dual core ARM Cortex-A9 CPU and a Xilinx XC7Z020 FPGA.

All the tests were run on a Intel Core i7 4700MQ under Linux, while we used Gurobi 6.5 [44] for solving the MILP models for the floorplanner described in Chapter 6 and IS-k [23]. PA, IS-1 and IS-5 were executed until completion, while PA-R was assigned a time budget equal to the time used by IS-5 in order to have a fair comparison in terms of computational efficiency between the two approaches.

7.7.2 Results analysis

Figure 7.2 shows the average schedule execution time for the solutions found by the proposed algorithms and by IS-k with respect to each group of taskgraphs, whereas Table 7.1 reports the execution time of the different algorithms, in which the elaboration time of PA has also been subdivided in time spent during the the scheduling and the floorplanning phase. In the following discussion we focus our attention on the comparison between PA and IS-k in terms of both algorithms running time and quality of the achieved results and between PA-R and IS-5 in terms of computational efficiency.

As we can see from Table 7.1, the time required by PA to compute the schedule grows almost linearly with respect to the number of application

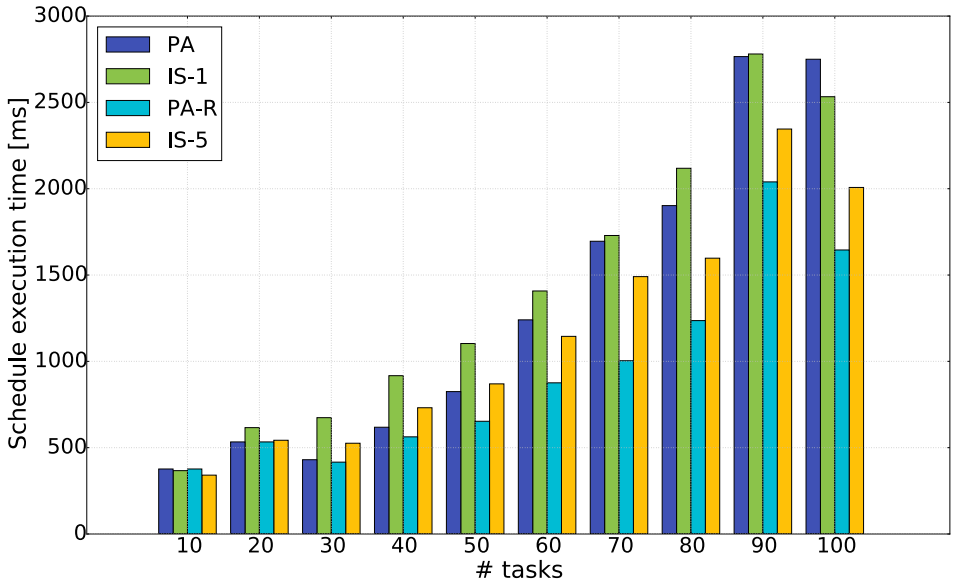


Figure 7.2: *Comparison between solutions*

tasks, while the overall PA execution time, that includes also floorplanning, is two order of magnitudes smaller than the one of IS-1 and IS-5 for taskgraphs consisting of 60 or more tasks. In order to better analyze the difference in terms of quality of the solutions, we report in Figure 7.3 and Figure 7.4 the average improvements of the schedules execution times achieved by PA using IS-1 and IS-5 as baselines respectively.

As shown in Figure 7.3, PA is able to improve IS-1 solutions by 14.8% on average, while the best results are achieved for medium-sized applications having a number of tasks in the range $[20, 60]$. The improvement is mainly obtained by considering hardware implementations having a low impact on the FPGA resources and by scheduling the tasks on the reconfigurable regions giving priority to those tasks whose implementations have higher efficiency indexes. However, for applications with a small number of tasks, there is less contention on the FPGA and thus the benefits of the proposed scheduler are less evident. Notice also that IS-1 tends to reduce the improvement gap of PA for large taskgraphs, this is due to the fact that IS-k spends an exponentially larger amount of time in optimizing the schedule when the number of tasks increases. Indeed IS-k is not a pure greedy algorithm and some of the time variables involved in its MILP formulation are allowed to be modified between subsequent scheduling iterations, thus leaving room for exploring a larger solution space. Overall, PA has a much

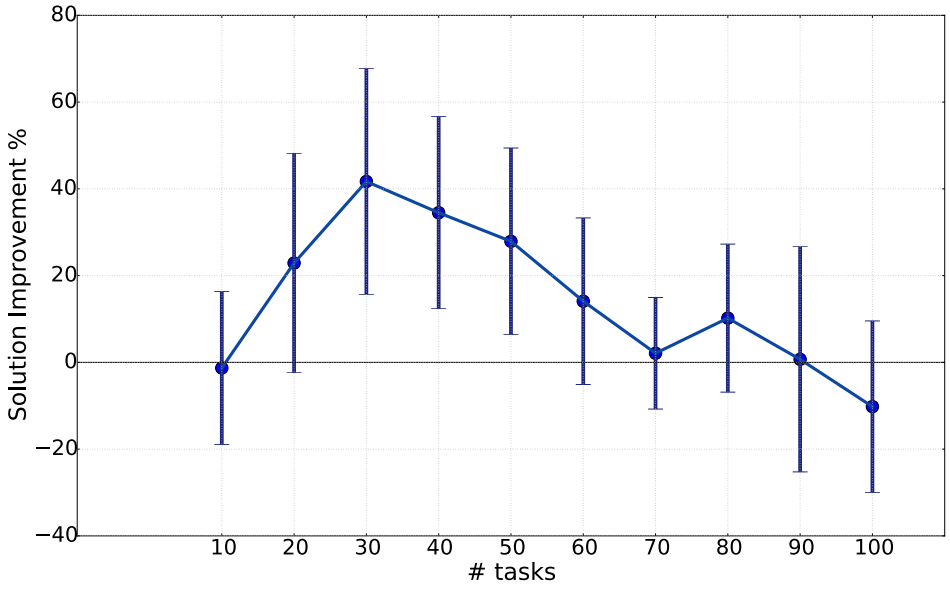


Figure 7.3: *Average solutions improvement of PA with respect to IS-1.*

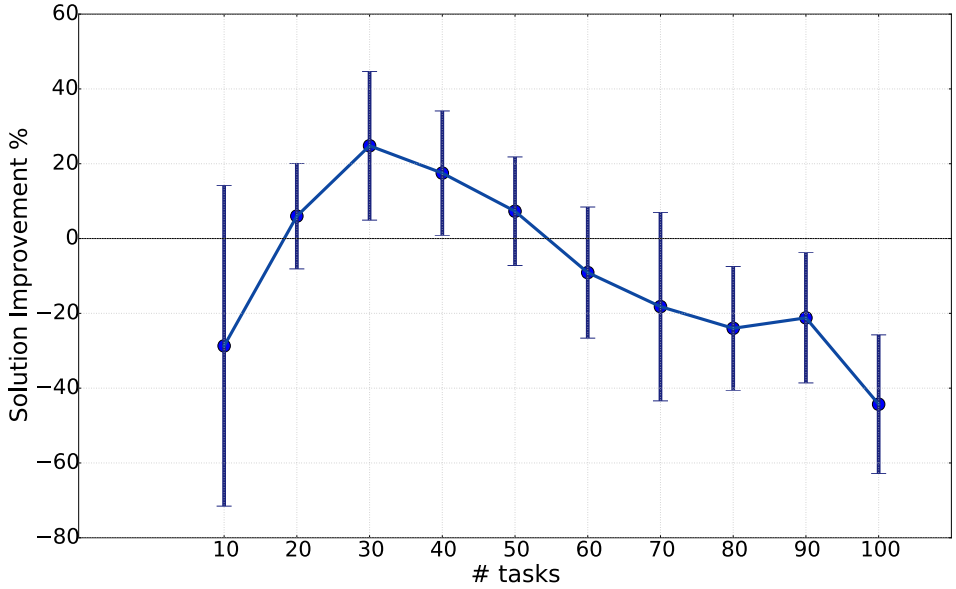


Figure 7.4: *Average solutions improvement of PA with respect to IS-5.*

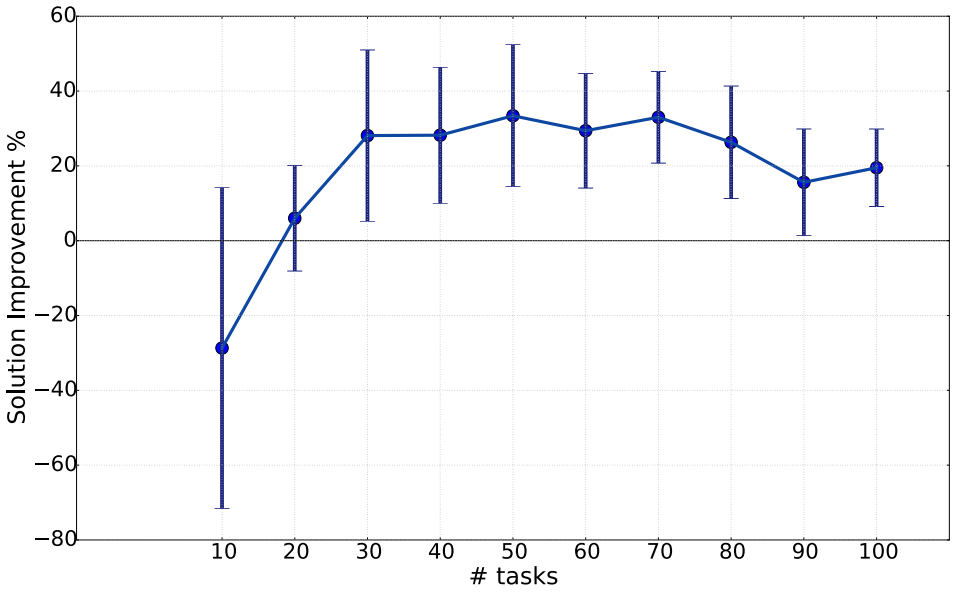


Figure 7.5: *Average solutions improvement of PA-R with respect to IS-5.*

lower running time than IS-1 and is in general able to improve IS-1 solutions. On the other hand, as shown in Figure 7.4, the improvement of PA with respect to IS-5 is reduced due to the additional flexibility given to the IS-5 MILP model that also translates in increased running time.

When the goal is shifted towards finding near optimal solutions, the randomized approach PA-R can be applied to trade off the small running time of PA to search for better schedules. Figure 7.5 shows the improvement of PA-R over IS-5 solutions where both algorithms have been executed for the same amount of time. For small applications consisting of 10 taskgrpahs, IS-5 is still able to provide better solutions by exploiting implementations having low execution time and high impact on the FPGA resources. However, for applications having more than 20 tasks FPGA contention becomes more relevant and PA-R is able to provide an average 22.3% improvement with respect to IS-5 by considering cost effective implementations.

It is worth noting that the high standard deviation shown in Figures 7.3, 7.4 and 7.5 is due to the fact that the number of tasks is not the only factor that influence the improvement achieved by the proposed approach. Further investigations are required in order to clearly identify these factors, however, we observed that the improvements achieved by PA-R with respect to IS-5 are more restrained when either the taskgraph expose a reduced level of parallelism or, at the opposite, when a great proportion of the application

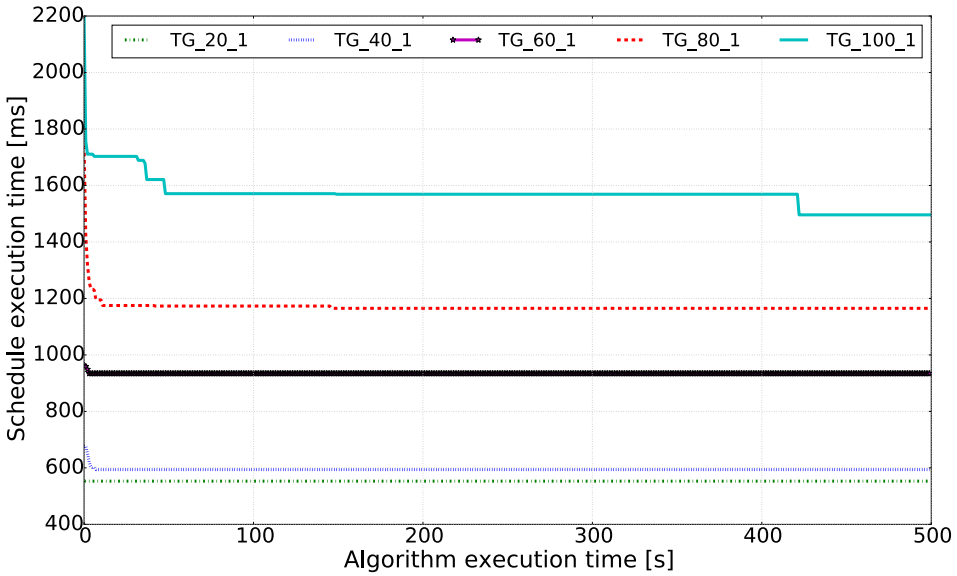


Figure 7.6: *Solution improvement over time for PA-R on different taskgraphs.*

tasks can be executed in parallel.

As a final analysis, we executed PA-R with an extended time limit of 1200 seconds over 5 of the given pseudo-random taskgraphs having a number of tasks in the range $\{20, 40, 60, 80, 100\}$. Figure 7.6 reports the best schedule execution time found by PA-R with respect to the running time of the algorithm. Notice that only the first 500 seconds of execution are shown since we observed that after this time interval the solutions found by PA-R did not improve further. As we can see from the figure, the algorithm generally converges to a good solution in a small amount of time and the convergence time increases together with the number of tasks involved in the taskgraph.

7.8 Final remarks

Within this chapter we proposed a novel scheduling approach for applications described in terms of DAG of tasks for SoC featuring homogeneous processing cores coupled with a partial dynamic reconfigurable FPGA. Two different algorithms were presented: the first one, is a deterministic scheduling heuristic that allows the designer to obtain a fast evaluation of the design performance on the target architecture, while the second one exploits randomization to explore a larger solutions space and is able to find op-

timized results. On average, For medium to large size applications, the proposed randomized algorithm reduces the execution time of the schedules by 22.3% with respect to the MILP-based approach presented in [23] that already proved to achieve better performance than [15].

Future work will investigate the possibility to leverage module reuse in order to further improve the solutions by removing the reconfiguration overhead for tasks sharing the same hardware implementations. Furthermore, we plan to relax the exact implementation selection phase in order to explore a larger solution phase that might lead to the identification of better solutions. Finally, we will also consider the possibility to explicitly model the communication overhead between the application tasks to improve the schedule accuracy.

CHAPTER 8

Conclusion

This chapter summarizes the contributions of this thesis and discusses current limitations and research directions for future work.

This dissertation presented CAD as an Adaptive Open-platform Service (CAOS), a platform whose main objective is to improve productivity and simplify the design of Field Programmable Gate Array (FPGA)-based accelerated systems, starting from pure high-level software implementations. As discussed in Chapter 1, the slowing rate at which general purpose processors improve performance is strongly pushing towards specialized hardware. We expect FPGAs to have a more prominent role in the upcoming years as a technology to achieve efficient and high performance solutions for a wide set of application domains both in the High Performance Computing (HPC) and cloud domains. Hence, by embracing this idea, we designed CAOS in a modular fashion (Chapter 2), providing well defined Application Programming Interface (API) that allow external researchers to integrate extensions or different implementations of the modules within the platform. Indeed, the second, yet not less important, objective of CAOS, is to foster research on tools and methods for accelerating software on FPGA-based architectures. The ultimate objective is indeed to reduce the productivity gap between pure software development and the implementation of

systems employing FPGA accelerators.

In order to achieve this ambitious objective, we have presented a design flow which revolves around the concept of *architectural template* as a mean to target the complexity and generality of high-level software, such as C/C++. An *architectural template* is a characterization of the accelerator both in terms of its computational model and the communication with the off-chip memory. An architectural template constrains the architecture to be generated on the reconfigurable hardware and poses restrictions on the application code that can be accelerated, so that the number and types of optimizations available can be tailored for a specific type of hardware implementation. Leveraging on the idea that different types of template architectures can be generated depending on specific characteristics of the high-level C/C++ code, we have presented three architectural templates integrated into CAOS.

The Master/Slave architectural template, discussed in Chapter 3, allows to achieve good acceleration results for a large set of C/C++ codes. The template leverages on existing High-Level Synthesis (HLS) vendor tools and features an automated design space exploration to evaluate different candidate optimizations. Overall, for regular applications with inherent data parallelism, the template is able to outperform CPU-based solutions in terms of energy efficiency and pure performance. Furthermore, the template allows to easily target both Xilinx Zynq-7000 System-on-Chip (SoC) and Amazon F1 cloud instances. While the gap between bespoke solutions and implementations achieved through CAOS leveraging on such template is still relevant (about 5x), yet the productivity improvement is remarkable, allowing to reduce the development time from several weeks to a few days.

When moving towards more specialized architectural templates, the range of supported codes reduces, but the higher specificity also allows to achieve higher performance and exploit the FPGA low-level resources more efficiently. This is the case of dataflow architectural template presented in Chapter 4 and the streaming architectural template discussed in Chapter 5. Broadly speaking, the dataflow architectural template targets C/C++ codes whose control flow graph is statically defined and for which accesses to the input/output arguments occur linearly through an outer loop iteration variable. For such type of functions, which are common in the finance and image analysis domain, the architectural template is able to automatically derive a dataflow implementation which, at the moment, is based on the MaxJ Domain Specific Language (DSL) from Maxeler. As we have shown, the template reaches performance very close to hand-tuned implementations yet in a very small amount of time, in the order of days instead

of weeks. On the other side, the streaming architectural template targets Iterative Stencil Loop (ISL) computations which are common in the scientific domain (e.g. linear equation solving, flow heat distribution simulation). Such template borrows from the dataflow architectural template the idea of computing while the data flows trough the accelerators' nodes. However, the architectures are built differently from the dataflow architectural template. This template generates a module for the computation of a single time step of the outermost loop of the ISL. Finally, a design space exploration based on floorplanning allows to maximize the number of modules that can be instantiated on the device, while minimizing inter-modules wire length. Such approach allows to improve performance of 13% compared to handmade designs and to reduce the design time up to 15x.

All the FPGA implementations achieved by the architectural templates described in Chapter 3, Chapter 4 and Chapter 5 are static, meaning that, once the FPGAs are configured, the configurations remain unchanged for the whole execution of the application. While this approach is simple and fits most of the use cases, there are circumstances in which the optimal design does not fit within a single FPGA or require different computational stages over time. In the last years, FPGA vendors such as Xilinx, have offered the possibility to perform Partial Dynamic Reconfiguration (PDR), i.e. reconfigure only a portion of the device at runtime while the rest of the design keeps working. Such technique opens up a wide range of possible designs, yet, it also requires the designer to floorplan a set of reconfigurable regions, to decide how to schedule the execution of the tasks of the applications and how to map them on the defined reconfigurable regions. For large designs this challenges become critical for achieving high quality results. Hence, in Chapter 6 we discussed an automated floorplanning framework with a configurable objective function, able to outperform state-of-the-art approaches in terms of exploration time and quality of the identified solution. Finally, Chapter 7, focused on the scheduling and mapping problem for applications taskgraphs on SoCs featuring processor cores and a partial dynamic reconfigurable FPGA. The synergy between the scheduler and mapping techniques discussed in Chapter 7, together with the floorplanning framework presented in Chapter 6, represent a building block for architectural templates in CAOS that need support for PDR.

8.1 Limitations and future works

Specific remarks and future works have been already presented in the final section of each chapter. Here instead, we want to discuss more general

remarks and future research directions on CAOS, as well as limitations that are still present within the current version of the platform.

One of the main assumptions considered throughout the discussion on CAOS, is that the granularity at which the platform analyzes acceleration opportunities and matches against one or more architectural templates, is at the function level. This assumption simplifies a number of analysis and allows to have a relatively small list of candidate functions to target, however, it comes at a cost. Indeed, depending on how the user expresses the computation and splits it across functions, the end result of the CAOS flow might change considerably. At the very extreme, if the application features a compute-intensive function in which the majority of the code is amenable to hardware acceleration, but a small portion cannot be accelerated (e.g. contains a few system calls for standard input/output), the whole function will not be considered for hardware acceleration. Nevertheless, CAOS will report diagnostic information to let the user modify the code and possibly move the undesired instructions out of the function.

The user might also affect the final design depending on how he/she organizes the function calls within the code. For instance, consider the case in which two functions B and C are always called one after the other within function A. Assume also that both B and C can be accelerated, whereas A contains code that does not map to any architectural template. In this scenario, the *architectural template applicability check* module detects two function trees rooted at B and C as hardware candidates. As of now, the platform will only allow to perform the acceleration of one of the two. Indeed the optimizations and estimations are on a per kernel basis and it is far from trivial to identify the optimal resource / performance trade-off for implementing the two kernels conflicting on the same set of resources. This could have been avoided by the insertion of a wrapper function D in charge of calling B and C.

Both issues arise either directly or indirectly from the same assumption: design flow granularity at the function level. In order to overcome such limitations and reduce the results sensitivity to code organization, a possible extension of the platform is to consider finer-grain elements, e.g. instructions or basic blocks, or even let each architectural template the possibility to specify the desired granularity. Departing from the function-level granularity assumption, also requires to properly define the kernel interfaces, since the inputs and outputs of the code portion to accelerate cannot be derived from the function arguments. Finally, the granularity choice might also impact on the time required to perform the applicability check analysis. All such considerations should be taken into account for extending the

CAOS platform in this direction.

Another critical aspect of CAOS, resides in the current implementations of several of its modules, especially in the context of the Master/Slave architectural template. Indeed, most of them rely on the resource and performance estimates of Xilinx Vivado HLS. Two shortcomings of this approach are: 1) latency estimates do not accurately account for data transfer overhead to and from the external memory and, 2) the execution time of the tool might be very high when heavy unrolling or memory partitioning is performed. One possible solution to such issues, would be to devise custom approaches for resource and performance estimation of a C/C++ code, without explicitly requiring proprietary tools. A very successful model that allows to perform bound and bottleneck analysis of the performance of a software code on a general purpose processor is the Berkeley roofline model [108]. The roofline model takes into account the memory bandwidth and peak performance of the architecture as well as the computational intensity of the algorithm. One of its main advantages is to condense in one graph the most relevant information on the architecture and the algorithm. Thanks to this, it allows to easily verify if the algorithm is compute or memory bound and to drive the proper optimizations. When dealing with FPGAs however, the architecture is not fixed a priori and a direct application of the roofline model is not feasible. In the literature, proposals to extended the roofline model for FPGA still rely on HLS tools [21]. Despite they allow to tackle the issue of underestimated memory transfer, the still incur in high execution time for complex HLS optimizations. With respect to this, an interesting research direction is to consider alternative solutions to evaluate the performance of C/C++ HLS code. For instance, it might be possible to obtain a meaningful, even if less accurate, design space exploration by relying on sub-optimal yet faster scheduling techniques.

Finally, we believe that in order to broaden the interest from the community towards the CAOS platform, an extensive set of predefined applications benchmark is needed. This work is currently being pursued and the overall idea is to allow other researchers to compare, on a common test bench, the results achieved with custom module implementations against the ones achieved with the default implementation of the CAOS modules.

Bibliography

[1] GNU Scientific Library, 2014.

[2] Saurabh N. Adya and Igor L. Markov. Fixed-outline floorplanning: enabling hierarchical design. *IEEE Trans. on VLSI Systems*, 11(6):1120–1135, 2003.

[3] D Applegate, R Bixby, Vasek Chvatal, and W Cook. Concorde TSP solver, 2006.

[4] M. Araya-Polo, J. Cabezas, M. Hanzich, M. Pericas, F. Rubio, I. Gelado, M. Shafiq, E. Morancho, N. Navarro, E. Ayguade, J. M. Cela, and M. Valero. Assessing accelerator-based hpc reverse time migration. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):147–162, Jan 2011.

[5] David F Bacon, Rodric Rabbah, and Sunil Shukla. Fpga programming for the masses. *Communications of the ACM*, 56(4):56–63, 2013.

[6] Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula. Tiling Stencil Computations to Maximize Parallelism. In *Proc. Intl. Conf. on High Performance Comp., Networking, Storage and Analysis*, pages 40:1–40:11, 2012.

[7] Pritha Banerjee, Megha Sangtani, and Susmita Sur-Kolay. Floorplanning for Partially Reconfigurable FPGAs. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 30(1):8–17, 2011.

[8] S. Banerjee, E. Bozorgzadeh, and N.D. Dutt. Integrating physical constraints in hw-sw partitioning for architectures with partial dynamic reconfiguration. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 14(11):1189–1202, Nov 2006.

[9] S. Banerjee, E. Bozorgzadeh, and N.D. Dutt. PARLGRAN: parallelism granularity selection for scheduling task chains on dynamically reconfigurable architectures. In *Proceedings of the 2006 Conference on Asia South Pacific Design Automation: ASP-DAC 2006, Yokohama, Japan, January 24-27, 2006*, pages 491–496, 2006.

[10] Christian Beckhoff, Dirk Koch, and Jim Torreson. Automatic floorplanning and interface synthesis of island style reconfigurable systems with goahead. In *Architecture of Computing Systems (ARCS)*, pages 303–316. 2013.

[11] Endri Bezati. *High-level synthesis of dataflow programs for heterogeneous platforms*. PhD thesis, EPFL, 2015.

Bibliography

[12] Mark Bohr. A 30 year retrospective on dennard's mosfet scaling paper. *IEEE Solid-State Circuits Society Newsletter*, 12(1):11–13, 2007.

[13] Cristiana Bolchini, Antonio Miele, and Chiara Sandionigi. Automated Resource-Aware Floorplanning of Reconfigurable Areas in Partially-Reconfigurable FPGA Systems. In *Proc. Intl. Conf. on Field Programmable Logic and Applications (FPL)*, pages 532–538, 2011.

[14] Salvatore Cardamone, Jonathan R Kimmitt, Hugh GA Burton, and Alex JW Thom. Field-programmable gate arrays and quantum monte carlo: Power efficient co-processing for scalable high-performance computing. *arXiv preprint arXiv:1808.02402*, 2018.

[15] R. Cattaneo, R. Bellini, G. Durelli, C. Pilato, M.D. Santambrogio, and D. Sciuto. Parasched: A reconfiguration-aware scheduler for reconfigurable architectures. In *Parallel Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 243–250, May 2014.

[16] Riccardo Cattaneo, Giuseppe Natale, Carlo Sicignano, Donatella Sciuto, and Marco Domenico Santambrogio. On How to Accelerate Iterative Stencil Loops: A Scalable Streaming-Based Approach. *ACM Trans. on Architectures and Code Optimization*, 12(4):53:1–53:26, December 2015.

[17] Riccardo Cattaneo, Giuseppe Natale, Carlo Sicignano, Donatella Sciuto, and Marco Domenico Santambrogio. On how to accelerate iterative stencil loops: a scalable streaming-based approach. *ACM Transactions on Architecture and Code Optimization (TACO)*, 12(4):53, 2016.

[18] Lei Cheng and M.D.F. Wong. Floorplan Design for Multimillion Gate FPGAs. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 25(12):2795–2805, 2006.

[19] Young-kyu Choi, Peng Zhang, Peng Li, and Jason Cong. Hlscope+: Fast and accurate performance estimation for fpga hls. In *Computer-Aided Design (ICCAD), 2017 IEEE/ACM International Conference on*, pages 691–698. IEEE, 2017.

[20] OpenSPL Consortium et al. Openspl: Revealing the power of spatial computing. Technical report, Technical report, Dec, 2013.

[21] Bruno da Silva, An Braeken, Erik H D'Hollander, and Abdellah Touhafi. Performance modeling for fpgas: extending the roofline model with high-level synthesis tools. *International Journal of Reconfigurable Computing*, 2013:7, 2013.

[22] Egidio DAngelo, Giovanni Danese, Giordana Florimbi, Francesco Leporati, Alessandra Majani, Stefano Masoli, Sergio Solinas, and Emanuele Torti. The human brain project: High performance computing for brain cells hw/sw simulation and understanding. In *2015 Euromicro Conference on Digital System Design (DSD)*, pages 740–747. IEEE, 2015.

[23] Enrico A Deiana, Marco Rabozzi, Riccardo Cattaneo, and Marco D Santambrogio. A multiobjective reconfiguration-aware scheduler for fpga-based heterogeneous architectures. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6. IEEE, 2015.

[24] Emanuele Del Sozzo, Riyadh Baghdadi, Saman Amarasinghe, and Marco D Santambrogio. A common backend for hardware acceleration on fpga. In *2017 IEEE 35th International Conference on Computer Design (ICCD)*, pages 427–430. IEEE, 2017.

[25] Emanuele Del Sozzo, Lorenzo Di Tucci, and Marco D Santambrogio. A highly scalable and efficient parallel design of n-body simulation on fpga. In *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International*, pages 241–246. IEEE, 2017.

[26] Emanuele Del Sozzo, Marco Rabozzi, Lorenzo Di Tucci, Donatella Sciuto, and Marco D Santambrogio. A scalable fpga design for cloud n-body simulation. In *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 1–8. IEEE, 2018.

[27] Emanuele Del Sozzo, Marco Rabozzi, Lorenzo Di Tucci, Donatella Sciuto, and Marco D Santambrogio. A scalable fpga design for cloud n-body simulation. In *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 1–8. IEEE, 2018.

[28] Robert H Dennard, Fritz H Gaensslen, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[29] J. B. Dennis. Data flow supercomputers. *Computer*, 13(11):48–56, November 1980.

[30] Lorenzo Di Tucci, Kenneth O'Brien, Michaela Blott, and Marco D Santambrogio. Architectural optimizations for high performance and energy efficient smith-waterman implementation on fpgas using opencl. In *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 716–721. IEEE, 2017.

[31] Lorenzo Di Tucci, Marco Rabozzi, Luca Stornaiuolo, and Marco D Santambrogio. The role of cad frameworks in heterogeneous fpga-based cloud systems. In *Computer Design (ICCD), 2017 IEEE International Conference on*, pages 423–426. IEEE, 2017.

[32] Docker. https://www.docker.com.

[33] Paul Feautrier and Christian Lengauer. Polyhedron model. In *Encyclopedia of Parallel Computing*, pages 1581–1592. Springer, 2011.

[34] S.P. Fekete, E. Kohler, and J. Teich. Optimal fpga module placement with temporal precedence constraints. In *Design, Automation and Test in Europe, 2001. Conference and Exhibition 2001. Proceedings*, pages 658–665, 2001.

[35] Yan Feng and Dinesh P. Mehta. Heterogeneous Floorplanning for FPGAs. In *Proc. Intl. Conf. on VLSI Design*, pages 257–262, 2006.

[36] Blair Fort, Andrew Canis, Jongsok Choi, Nazanin Calagar, Ruolong Lian, Stefan Hadjis, Yu Ting Chen, Mathew Hall, Bain Syrowik, Tomasz Czajkowski, et al. Automating the design of processor/accelerator embedded systems with legup high-level synthesis. In *Embedded and Ubiquitous Computing (EUC), 2014 12th IEEE International Conference on*, pages 120–129. IEEE, 2014.

[37] Lin Gan, Haohuan Fu, Chao Yang, Wayne Luk, Wei Xue, Oskar Mencer, Xiaomeng Huang, and Guangwen Yang. A highly-efficient and green data flow engine for solving euler atmospheric equations. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–6. IEEE, 2014.

[38] Michael R Garey and David S Johnson. *Computers and intractability*, volume 174. freeman San Francisco, 1979.

[39] Sabih H Gerez. *Algorithms for VLSI design automation*, volume 8. Wiley New York, 1999.

[40] Soheil Ghiasi, Ani Nahapetian, and Majid Sarrafzadeh. An optimal algorithm for minimizing run-time reconfiguration delay. *ACM Transactions on Embedded Computing Systems (TECS)*, 3(2):237–256, 2004.

[41] D. E. Goldberg. *Genetic Algorithms in Search Optimization and Machine Learning*. Addison-Wesley, 1989.

[42] Paul Grigoraş, Xinyu Niu, Jose GF Coutinho, Wayne Luk, Jacob Bower, and Oliver Pell. Aspect driven compilation for dataflow designs. In *Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference on*, pages 18–25. IEEE, 2013.

Bibliography

[43] Eduardo Gudis, Pullan Lu, David Berends, Kevin Kaighn, Gooitzen Wal, Gregory Buchanan, Sek Chai, and Michael Piacentino. An embedded vision services framework for heterogeneous accelerators. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 598–603, 2013.

[44] LLC Gurobi Optimization. Gurobi optimizer reference manual, 2018.

[45] James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. Darkroom: Compiling high-level image processing code into hardware pipelines. *ACM Trans. Graph.*, 33(4):144:1–144:11, July 2014.

[46] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2017.

[47] Justin Holewinski, Louis-Noël Pouchet, and P. Sadayappan. High-performance Code Generation for Stencil Computations on GPU Architectures. In *Proc. of the Intl. Conf. on Supercomputing*, pages 311–320, 2012.

[48] Tang Jian. An o (2 0.304 n) algorithm for solving maximum independent set problem. *IEEE Transactions on Computers*, 100(9):847–851, 1986.

[49] S. Kapur and D. E. Long. N-body problems: IES^3: Efficient electrostatic and electromagnetic simulation. *IEEE Computational Science and Engineering*, 5(4):60–67, Oct 1998.

[50] Vinod Kathail, James Hwang, Welson Sun, Yogesh Chobe, Tom Shui, and Jorge Carrillo. Sdsoc: A higher-level programming environment for zynq soc and ultrascale+ mpsoc. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 4–4. ACM, 2016.

[51] M. S. Kim and K. Shimada. *Geometric Modeling and Processing - GMP 2006: 4th International Conference, GMP 2006, Pittsburgh, PA, USA, July 26-28, 2006, Proceedings*. Springer, 2006.

[52] Dimitris Koukounis, Christos Ttofis, Agathoklis Papadopoulos, and Theocharis Theocharides. A high performance hardware architecture for portable, low-power retinal vessel segmentation. *INTEGRATION, the VLSI journal*, 47(3):377–386, 2014.

[53] Yuet Ming Lam, J Coutinho, Wayne Luk, and Philip Heng Wai Leong. Mapping and scheduling with task clustering for heterogeneous computing systems. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pages 275–280. IEEE, 2008.

[54] J. Lamprecht and B. Hutchings. Profiling FPGA floor-planning effects on timing closure. In *Proc. Intl. Conf. on Field Programmable Logic and Applications (FPL)*, pages 151–156, 2012.

[55] Chris Lattner. Llvm and clang: Next generation compiler technology. In *The BSD conference*, pages 1–2, 2008.

[56] Zhiyuan Li and Yonghong Song. Automatic Tiling of Iterative Stencil Loops. *ACM Trans. on Programming Languages and Systems*, 26(6):975–1028, November 2004.

[57] Liang Ma, Luciano Lavagno, Mihai Teodor Lazarescu, and Arslan Arif. Acceleration by inline cache for memory-intensive algorithms on fpga via high-level synthesis. *IEEE Access*, 5:18953–18974, 2017.

[58] Junichiro Makino and Makoto Taiji. *Scientific Simulations with Special-Purpose Computers–the GRAPE Systems*. 1998.

[59] J. Marshall, A. Adcroft, C. Hill, L. Perelman, and C. Heisey. A finite-volume, incompressible Navier-Stokes model for studies of the ocean on parallel computers. *Journal of Geophysical Research*, 102(C3):5733–5752, March 1997.

[60] Maxeler Technologies. Maxeler technologies website.

[61] Bingfeng Mei, Patrick Schaumont, and Serge Vernalde. A hardware-software partitioning and scheduling algorithm for dynamically reconfigurable embedded systems. In *Proceedings of ProRISC*, pages 405–411. Citeseer, 2000.

[62] Ingo Molnar. Performance counters for linux, v8. *LWN. net, http://lwn. net/Articles/310150*, 2008.

[63] Alessio Montone, Marco D. Santambrogio, Donatella Sciuto, and Seda Ogrenci Memik. Placement and Floorplanning in Dynamically Reconfigurable FPGAs. *ACM Trans. on Reconfigurable Technology and Systems*, 3(4), 2010.

[64] G Moore. Cramming more components onto integrated circuits', electronics, vol. 38, no. 8, 1965.

[65] Jean-Michel Muller and Jean-Michael Muller. *Elementary functions*. Springer, 2006.

[66] A. A. Nacci, V. Rana, F. Bruschi, D. Sciuto, P. di Milano, I. Beretta, and D. Atienza. A high-level synthesis flow for the implementation of iterative stencil loop algorithms on FPGA devices. In *Proc. of Design Automation Conference (DAC)*, pages 1–6, 2013.

[67] A. Nakano, R. K Kalia, and P. Vashishta. Multiresolution Molecular Dynamics Algorithm for Realistic Materials Modeling on Parallel Computers. *Computer Physics Comm.*, 83(2):197–214, 1994.

[68] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, et al. A survey and evaluation of fpga high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, 2016.

[69] Tetsu Narumi, Ryutaro Susukita, Toshikazu Ebisuzaki, Geoffrey McNiven, and Bruce Elmegreen. Molecular dynamics machine: Special-purpose computer for molecular dynamics simulations. *Molecular Simulation*, 21(5-6):401–415, 1999.

[70] Giuseppe Natale, Giulio Stramondo, Pietro Bressana, Riccardo Cattaneo, Donatella Sciuto, and Marco D Santambrogio. A polyhedral model-based framework for dataflow implementation on fpga devices of iterative stencil loops. In *Proceedings of the 35th International Conference on Computer-Aided Design*, page 77. ACM, 2016.

[71] Christopher E. Neely, Gordon Brebner, and Weijia Shang. ReShape: Towards a High-Level Approach to Design and Operation of Modular Reconfigurable Systems. *ACM Trans. Reconfigurable Technol. Syst.*, 6(1):5:1–5:23, May 2013.

[72] Anna Maria Nestorov, Enrico Reggiani, Hristina Palikareva, Pavel Burovskiy, Tobias Becker, and Marco D Santambrogio. A scalable dataflow implementation of curran's approximation algorithm. In *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International*, pages 150–157. IEEE, 2017.

[73] Tuan D. A. Nguyen and Akash Kumar. PRFloor: An Automatic Floorplanner for Partially Reconfigurable FPGA Systems. In *Proc. of Intl. Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 149–158, 2016.

[74] Alexander Novikov, Scott Alexander, Nino Kordzakhia, and Timothy Ling. Pricing of asian-type and basket options via upper and lower bounds. *arXiv preprint arXiv:1612.08767*, 2016.

[75] NVIDIA Corp. nbody - CUDA N-Body Simulation. `https://docs.nvidia.com/cuda/cuda-samples/index.html#cuda-n-body-simulation`.

[76] Kenneth O'Brien, Lorenzo Di Tucci, Gianluca Durelli, and Michaela Blott. Towards exascale computing with heterogeneous architectures. In *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 398–403. IEEE, 2017.

Bibliography

[77] Nobuyuki Otsu. A threshold selection method from gray-level histograms. *Automatica*, 11(285-296):23–27, 1975.

[78] A. Panella, M. D. Santambrogio, F. Redaelli, F. Cancare, and D. Sciuto. A design workflow for dynamically reconfigurable multi-fpga systems. In *2010 18th IEEE/IFIP International Conference on VLSI and System-on-Chip*, pages 414–419, Sept 2010.

[79] Francesco Peverelli, Marco Rabozzi, Emanuele Del Sozzo, and Marco D Santambrogio. Oxigen: A tool for automatic acceleration of c functions into dataflow fpga-based kernels. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 91–98. IEEE, 2018.

[80] Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. Programming heterogeneous systems from an image processing dsl. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(3):26, 2017.

[81] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, et al. Going deeper with embedded fpga platform for convolutional neural network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 26–35. ACM, 2016.

[82] M. Rabozzi, A. Miele, and M.D. Santambrogio. Floorplanning for Partially-Reconfigurable FPGAs via Feasible Placements Detection. In *Proc. Intl. Symp. on Field-Programmable Custom Computing Machines (FCCM)*, pages 252–255, 2015.

[83] Marco Rabozzi, Rolando Brondolin, Giuseppe Natale, Emanuele Del Sozzo, Michael Huebner, Andreas Brokalakis, Catalin Ciobanu, Dirk Stroobandt, and Marco Domenico Santambrogio. A cad open platform for high performance reconfigurable systems in the extra project. In *VLSI (ISVLSI), 2017 IEEE Computer Society Annual Symposium on*, pages 368–373. IEEE, 2017.

[84] Marco Rabozzi, Emanuele Del Sozzo, Lorenzo Di Tucci, and Marco D Santambrogio. Fivepoint algorithm: An efficient cloud-based fpga implementation. In *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 1–8. IEEE, 2018.

[85] Marco Rabozzi, John Lillis, and Marco D Santambrogio. Floorplanning for partially-reconfigurable fpga systems via mixed-integer linear programming. In *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, pages 186–193. IEEE, 2014.

[86] Marco Rabozzi, Antonio Miele, and Marco D Santambrogio. Floorplanning for partially-reconfigurable fpgas via feasible placements detection. In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, pages 252–255. IEEE, 2015.

[87] Marco Rabozzi, Giuseppe Natale, Biagio Festa, Antonio Miele, and Marco D Santambrogio. Optimizing streaming stencil time-step designs via fpga floorplanning. In *Field Programmable Logic and Applications (FPL), 2017 27th International Conference on*, pages 1–4. IEEE, 2017.

[88] Steffen Rebennack. Stable set problem: Branch & cut algorithms. In Christodoulos A. Floudas and Panos M. Pardalos, editors, *Encyclopedia of Optimization*, pages 3676–3688. Springer, 2009.

[89] F. Redaelli, M. D. Santambrogio, and S. Ogrenci Memik. An ilp formulation for the task graph scheduling problem tailored to bi-dimensional reconfigurable architectures. *Int. J. Reconfig. Comput.*, 2009:7:1–7:12, January 2009.

[90] Davide Sampietro, Chiara Crippa, Lorenzo Di Tucci, Emanuele Del Sozzo, and Marco D Santambrogio. Fpga-based pairhmm forward algorithm for dna variant calling. In *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 1–8. IEEE, 2018.

[91] K. Sano, Y. Hatsuda, and S. Yamamoto. Scalable Streaming-Array of Simple Soft-Processors for Stencil Computations with Constant Memory-Bandwidth. In *Proc. of Intl. Symposium on Field-Programmable Custom Computing Machines*, pages 234–241, 2011.

[92] Kentaro Sano. Fpga-based systolic computational-memory array for scalable stencil computations. In *High-Performance Computing Using FPGAs*, pages 279–303. Springer, 2013.

[93] Kentaro Sano, Yoshiaki Hatsuda, and Satoru Yamamoto. Multi-fpga accelerator for scalable stencil computation with constant memory bandwidth. *IEEE Transactions on Parallel and Distributed Systems*, 25(3):695–705, 2014.

[94] M. D. Santambrogio, D. Pnevmatikatos, K. Papadimitriou, C. Pilato, G. Gaydadjiev, D. Stroobandt, T. Davidson, T. Becker, T. Todman, W. Luk, A. Bonetto, A. Cazzaniga, G. C. Durelli, and D. Sciuto. Smart technologies for effective reconfiguration: The faster approach. In *7th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, pages 1–7, July 2012.

[95] Marco D Santambrogio and Donatella Sciuto. Design methodology for partial dynamic reconfiguration: a new degree of freedom in the hw/sw codesign. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8. IEEE, 2008.

[96] Michael J Schulte, Mike Ignatowski, Gabriel H Loh, Bradford M Beckmann, William C Brantley, Sudhanva Gurumurthi, Nuwan Jayasena, Indrani Paul, Steven K Reinhardt, and Gregory Rodgers. Achieving exascale capabilities through heterogeneous computing. *IEEE Micro*, 35(4):26–36, 2015.

[97] David E Shaw, JP Grossman, Joseph A Bank, Brannon Batson, J Adam Butts, Jack C Chao, Martin M Deneroff, Ron O Dror, Amos Even, Christopher H Fenton, et al. Anton 2: raising the bar for performance and programmability in a special-purpose molecular dynamics supercomputer. In *Proceedings of the international conference for high performance computing, networking, storage and analysis*, pages 41–53. IEEE Press, 2014.

[98] Love Singhal and Elaheh Bozorgzadeh. Multi-layer Floorplanning on a Sequence of Reconfigurable Designs. In *Proc. Intl. Conf. on Field Programmable Logic and Applications (FPL)*, pages 1–8, 2006.

[99] Robert Stewart, Greg Michaelson, Deepayan Bhowmik, Paulo Garcia, and Andy Wallace. A dataflow ir for memory efficient ripl compilation to fpgas. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 174–188. Springer, 2016.

[100] Prasanna Sundararajan. High performance computing using fpgas. *Xilinx White Paper: FPGAs*, pages 1–15, 2010.

[101] Michael B Taylor. A landscape of the new dark silicon design regime. *IEEE Micro*, 33(5):8–19, 2013.

[102] C. Tomas, L. Cazzola, D. Oriato, O. Pell, D. Theis, G. Satta, and E. Bonomi. Acceleration of the anisotropic pspi imaging algorithm with dataflow engines. In *Proceedings of 82nd Annual Meeting and International Exposition of the Society of Exploration Geophysics-SEG, Las Vegas - Nevada*, 2012.

[103] Topic Embedded Products. Dynamic process loader - dyplo.

[104] Robert A Van Engelen. Efficient symbolic analysis for optimizing compilers. In *International Conference on Compiler Construction*, pages 118–132. Springer, 2001.

Bibliography

[105] Kizheppatt Vipin and Suhaib A. Fahmy. Architecture-aware reconfiguration-centric floorplanning for partial reconfiguration. In *Proc. Intl. Conf. on Reconfigurable Computing (ARC)*, pages 13–25, 2012.

[106] Matthew Wall. GAlib, 2007.

[107] Wen Wan and Jeffrey B Birch. An improved hybrid genetic algorithm with a new local search procedure. *Journal of Applied Mathematics*, 2013, 2013.

[108] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[109] Loring Wirbel. Xilinx sdaccel: a unified development environment for tomorrow's data center. Technical report, Technical Report, The Linley Group Inc, 2014.

[110] Xilinx Inc. Vivado Design Suite. `http://www.xilinx.com/products/design-tools/vivado.html`.

[111] Xilinx Inc. Vivado High-Level Synthesis. `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug902-vivado-high-level-synthesis.pdf`.

[112] Xilinx Inc. Vivado Design Suite User Guide: Partial Reconfiguration, 2015.

[113] Xilinx Inc. Zynq-7000 all programmable SoC: Technical Reference Manual, 2015.

[114] Shaon Yousuf and Ann Gordon-Ross. DAPR: Design Automation for Partially Reconfigurable FPGAs. In *Proc. Intl. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pages 1–7, 2010.

[115] Ping-Hung Yuh, Chia-Lin Yang, and Yao-Wen Chang. Temporal floorplanning using the three-dimensional transitive closure subGraph. *ACM Trans. on Design Automomation of Electronic Systems*, 12(4), 2007.

[116] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 161–170. ACM, 2015.

[117] Zhiru Zhang and Bin Liu. Sdc-based modulo scheduling for pipeline synthesis. In *Proceedings of the International Conference on Computer-Aided Design*, pages 211–218. IEEE Press, 2013.