Leandro de Souza Rosa

Fast Code Exploration for Pipeline Processing in FPGA Accelerators

Doctoral dissertation submitted to the Institute of Mathematics and Computer Sciences – ICMC-USP, in partial fulfillment of the requirements for the degree of the Doctorate Program in Computer Science and Computational Mathematics. *FINAL VERSION*

Concentration Area: Computer Science and Computational Mathematics

Advisor: Prof. Dr. Vanderlei Bonato

USP – São Carlos
June 2019

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados inseridos pelo(a) autor(a)

S729f

Souza Rosa, Leandro de
Fast Code Exploration for Pipeline Processing in FPGA Accelerators / Leandro de Souza Rosa; orientador Vanderlei Bonato. -- São Carlos, 2019.
138 p.

Tese (Doutorado - Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional) -- Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2019.

1. Field-Programmable Gate Array. 2. High-Level Synthesis. 3. Pipeline. 4. Design Space Exploration. I. Bonato, Vanderlei, orient. II. Título.

Bibliotecários responsáveis pela estrutura de catalogação da publicação de acordo com a AACR2:
Gláucia Maria Saia Cristianini - CRB - 8/4938
Juliana de Souza Moraes - CRB - 8/6176

Leandro de Souza Rosa

Exploração Rápida de Códigos para Processamento Pipeline em Aceleradores FPGA

Tese apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP, como parte dos requisitos para obtenção do título de Doutor em Ciências – Ciências de Computação e Matemática Computacional. *VERSÃO REVISADA*

Área de Concentração: Ciências de Computação e Matemática Computacional

Orientadora: Prof. Dr. Vanderlei Bonato

USP – São Carlos
Junho de 2019

To gramma, mamma, and "sister-now-mama".

ACKNOWLEDGEMENTS

I would like to acknowledge Prof. Dr. Bonato and Prof. Dr. Bouganis for advising, teaching, and guiding me through the years of this project.

Further thanks to Profs. Dr. Toledo and Dr. Delbem for all lessons.

Personal thanks to my dears:

Rafael Carlet, Vinicius Gomes, Thiago Pantaleão, Caio Felipe dos Santos, Breno Andrade, and Paulo Caue dos Santos.

Douglas Henrique, Marcelo Maia, Djalma Ferraz, Rafael Guazelli, Etiene Consoni, Marcus Túlio, Cristovam Cerqueira, and to the memory of Julio Cesar Sousa.

Arthur Brandolin, Flávio Moreira, Amanda Mondolin, Sérgio Dias, Natalia Gonzaga, Marcelo Pontes, Renato Winnik, Mateus Moreira, and Amanda Vanessa Prado.

Lana Galego, Ederson Lima, Luiz Souza, Henrique Nascimento, Daniel Mazak, Rogério Pompermayer, Dominik Marksim, Pedro Nakasu, André Perina, and Caio S. Oliveira.

Rodolfo Silva Martins.

Also, I would like to acknowledge the São Paulo Research Foundation (FAPESP) for the financial support for this project (grant #2014/14918-2).

“Nobody’s born knowing. And nobody dies knowing much.”
- Therezinha Malta de Souza “Gramma”

ABSTRACT

LEANDRO, DE S. R. **Fast Code Exploration for Pipeline Processing in FPGA Accelerators**. 2019. 138 p. Tese (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2019.

The increasing demand for energy efficient computing has endorsed the usage of Field-Programmable Gate Arrays to create hardware accelerators for large and complex codes. However, implementing such accelerators involve two complex decisions. The first one lies in deciding which code snippet is the best to create an accelerator, and the second one lies in how to implement the accelerator. When considering both decisions concomitantly, the problem becomes more complicated since the code snippet implementation affects the code snippet choice, creating a combined design space to be explored. As such, a fast design space exploration for the accelerators implementation is crucial to allow the exploration of different code snippets. However, such design space exploration suffers from several time-consuming tasks during the compilation and evaluation steps, making it not a viable option to the snippets exploration. In this work, we focus on the efficient implementation of pipelined hardware accelerators and present our contributions on speeding up the pipelines creation and their design space exploration. Towards loop pipelining, the proposed approaches achieve up to 100× speed-up when compared to the state-uf-the-art methods, leading to 164 hours saving in a full design space exploration with less than 1% impact in the final results quality. Towards design space exploration, the proposed methods achieve up to 9.5× speed-up, keeping less than 1% impact in the results quality.

Keywords: Field-Programmable Gate Array, High-Level Synthesis, Pipeline, Design Space Exploration.

RESUMO

LEANDRO, DE S. R. **Exploração Rápida de Códigos para Processamento Pipeline em Aceleradores FPGA**. 2019. 138 p. Tese (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2019.

A demanda crescente por computação energeticamente eficiente tem endossado o uso de *Field-Programmable Gate Arrays* para a criação de aceleradores de hardware para códigos grandes e complexos. Entretanto, a implementação de tais aceleradores envolve duas decições complexas. O primeiro reside em decidir qual trecho de código é o melhor para se criar o acelerador, e o segundo reside em como implemental tal acelerador. Quando ambas decisões são comsideradas concomitantemente, o problema se torna ainda mais complicado dado que a implemetação do trecho de código afeta a seleção dos trechos de código, criando um espaço de projeto combinatorial a ser explorado. Dessa forma, uma exploração do espaço de projeto rápida para a implementação de aceleradores é crucial para habilitar a exploração de diferentes trechos de código. Contudo, tal exploração do espaço de projeto é impedida por várias taferas que consumem tempo durante os passos de compilação a análise, o que faz da exploração de trechos de códigos inviável. Neste trabalho, focamos na implementação eficiente de aceleradores pipeline em hardware e apresentamos nossas contribuições para o aceleramento da criações de pipelines e de sua exploração do espaço de projeto. Referente à criação de pipelines, as abordagens propostas alcançam uma aceleração de até $100\times$ quando comparadas às abordagens do estado-da-arte, levando à economia de 164 horas em uma exploração de espaço de projeto completa com menos de 1% de impacto na qualidade dos resultados. Referente à exploração do espaço de projeto, as abordagens propostas alcançam uma aceleração de até $9.5\times$, mantendo menos de 1% de impacto na qualidade dos resultados.

Palavras-chave: *Field Programmable Gate Arrays*, Síntese em Alto Nível, Pipeline, Exploração do Espaço de Projeto.

LIST OF FIGURES

LIST OF CHARTS

LIST OF ALGORITHMS

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

(AS)^2 Accelerator Synthesis using Algorithmic Skeletons for rapid DSE.

ACO Ant Colony Optimization.

ADRS Average Distance from Reference Set.

Aladdin pre-RTL, power performance simulator for rapid accelerator-centric DSE.

ALM Adaptive Logic Module.

AO-DSE Area Oriented DSE.

ASA Adaptive Simulated Annealing DSE.

ASAP As Soon As Possible.

ASIC Application-Specific Integrated Circuit.

AutoAccel Automated Accelerator Generation and Optimization with Composable, Parallel and Pipeline Architecture.

BB Basic Block.

BBO Bambu HLS Compiler.

BRAM Block RAM.

DC-ExpA Divide and Conquer Exploration Algorithm.

DEMUX De-Multiplexers.

DFG Data-Flow Graph.

DSE Design Space Exploration.

DSG Design Space Graph.

DSP Digital Signal Processor.

ERDSE Efficient and Reliable HLS DSE.

FPGA Field-Programmable Gate Array.

FU Functional Unity.

GA Genetic Algorithm.

GAS GA Modulo Scheduler.

GCC GNU compiler Collection.

GPP General Purpose Processor.

GPU Graphics Processing Unit.

HDL Hardware Description Language.

HLS High-Level Synthesis.

II Initiation Interval.

ILP Integer Linear Program.

ILPS ILP Modulo scheduler.

IOC Intel OpenCL Framework for OpenCL.

IP Intellectual Property.

IR Intermediate Representation.

LBDSE Lattice-Based DSE.

Lin-Analyzer Fast and Accurate FPGA Performance Estimation and DSE.

LLVM Low-Level Virtual Machine.

LUP LegUp HLS Compiler.

MKDSE Multi-Kernel DSE.

ML Machine Learning.

MLP Machine Learning Predictive DSE.

MPSeeker High-Level Analysis Framework for Fine- and Coarse-Grained Parallelism on FPGAs.

MRT Module Reservation Table.

MUX Multiplexers.

NIS Non-Iterative Modulo Scheduler.

NRCASAP Non-Resource Constrained As Soon As Possible.

P+LBDSE Path Seed Lattice-Based DSE.

PBDRO Polyhedral Based Data-Reuse Optimization.

PBDSE Path-based DSE.

PCA Principal Component Analysis.

PCIe Peripheral Component Interconnect Express.

PMK Probabilistic MultiKnob HLS DSE.

PSCEFP Polyhedral-based SystemC framework for Effective Low-Power DSE.

RAM Random Access Memory.

RHS Right-Hand Side.

SDC System of Difference Constraints.

SDCS SDC Modulo Scheduler,.

SDSE Standalone Design Space Exploration.

SMS Swing Modulo Scheduler.

SPIRIT Spectral-Aware Pareto Iterative Refinement Optimization for Supervised HLS.

SRPD Set Relative Position Distance.

TUM Totally Uni-Modular.

VVD Xilinx Vivado HLS Compiler.

CONTENTS

CHAPTER

1

INTRODUCTION

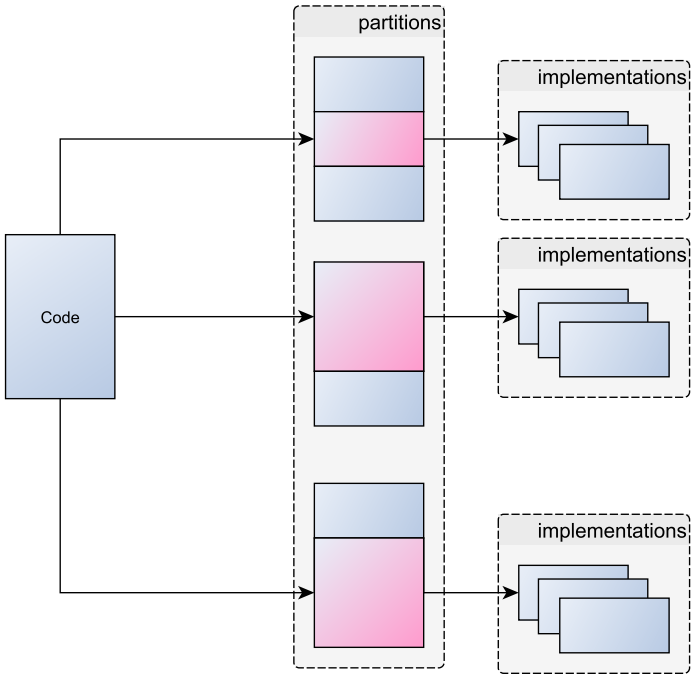
For the past years, Field-Programmable Gate Arrays (FPGAs) have been shown to be promising platforms to achieve computational FLOPS/Watt performance comparable to Graphics Processing Units (GPUs) while achieving higher energy efficiency (Mittal and Vetter 2014). Such energy efficiency makes FPGA promising platforms to accelerate the next generations of Neural Networks (Nurvitadhi *et al.* 2017), Sparse Matrix Algebra (Giefers *et al.* 2016), network applications (Nurvitadhi *et al.* 2016), financial market applications (Schryver *et al.* 2011), image processing (Fowers *et al.* 2012), and data centres (Weerasinghe *et al.* 2015). As a result, a common configuration is to have GPP acting as a host and FPGA as a hardware accelerator.

Such applications are usually not implemented as a whole in FPGA for two main reasons. The first one is that the applications are too large and would not fit in FPGA. The second reason if that such applications have parts that are suitable to be implemented in FPGA, while other parts are more suitable to be processed on traditional General Purpose Processors (GPPs) or GPU (Escobar, Chang and Valderrama 2016).

An application code can be separated in different forms, creating a combinatorial number of possible partitions to be executed on the host and the accelerator. Each "part" of the partitioned code is called "snippet". High-Level Synthesis (HLS) compilers (Nane *et al.* 2016) have been adopted in this scenario, which mitigates the creation of hardware accelerators from high-level languages (typically a `C`-like one).

For FPGA accelerators, snippets can be implemented in different ways, resulting in different throughputs, resources usage, and power consumptions. To control how a snippet is mapped on the FPGA, HLS languages provide several directives, making the possible number of implementations a combination of all possible values of the directives. A snipped implementation in FPGA is called kernel, and the set of its characteristics as resources usage, power consumption, throughput, and frequency are generically called "kernel metrics", or "hardware metrics". Figure 1 exemplifies an application code partitioned in different ways, each one with many possible

Figure 1 – Different code partitions with different possible implementations.



Source: Elaborated by the author.

different implementations.

As such, the Design Space Exploration (DSE) has two fronts. The first being the code partition exploration, which consists in deciding which code snippets will be implemented as hardware accelerators and which will be executed by the host. The second front is the kernel DSE, which consists in deciding which values for each possible directive given by the HLS compiler will return the desired hardware metrics.

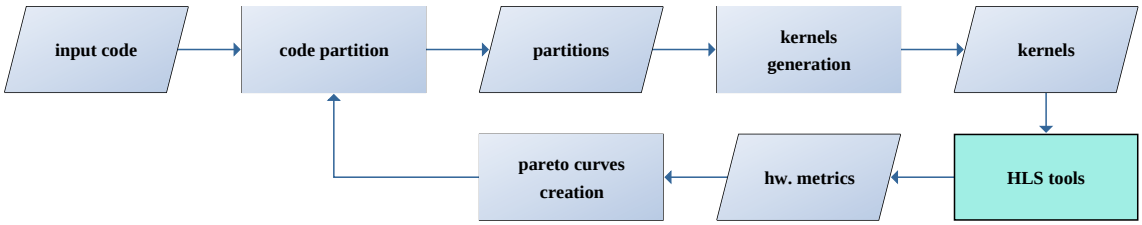
However, the kernel implementation affects the code snippets choice, making the whole problem a combination of both explorations. We define the simultaneous exploration of both fronts as “global DSE” and the kernel implementation exploration as “local DSE”.

Figure 2 presents a global DSE flow overview, where the code is partitioned, and each partition is used to generate several kernels, which are compiled by HLS tools. After compilation, the hardware resources usage, maximum frequency, number of clock cycles, and energy consumption (namely “hardware metrics”) are used to select the best designs, which is generally done through Pareto-curve analysis.

The HLS tools’ role in Figure 2 is to create a hardware design for the given kernel code input. Figure 3 expands the “HLS tools” box in Figure 2 to detail the local DSE flows performed within the HLS tools.

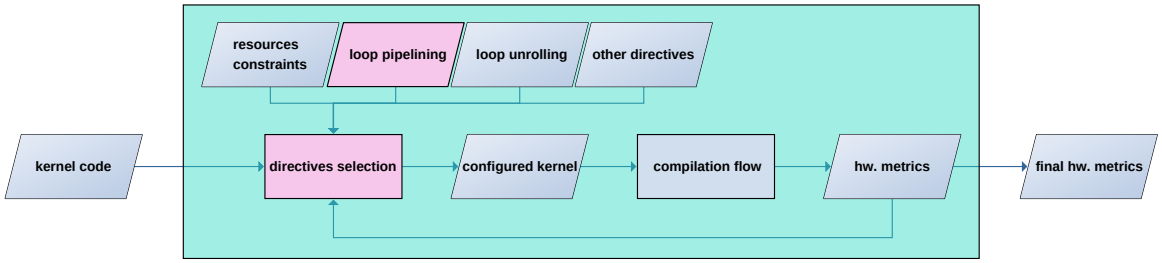
In Figure 3, the local DSE is performed by selecting between appropriated values for the available directives (namely “a configuration”), then the configured design is compiled,

Figure 2 – Global DSE flow overflow.



Source: Elaborated by the author.

Figure 3 – Local DSE flow overview.



Source: Elaborated by the author.

generating the hardware metrics. The hardware metrics can be estimated at high-level or at Hardware Description Language (HDL) level, or can be actually measured after the design full implementation. The hardware metrics are usually used as feedback to the local DSE methods to help deciding new and better configurations iteratively.

Thus, a fast local DSE is crucial to make the global DSE feasible. Thus we focus the contributions of this thesis in exploring and accelerating the local DSE.

The boxes highlighted in pink, in Figure 3, represent points where this thesis focus its efforts. We identified that the loop pipelining directive is a bottleneck that can make the local DSE inviable, and propose two new loop pipelining techniques that reduce the asymptotic computation time, removing it as a local DSE bottleneck. We propose a “directives selection” paradigm shift to improve the local DSE speed, and precision, which is demonstrated by means of a new proposed approached to explore the resources constraints, loop pipelining, and loop unrolling directives.

The rest of this thesis is organized as follows:

Chapter 2 presents a detailed bibliographic review and related works on the code partition problem, DSE methods, hardware metrics estimation, and loops directives. This review allows us to highlight the open problems in the area, to which we focus our efforts.

Chapter 3 presents a quantization of the design space provided by the main HLS tools, which is necessary to understand the size and nature of the DSE problem. The quantization reveals that loop pipelining is a HLS bottleneck that can make it impracticable.

Chapter 4 presents a review on the state-of-the-art methods to create loop pipelines and present two new methods to speed-up loop pipelining.

Chapter 5 presents a comparison between the related works on local DSE and a new proposed “white-box” paradigm for developing methods to accelerate it. Furthermore, we present new “white-box”-based exploration methods for resources constraints, loop pipelining and loop unrolling.

Finally, Chapter 6 concludes this thesis.

CHAPTER

2

RELATED WORKS

As introduced in Chapter 1, global DSE is composed of several sub-areas. The goal of this chapter is to present a literature review on each of the areas, highlighting the state-of-the-art for each one and the open problems.

The first sub-area is the code partitioning problem, which consists of, given a code, deciding which snippets are the most suitable/adequate to implement accelerators. Section 2.1 presents the related works on code partitioning.

The second sub-area is the local DSE problem, which consists in, given a code snippet, deciding which directives will result in the best hardware metrics combinations. This problem is usually a multi-objective one, since FPGA implementations have a trade-off between hardware resources usage and design performance. As such, local DSE tries to find a set of designs that result in the best balancing between resources usage and design speed. Section 2.2 presents the related works on local DSE.

The third sub-area is the hardware metrics estimation problem, which consists in, given a code snippet and a set of directives, estimating the hardware resources usage of the final design, maximum frequency, energy consumption and "speed" (clock cycles, latency, or execution time). The hardware metrics estimation is an important step to allow a fast local DSE since good estimations allow to skip the time-consuming hardware synthesis, mapping and routing. Section 2.3 presents the related works on metrics estimation.

2.1. Related Works on Code Partitioning

This section presents the works in literature focusing specifically on the code partitioning problem, aiming to identify why global DSE is still an open problem. We are not analysing the methods and its results since this is out of this thesis scope.

As stated in Chapter 1, the local DSE process is often not-viable due to the time-

consuming steps involved and the large design space formed by the directives. As such, many approaches in the literature avoid adopting local DSE to enable the global DSE by using a single high-level estimation performance, ignoring the possible hardware implementation of each kernel code.

Examples of approaches that fit in this category are: Guo *et al.* 2005, and Castro *et al.* 2015, which use GPP profile execution frequency for loop selection; Sotomayor *et al.* 2015, which use GPP profiling to select candidate loops for later user selection; Oppermann and Koch 2016, which uses a HLS execution model for loop selection.

Another common way to avoid the need of local DSE is to compile the kernel to a particular architecture, thus eliminating the huge design spaces when targeting FPGA accelerators. These approaches map all candidate loops into accelerators, also removing the trade-offs selection in the global DSE.

Examples of approaches that fit in this category are: Kenter, Vaz and Plessl 2014, which selects all vectorizable loops and maps them to a "vector personality"; Vandierendonck, Rul and Bosschere 2010, which finds parallel loop regions and maps them to a threaded architecture; Matai *et al.* 2016, which selects kernels based on templates matching.

All works in this scope agree that the bulk of the computations are located within the code loops, focusing only on loops selection.

There are many approaches in the literature attacking the partition problem, also called "hardware-software co-design", focusing on specific applications. However, such works are out of the scope of this thesis.

All works presented in this section cannot be said to deal with the global DSE problem as stated in Chapter 1 since they use methods to eliminate the kernels local DSE. The usage of such methods are justified and required due to the lack of efficient local DSE methods. Furthermore, these works highlight the global DSE dependency on accelerator performance knowledge.

We can conclude that local DSE and the ability to evaluate the accelerators performance are requirements to allow global DSE code partition, motivating the rest of this thesis.

2.2. Related Works on Local DSE

This section summarizes related works on accelerating local DSE for HLS FPGA designs, highlighting the approach used in order to define where are the best opportunities and methods to do so.

Schafer, Takenaka and Wakabayashi 2009 Presents the Adaptive Simulated Annealing DSE (ASA). The method uses two parameters to weight the design speed and hardware resources

usage, and varies the parameters. The synthesized hardware metrics are used to create statistical models to decide the configuration values for the method's next iteration.

Schafer and Wakabayashi 2012 presents the Machine Learning Predictive DSE (MLP). The method starts generating and synthesizing random design configurations, until reach a threshold, when it creates a ML-based model to create hardware metrics estimations. A Genetic Algorithm (GA) is used to explore more designs with the model.

Schafer and Wakabayashi 2012 presents the Divide and Conquer Exploration Algorithm (DC-ExpA), which is based on two previous works: ASA, which is a slow convergence method, but finds good solutions, and the Clustering Design Space Exploration Acceleration (Schafer and Wakabayashi 2010), which is focused on quickly finding sub-optimal Pareto curves.

DC-ExpA uses the CyberWorkBench compiler (which we will refer by "Cyber") (Wakabayashi and Schafer 2008), and uses synthesis results to feed back the DSE. The DSE is made in an hierarchical approach, where the code is parsed into a tree of $(loops, functions, arrays)$ and all the leaves are explored first and the other nodes are recursively explored in a children to parent order.

The design space reduction is achieved since only subspace of different code partitions is explored. Furthermore, each partition has a specific type (loop, function, array) allowing only a subset of directives to be applied.

Liu and Carloni 2013 presents the "Transductive Experimental Design" approach, which consists in a methodology to compare and select the best DSE learning-based methods in the literature.

First the design space size is highlighted in order to define groups for directives, called knobs. Then, the Average Distance from Reference Set (ADRS) is used to measure the quality of each analysed DSE method. The training set for each method is defined as a subset of all directives. And a exhaustive search is performed over the design space formed by the directives subset.

Pouchet *et al.* 2013 presents the Polyhedral Based Data-Reuse Optimization (PBDRO), a framework to explore data reuse through pre-fetching, task-level parallelization, and loop transformations as unroll, and pipeline. To optimize the problem, PBDRO uses "Tiling Hyperplanes" (Bondhugula *et al.* 2008).

The search space is defined by the cross-product of arrays buffer size, loops, and loops required bandwidth. Note that this work considers only loops with static bounds.

PBDRO uses ROSE, a source-to-source compiler and ISL/CLoog polyhedral library. AutoESL is used to compile the final code to execute on Convey HC-1 platform. The

solutions are synthesized to give results and no method for selecting a subset of the design space is used.

Shao *et al.* 2014 presents pre-RTL, power performance simulator for rapid accelerator-centric DSE (Aladdin), which estimates power and performance of accelerators without synthesis. Aladdin takes the input C code, calculates the "Dynamic Data Dependence Graph" and constraints it to fit the available resources. Then is performs the metrics estimation.

Aladdin uses the ILDJIT Intermediate Representation (IR) compiler to construct the "Dynamic Data Dependence Graph". All loops are assumed to be fully unrolled and pipelined. The DSE is made by enumerating the designs, but the approach focus on models to estimate hardware metrics as acceleration source.

Prost-Boucle, Muller and Rousseau 2014 presents Standalone Design Space Exploration (SDSE), which is a DSE heuristic that can be implemented along-with any HLS tool.

The DSE is based on evaluating a purely sequential initial design. Then a parallel design is created by applying a set of rules for adding additional computing operators, wiring conditionals, unroll loops, add ports to memories, and replacing memory banks for registers. Finally a set of transformations is chosen to generate the final solution.

SDSE is implemented in AUGH, which estimation results are used to feedback the method, while the designs are fully implemented with Vivado. Only transformations that increase parallelism are applied, what generally increases the circuit area, restricting the exploration towards faster-and-larger designs only.

Fernando *et al.* 2015 presents Accelerator Synthesis using Algorithmic Skeletons for rapid DSE ($(AS)^2$). The approach tries to identify an "algorithm skeleton" for the given kernel code, which contains a directives set to be applied. Note that several skeletons must be compatible with the same code, for example, skeletons with different unroll factors for loops. Skeletons are provided by the user in a library-like fashion. The skeleton selection method is presented in Nugteren and Corporaal 2014.

The DSE space is mitigated since the number of possible skeletons are usually smaller than the design space set made by combining the possible directives.

Zuo *et al.* 2015 presents the Polyhedral-based SystemC framework for Effective Low-Power DSE (PSCEFP), which compiles `C` affine loops to SystemC before performing the DSE. PolyOpt is used to optimized the `C` code memory accesses before the compilation, and a model is used to estimate the design power and latency.

A logarithmic sampling is made over the design space, and all points are synthesized. The synthesis results are used to interpolate equations that estimate the hardware metrics, and such estimations are used to select the best designs.

Schafer 2016 presents the Probabilistic MultiKnob HLS DSE (PMK), which classifies directives in the following three "knobs": The "Global Synthesis Options", which is applied to the whole design by the compiler; The "Local Synthesis Directives", which are `pragmas` to fine control the behavioural description; And the "Functional Units and Number Types", that controls the Number of Functional Unitys (FUs) (e.g. fixed/floating points).

PMK explores the Local Synthesis Directives (local knob) and Functional Units and Number Types (FU knob). The knobs non-complete orthogonality is observed, implying that the same design can be achieved using transformations from different knobs.

PMK also focus on Application-Specific Integrated Circuits (ASICs), where the assumptions that "The area decreases when resource sharing increases" and "the smallest design is always the design with only a single FU for each unique FU type" are not true, as they are for FPGA designs.

The main idea behind the approach is to first perform a local knob search, but focusing in allocating as many resources are needed to achieve the fastest design, called D_{max}. Then, a FU knob optimization is performed to obtain the design with minimal resources, called D_{min}. The area defined by the pair $M_i = (D_{max}, D_{min})$ is defined as the probability of finding optimal designs.

The search space is significantly reduced by exploring one knob at a time, resulting in a faster DSE.

Silva and Bampi 2015 presents Area Oriented DSE (AO-DSE), which is based on an equation that estimates the area and speed-up gradients between designs. Prost-Boucle, Muller and Rousseau 2014 and Silva and Bampi 2015 belong to the family of "Gradient-based pruned searching" DSE.

AO-DSE considers only the speed-up/area-increment coefficient, eliminating the need of user defined weights, making it a totally automatic approach. Results show an improvement in the number of registers used, without affecting other hardware metrics for the only FIR filter it has been tested on.

No results are given on the DSE number of designs, or their quality. And the register problem is also not discussed.

Xydis *et al.* 2015 presents Spectral-Aware Pareto Iterative Refinement Optimization for Supervised HLS (SPIRIT), a method based on iterative Pareto-curve refinement. The idea behind this approach is to create a signal $s[x] = area(x)latency(x)$ (time-area product), where x is the HLS directives configuration, and use spectral analysis on $s[x]$, to identify high-randomness regions. Once those regions are identified, more points are compiled to improve the model.

First, a Random Surface Model (ML) is trained with a set of randomly generated designs. The model is used to create estimations for all designs points, allowing to calculate the signal $s[x]$ for the whole design space.

The iterative refinement starts by selecting the predicted Pareto-points, which are synthesized. Then the power-spectral analysis are used to derive how many new random should be evaluated. After synthesized, $s[x]$ is calculated and the designs are added to the RSM training set. Then the algorithm iterates.

Results show that the accuracy is improved when compared with other ML-based DSE methods, while the number of synthesized designs is comparable to the other method's.

Liu and Schafer 2016 presents the Efficient and Reliable HLS DSE (ERDSE), an extension of Schafer and Wakabayashi 2012. First the GA method (Schafer and Wakabayashi 2012) is used to select and synthesize few points in the design space. Then, a Machine Learning (ML) method generates many un-synthesized designs, which are divided into windows, and pruned. Finally, the un-pruned designs are selected to be further synthesized.

Results compare the method with exhaustive synthesis search and with synthesizing only the optimal results given by the HLS estimation exhaustive search.

Zhong *et al.* 2016 presents Fast and Accurate FPGA Performance Estimation and DSE (Lin-Analyzer), an approach to accelerate the local DSE based on dynamic run-time analysis to accurately predict the accelerator performance (execution time).

First the code is instrumented to generate an execution profile trace. Then the code is optimized and scheduled with resource constraints and designer given `pragmas`. A list-scheduling is used to schedule the loop traces, resulting in the total number of cycle for loop completion, which is the only measure used in the DSE.

The DSE is made exhaustively and the fastest configuration is returned, being the clock cycles estimation the only source of speed-up.

Zhong *et al.* 2017 presents High-Level Analysis Framework for Fine- and Coarse-Grained Parallelism on FPGAs (MPSeeker), an extension of Lin-Analyzer, which includes a Gradient Boosted Machine Learning (ML) method to estimate hardware resources usage, allowing a DSE that explores the resource/speed balancing.

The ML is trained with 80% of the full designs space, which is a drawback since many configurations must be synthesized in order to train the method, even though the training itself is takes only few minutes.

The DSE makes an exhaustive search and returns the fastest configuration which uses less resources.

Cong *et al.* 2018 presents Automated Accelerator Generation and Optimization with Composable, Parallel and Pipeline Architecture (AutoAccel), which uses a micro-architecture template to efficiently implement codes with parallel and pipeline computations.

The computation model is used to derive the hardware metrics estimations and allow a precise estimation considering loop unrolling, pipelining and memory buffer sizes.

A set of strategies are used to reduce the design space. Between them: Small loop flatten; Loop Unroll factor pruning, which limits the unrolling factor based on the BRAM usage; Saddleback Search for Unroll Factors, which limits the unrolling factor based on the trip count of the 2 loops with highest trip count; Fine-Grained Pipeline pruning, which decides either to apply loop pipelining or not based on hardware metrics before and after the pipeline; Power-of-two buffer bit-widths and capacities, that reduces the design space by allowing only power of 2 bit-widths and buffer sizes.

No DSE times are presented, and it is not clear if only estimations results are used in the DSE (further than the two initial synthesis for fixed values).

Roozmeh and Lavagno 2018 presents Multi-Kernel DSE (MKDSE), an approach to model multiple kernels communicating over the AXI (Azarkhish *et al.* 2015) interconnect. The number of clock cycles is measured using simulation, while the method used to estimate hardware metrics is not reported.

The DSE is divided in two parts. In the first part, kernels are optimized varying loop pipelining, unrolling, and array partitioning. In the second part, all kernels are added in the same design, and memory transfer metrics are measured through simulation. Finally, a quality metric is evaluated for all configurations, allowing to select the most efficient one.

The DSE speed-up is achieved by exploring the directives separately for each kernel.

Ferretti, Ansaloni and Pozzi 2018 presents Lattice-Based DSE (LBDSE), which is based on the observation that Pareto optimal points share a low variance between their configurations. The idea behind this work is that there the correlation between design and configuration spaces is large in a $area \times cycles$ space, however, when applying Principal Component Analysis, the Pareto-optimal points tend to cluster together in the configuration space.

The approach first chooses points using an U-shaped distribution (Ferretti, Ansaloni and Pozzi 2018), which are synthesized to create a first Pareto approximation. Then, the Pareto points between the synthesized ones are chosen and their σ-neighbours in the lattice space are selected to be synthesized. Then, the algorithm iterates.

Cong *et al.* 2018 presents a set of five "Best-programming" guidelines that allow to easily improve an HLS hardware accelerator performance, thus, avoiding the time consuming DSE from the hardware development flow.

The methodology iteratively analyses and identifies the design bottleneck at each iteration, and applies a small HLS optimizations set to enhance the performance.

The best-practices are: First, explicit data caching, which is done by batch processing or data tiling; Second, parallelism exploration, which is done by creating custom pipelines and processing units duplication. Third, data movement acceleration, which improves the throughput by creating double buffers and re-organizing the scratch-pad memories (Cong *et al.* 2017).

All works presented in this section are motivated by the design space size, which is commonly recognized as a combinatorial problem unsuitable to be searched exhaustively. Between the woks presented in this section, we can recognize the following main ways to accelerate the DSE:

"Smart" selection: These approaches handle the relation between configuration and design space as a "black-box" and apply a ML method to infer information and help the selection. The ML method is often trained using data from a synthesized design space sub-set. However, as observed in most approaches, such relations are complex and unpredictable, what motivated the usage of more elaborated ML methods.

Modeling: These approaches create an explicit model to relate the configuration and design spaces. The model is used to estimate quickly the design space and use such results to choose the best configurations. The models can be built using mathematical models, equations, or regressions using data from a synthesized design space sub-set. However, these approaches also suffer from the design space unpredictability, what makes the models inaccurate.

Architecture Fitting: These approaches bind the code implementation to an specific target architecture. As such, they can reduce the number of directives to be searched and also create better estimation models. However, this approach is target/compiler/platform dependent and can not be easily re-targeted.

Chart 1 shows how the works presented in this section fit with the above-mentioned classification.

Chart 1 shows that many approaches combine the "smart" selection and modelling in order to maximize the DSE speed-up. This is necessary since even a single full synthesis takes from minutes to hours (depending on the design and FPGA sizes), what can render "smart" selection only approaches not viable for most cases. As such, the hardware metrics estimation is a crucial step in the local DSE.

Chart 1 – Related works on local DSE classification according to the speed-up method used.

Work	ASA	MLP	DC-ExpA	PBDRO	Aladdin	SDSE	$(AS)^2$	PSCEFP	PMK	Lin-Analyzer	MPSeeker	AutoAccel	AO-DSE	SPIRIT	ERDSE	MKDSE	LBDSE
"Smart" Selection	✓	✓	✓	-	-	✓	-	-	✓	-	✓	-	-	✓	✓	-	✓
Modelling	-	✓	-	-	✓	✓	-	✓	✓	✓	✓	-	✓	✓	-	-	-
Architecture Fitting	-	-	-	-	-	-	✓	-	-	-	-	✓	-	-	-	✓	-

Source: Elaborated by the author.

2.3. Related Works on Hardware Metrics Estimations

As pointed in Section 2.2, the hardware metrics estimation is a key feature to enable a fast and precise DSE. This section presents related works on hardware metrics estimation aiming to highlight their usability and precision, which will help to decide which methods can be used in the future steps of this thesis. From the literature, we can highlight two main categories for hardware metrics estimation methods.

The first category is estimating the metrics from the HLS output HDL code, which is compatible with most HLS tools and usually generate more accurate results since it contains architecture and platform specific details. However, the HDL description is generally large, and all specific information can impact the estimation time.

Examples of approaches that fit into this category are: Schumacher and Jha 2008, which creates a components interpolation estimation and models the synthesis tools optimizations to improve the final estimation quality; Dwivedi *et al.* 2006, which estimated the design timing using a hierarchical rounding approach; Aung, Lam and Srikanthan 2015, which estimates the Digital Signal Processors (DSPs) usage based using a mathematical model based on multipliers synthesis results;

These approaches aim to create estimations for one metric each, what is justifiable since each metric requires a unique estimation method. As such, these approaches are interesting for the synthesis tools and fall out of this thesis scope.

The second category is estimating the metrics from the HLS code. These approaches generally create an interpolation or approximation formula based on the synthesis results of smaller codes or functional units. However, such approaches cannot model information on the further compilation and synthesis steps, resulting in larger errors. Furthermore, these approaches may require many synthesis runs to train or interpolate the model, which constraint the performance or make the model not portable.

Examples of approaches that fit in this category are: Vahid and Gajski 1995, which sums the components hardware resources usage to compose the design estimation; Henkel and Ernst 1998, which uses scheduling and interpolation models to estimate the hardware resources and communication costs; Nayak *et al.* 2002, which reduces and synthesizes each component and creates a proportional model to compose the design estimation; Enzler *et al.* 2000, Kulkarni *et al.* 2002, Kulkarni *et al.* 2006, Jiang, Tang and Banerjee 2006, Deng *et al.* 2011, and Wang, Liang and Zhang 2017, which interpolate a model based on components synthesis results; Marugán, González-Bayón and Espeso 2011, which estimates the computation time based on the scheduling information; Okina *et al.* 2016, which estimates the power performance based on a one-time synthesis results; Silva *et al.* 2013 and Silva *et al.* 2014, which estimate the latency and computation time based on a roof-line model

Also, the HLS tools offer their estimations. As examples, Altera OpenCL provides hardware resources estimation for OpenCL kernels (Czajkowski *et al.* 2012); Xilinx SDSoC and SDAccel provide estimations for kernel metrics (Xilinx 2016, Xilinx 2016).

The HLS estimations are bounded to the little knowledge they have about the platform and architecture. Without information about the optimizations used by the synthesis tools, platform architecture, mapping and routing, they are incapable of creating precise estimations.

As such, the first estimations type are slow and not portable, being often uninteresting in a HLS scope, and the second estimations type have limited estimation capabilities, what insert uncertainties in a DSE scope, leaving HLS DSE techniques lacking on good predictions models.

O'Neal and Brisk 2018 identifies the same problem and states:

"Thus, DSE techniques lack on good predictions model to enable its potential in the FPGA world." (O'Neal and Brisk 2018)

The hardware metrics estimation is out of the scope of this project. Thus, this thesis does not consider creating new methods for the problem or implement a method from the literature. As such, we are left to use the HLS tools or the synthesis tools estimations, which is a trade-off between estimation speed and precision. In order to evaluate the HLS tools estimations viability, Section 2.3.1 presents tests that measure the precision and computation time to generate the estimations for two HLS tools.

2.3.1. HLS Tools Estimation Precision

In this section, we present tests to highlight HLS tools estimations precision and time. As pointed in Kulkarni *et al.* 2002, these estimations often are unable to consider many compilation steps, making them imprecise or incapable of estimating specific metrics as the maximum clock frequency.

Next, we present tests on code constructs for which HLS tools failed to perform well.

Applications with such constructs will serve as benchmarks to test the estimations precision. The reason for using such applications is to avoid having good results which do not reflect a general case.

Data Dependencies: Darte 2000 presents a study on the loop fusion (loop merge) problem complexity. The loop body is separated in its strongly connected components, and classifies loop carried dependencies as sequential computations, what degrades the generated hardware performance.

Memory Access Pattern: Weinberg *et al.* 2005 and Murphy and Kogge 2007 present metrics to measure spatial and temporal locality on benchmarks. Code emulation and cache simulation is used to demonstrate how the lack of suck localities degrade the hardware performance.

Jang *et al.* 2011 and Che, Sheaffer and Skadron 2011 classify memory access patterns that allow vectorization during compilation. Results show that these non-trivial memory access patterns degrade the code performance if not vectorized.

Chen *et al.* 2006 and Jang *et al.* 2011 presents a dynamically scratch-pad memory manager that considers regular, irregular and indirect memory access patterns. Results show that HLS tools generate conservative hardware for such accesses, degrading the performance.

Cilardo and Gallo 2015 shows how HLS tools struggle to generate efficient memory bank partitions for multidimensional arrays when data access is not completely decoupled at compilation time. A polyhedral formulation is proposed to generate an efficient number of banks and a respective partition order.

Zhao *et al.* 2016 shows that complex data structures as “priority queue”, “dynamically sized stacks”, “random decision trees”, and “hash tables” generally are poorly implemented by HLS tools, since its manipulation involves loops with variable-bounds, array index data dependencies, and variable size arrays. The solution proposed is based on implementing an specialized container unit, which decouples and hide his operations from the computation code. The container unit is implemented based on a template that allows pipeline on different channels for reading and modifying the complex data structure.

Loop Bounds: Liu, Bayliss and Constantinides 2015 presents an off-line synthesis of an on-line dependency tester for HLS loop pipelining, which proposes an alternative for synthesizing loops with unknown dependencies at compiler time. Results show HLS compilers generate conservative hardware for such constructions, degrading the final performance.

The works presented show code constructs for which HLS tools do not infer the most efficient hardware, and also highlight examples that expose such code constructs. Next, Table 1 presents the selected benchmarks that expose these characteristics and will serve as a base for testing the HLS tools estimation precision later.

Table 1 – Selected applications for which HLS tools infer inefficient hardware to be used as benchmarks for testing the HLS tools estimations time and precision.

Algorithm	Used on	benchmark	bounds	# loops	# nests	arrays size
row col	(Liu, Bayliss and Constantinides 2015)	Xilinx	dynamic	2	2	100
typ loop	(Liu, Bayliss and Constantinides 2015)	none	dynamic	2	2	100
jacobi 2d	(Liu, Bayliss and Constantinides 2015)	Polybench	static	3	3	100
add int	(Liu, Bayliss and Constantinides 2015)	LivemoreC	static	3	3	100
fft	none		static	3	3	1024
gauss pivotin	(Liu, Bayliss and Constantinides 2015)	MIT (MIT 2016)	dynamic	3	3	unknown
correlation	(Zuo *et al.* 2015)	Polybench	dynamic	9	3	500
covariance	(Zuo *et al.* 2015)	Polybench	dynamic	7	3	500
sha	(Prost-Boucle, Muller and Rousseau 2014)	CHStone	data dependent	12	3	8192
sha	(Prost-Boucle, Muller and Rousseau 2014)	(Agarwal, Ng and Arvind 2010)	static	7	2	1024

Source: Elaborated by the author.

The benchmarks on Table 1 were compiled with the Intel OpenCL Framework for OpenCL (IOC) and LegUp HLS Compiler (LUP), which are representatives of paid and academic HLS tools. Table 2 presents the time for each tool create the hardware metrics estimations and their precision when compared with the synthesized hardware.

Table 2 IOC is slower to make the estimations, what is expected since more compilation steps are made and the design is larger since it includes the Peripheral Component Interconnect Express (PCIe) interface in the kernel design. The longer time taken by the **fft** benchmark is due to "aggressive memory optimization" performed by IOC.

As a counter part, IOC is more precise and can estimate more metrics than LUP, what is expected since LUP does not consider resources constraints, binding, register allocation, and all synthesis steps(Canis *et al.* 2013, Kulkarni *et al.* 2002, Kulkarni *et al.* 2006). Nevertheless, we can observe no correlation between the estimation precision and the code constructs presented in this section.

As such, we can conclude that the estimations made by the HLS tools are too imprecise and are not much faster than the estimations given by the synthesis tools (generally obtained

Table 2 – Estimation and precision of HLS tools for Table 1 benchmarks.

Algorithm	IOC		LUP	
	time (s)	ALUT, FF, RAM, DSP ($\frac{estimation}{synthesis}$)	time (s)	ALUT, Reg., DSP ($\frac{estimation}{synthesis}$)
row_col	6.75	$\frac{7184}{12818}, \frac{22092}{21165}, \frac{155}{135}, \frac{0}{0}$	1.14	$\frac{2627}{2264}, \frac{1961}{2455}, \frac{8}{8}$
typ_loop	fail	fail	0.83	$\frac{822}{966}, \frac{1544}{693}, \frac{8}{12}$
jacobi_2d	6.98	$\frac{8336}{17222}, \frac{19172}{27257}, \frac{84}{137}, \frac{10}{10}$	1.18	$\frac{6363}{4386}, \frac{2409}{5363}, \frac{12}{12}$
add_int	20.41	$\frac{30149}{41544}, \frac{60123}{60458}, \frac{224}{207}, \frac{21}{21}$	1.25	$\frac{27924}{8508}, \frac{5993}{10652}, \frac{33}{5}$
fft	312.33	$\frac{11466}{16688}, \frac{29932}{35144}, \frac{147}{137}, \frac{8}{8}$	3.14	$\frac{806}{1822}, \frac{13946}{1146}, \frac{4}{2}$
gauss_pivotin	11.21	$\frac{28793}{44763}, \frac{92327}{43692}, \frac{504}{439}, \frac{46}{52}$	1.26	$\frac{35354}{22707}, \frac{5776}{19828}, \frac{24}{16}$
correlation	10.78	$\frac{31169}{32960}, \frac{89184}{69066}, \frac{562}{287}, \frac{36}{36}$	1.33	$\frac{38611}{12302}, \frac{6850}{14509}, \frac{13}{24}$
covariance	6.91	$\frac{20116}{21356}, \frac{57965}{45162}, \frac{341}{194}, \frac{16}{16}$	1.34	$\frac{11355}{8440}, \frac{3025}{7446}, \frac{12}{16}$
sha	15.67	$\frac{37634}{72191}, \frac{105992}{117227}, \frac{747}{409}, \frac{2}{2}$	1.03	$\frac{6116}{7544}, \frac{8886}{7119}, \frac{4}{2}$
geomean	15.82	0.67, 1.09, 1.25, 0.98	1.29	1.31, 0.90, 1.30

Source: Research data.

with seconds to minutes) to justify their usage.

2.4. Final Remarks on Related Works

In this chapter, we highlighted the related works on the three sub-areas involved in this project: code partition, local DSE, and hardware metrics estimation.

On code partitioning, the literature works indicate that global DSE methods are being held back due to the lack of fast local DSE methods. On local DSE, the literature shows that a fast local DSE requires “smart” decision methods and hardware metric estimation, to reduce and speed-up the evaluated designs. On hardware metrics estimation, the literature shows that HLS estimation methods are limited and have low precision, as demonstrated by our tests, while HDL estimation methods are more time-consuming and target specific.

This thesis will focus on the local DSE problem, for which improvements are necessary and key to enable the global DSE for future generations HLS tools. As such, our next step is to present an evaluation of the design space size aiming to identify which are the most important and impactful directives.

CHAPTER

3

DESIGN SPACE EVALUATION

As shown the Chapter 2, a fast local DSE is key to allow the global DSE. Works in the area focus to reduce the design space and use estimations to speed-up the DSE process. However both approaches can not explore the relations between the design and configuration spaces, leading to inaccuracies.

All local DSE works presented in Section 2.2 handle such relation as a “black-box” approach. In other words, to the best of our knowledge, there is no previous works that analyse the directives effects in the design to extract information that can be useful to accelerate the DSE.

Thus, the goal of this section is to evaluate the local DSE directives for the main HLS tools, and also to evaluate the design-space size and complexity. Such study can determine which are the most important and impactful directives. Section 3.1 presents the main HLS tools available, both commercial and academic, and also the resources they offer in order to analyse the kernel design space size.

3.1. Kernel Design Space Quantification

In this section, we analyse the main HLS tools available and the design space offered by a representative set of them. Then, an evaluation of the directives is presented, allowing to analyse the design space nature and to evaluate its size and complexity formally.

The design space size problem is present in all HLS tools, formed by the many directives they offer to the user to control how the high-level code should be mapped to the FPGA fabric. Such control is desired by the user since a high-level language is much more abstract than the hardware implementation, hiding low-level and architecture dependent operations that must be specified in the hardware implementation.

Directives can be `pragmas`, compiler `optimizations`, or even `code styles`. As such, the number of options for a kernel implementation depends on the number of directives offered

by the HLS tools.

3.1.1. HLS Tools

Chart 2 list HLS tools used in the literature with their characteristics. Our goal is to select a subset of those tools that contains a representative set of directives and also does not bind us to a specific input language or output. Unknown values on Chart 2 mean that we were not able to test the tool or to obtain information about it due to licenses or code unavailability.

From Chart 2, we decided to further analyse the design space size of Bambu HLS Compiler (BBO), IOC, LUP and Xilinx Vivado HLS Compiler (VVD). BBO and LUP are academic open-source tools, allowing us to access and modify features inside the tools. IOC and VVD are the most common commercial tools. Furthermore, LUP, IOC and VVD use the same framework, what makes developments on LUP easier to be incorporated into the other tool-flows. BBO has the GNU compiler Collection (GCC), which lacks documentation and a well structured framework disadvantage (for the GCC version 4.7), making its learning curve more time consuming.

3.1.2. Directives

This section lists the main directives for each BBO, IOC, LUP and VVD, in order to evaluate the kernel design space size for each tool. Charts 11 to 15, in Appendix A, show the most relevant directives offered by the compilers from Chart 2, where we can see that state-of-art commercial tools offer several options, while academic tools offer less options. The directives can be subdivided into three sub-groups, according to how they are used and the scope they act, as listed next:

- Macros or programming language constructs (`pragmas`),
- Programming language style,
- Compiler options.

Macros and programming language constructs are generally lines where the designer adds information, without changing the source code logic. Compilers that get information from the Programming language style require that the designer change the source code in a way the compiler will recognize the features. Finally, compiler options are a less common way to control how the compiler will map the algorithm into hardware, since most of the compiler options are applied on the whole code, denying the designer the capability to control precisely where each feature should be applied.

Chart 11 shows the BBO directives. BBO focus its directives and optimizations around the scheduler, what is the most impactful optimization due to BBO hardware model (Pilato and

Chart 2 – HLS tools summary.

Acronym	Tool Name	Owner	License	Frame-work	Input	Output	Est. Time
AGT	Agility	Mentor Graphics link	paid	N/A	SystemC subset	HDL	unknown
AUG	AUGH	TIMA Lab link	unknown	unknown	N/A	unknown	unknown
BBO	Bamboo	Polimi link	GPL3 (free to use)	GCC	C	HDL	seconds
CTP	CatapultC	Methor Graphics link	paid	N/A	SystemC	HDL	unknown
CTS	Cynthesizer	Cadence link	paid	N/A	SystemC	HDL	unknown
DMC	Dime-C	Nallatech (no link)	paid	N/A	C subset	Proprietary HDL	unknown
DKS	DK Design Suite	Mentor Graphics link	paid	N/A	Handel-C	HDL	unknown
DWV	DWARV	TU. Delft (no link)	unknown	CoSy	C	HDL	unknown
IOC	Intel OpenCL SDK	Intel FPGA link	paid	LLVM	C-OpenCL	proprietary HDL	minutes
IPC	Impulse C	Impulse Accelerated Technologies link	paid	N/A	C	HDL	unknown
LUP	LegUP 4.0	EECE Toronto University link	open source	LLVM	C	HDL	seconds
MHC	Matlab HDL Coder	Mathworks link	paid	N/A	Matlab	HDL	minutes
RCC	ROCCC	Jacquard Computing Inc link	legacy open source	SUIF2	C	HDL	unknown
SPC	Synphony C	Synopsys link	paid	N/A	C	RTL	unknown
STT	Stratus	Cadence	link	Smaltalk	SystemC	HDL	unknown
VVD	Vivado HLS	Xilinx link	paid	LLVM	C / C++	HDL	minutes

Source: Elaborated by the author.

Ferrandi 2012). Furthermore, BBO enables the usage of native GCC options, with 22 options that can impact the memory architecture, but there are no formal studies about how much the performance is affected by those options. Furthermore, most optimizations are specific to BBO, which is not ideal since models developed to them might not be easily applicable to other compilers.

Chart 12 shows the IOC directives. Differently from BBO and LUP, IOC adopts the host-kernel OpenCL architecture, where kernels are mapped into FPGA accelerator and the host code is executed in a GPP. As consequence, most directives are OpenCL specifically related to parallel data processing. Furthermore, IOC vastly explored the possible memories architectures, making use of the in-fabric FPGA memory to improve throughput and parallelism.

Chart 13 shows the LUP directives. LUP uses the Low-Level Virtual Machine (LLVM) optimizations to reduce and simplify the code, and also to control the processing units and memory architecture. LUP transformations are implemented as LLVM passes, making them easily applicable for any LLVM intermediate representation.

Charts 14 and 15 show the VVD directives. VVD makes extensive Intellectual Propertys (IPs) usage, and offers directives to control the IP interfaces and other parameters. VVD also implements several loop optimizations, which allows a rich design space for loops.

Charts 11 to 15 give us an idea of the total number of possible different ways that HLS tools can map a code into hardware. It is important to notice that the observed configuration and design spaces relation unpredictability mentioned in the works presented in Section 2.2 are likely to be a consequence of the directives acting differently over overlapping scopes, creating a non-orthogonal relation which cannot be predicted by regression-based models.

Considering that the source code has n_l **for** loops, and n_f functions, where a loop is defined by the triple $(initialization, condition, iteration)$ and a function defines the duple (n_a, n_s), which are the number of arrays and scalar variables respectively. Considering that the given tool allows N_f, N_a, N_s and N_l options for each function, array, scalar variable and loop respectively. Equation 3.1 presents the possible number of ways that a code can be mapped into hardware considering only inner-most loops and that arrays are transformed parsed into one dimension by the HLS tool.

$$Possibilities = \prod_{if=0}^{n_f} N_f^{if} \times \prod_{ia=0}^{n_a} N_a^{ia} \times \prod_{is=0}^{n_s} N_s^{is} \times \prod_{il=0}^{n_l} N_l^{il} \tag{3.1}$$

Note that, if functions are inlined, the number of arrays and scalar variables within the function are multiplied by the number of functions calls. We will assume that, if a directive is set for either an array or variable in a function, this directive will be reproduced in all inlined calls of that function. Thus, Equation 3.1 considers the directives for arrays and variables only once per function.

The values for N_f, N_a, N_s and N_l can be evaluated from Tables 11 to 15. Furthermore, we have to consider that some directives on Tables 11 to 15 are exclusive and other have dependences.

To calculate the N_l values, we will consider only loops in the "normal form" $(i = 0; i < bound; i = i + inc)$, where i is the loop iterator variable and inc an increment. Thus, we can represent loops using the pair $(bound, inc)$. Note that loops with initialization different than zero, or with different conditions can be transformed into the normal form.

To calculate N_a we consider that a function has multiple arrays with different sizes. Let as_i be the size of the i^{th} array of a function, where $i \in (0, \ldots, n_a)$. Thus, if a directive allows $f(as_i)$ options to arrays, the function will have $N_a = \prod_{i=0}^{n_a} f(as_i)$. Similarly, for **for** loops, we have $N_l = \prod_{i=0}^{n_l} f(bound, inc)$.

Chart 3 summarizes the number of directives combinations that can be applied to a single code, considering only the options in Charts 11 to 15. Note that VVD allows the designer to specify the Initiation Interval (II) for loops and functions. Furthermore, LUP and VVD also define the n_o parameter which is the maximum latency that a target should have.

Chart 3 – Number of directives per target for the three main HLS tools.

Tool	Target	Number of Options	Acronym
IOC	Function	$2^2 \times \log_2(16 * 1024)$	ioc_f
	Scalar	3	ioc_s
	Array	$3 \times (2 + 3as_i^3 \times \log_2(as_i))$	ioc_a
	Loop	$\prod_{i=0}^{n_l=4} \left(\frac{bound}{inc}\right)_i \times \log_2\left(\frac{bound}{inc}\right)$	ioc_l
LUP	Function	$2^3 \times (\max(8, n_o))$	lup_f
	Scalar	2	lup_s
	Array	$2^2 \times as_i$	lup_a
	Loop	2^3	lup_l
VVD	Function	$4 \times 2 \times n_o^2 \times (n_f - 1)! \times n_f^3 \times 4II$	vvd_f
	Scalar	1	vvd_s
	Array	$12 \times 4as^4 \times 9\log_2(as) \times 54$	vvd_a
	Loop	$4 \times 8II \times \prod_{i=0}^{n_l=2} \left(4 * 8\frac{bound}{inc}\right) \times 2 + n_o^2 \times 2 \times \frac{(n_l - 1)(n_l - 2)}{2} \times 3$	vvd_l

Source: Elaborated by the author.

It is worth to note that, even if a tool allows only two options per target (function, array,

scalar or loop), the complexity of the design space would have $\mathcal{O}(n^4)$ according to Equation 3.1. Considering the equations on Chart 3, we can see that, for VVD, this complexity grows to $\mathcal{O}\left(n^3 \times nl^{n_l^2}\right)$, due to the **merge** directive that allows $\left(\frac{(n_l-1)(n_l-2)}{2}\right)$ combinations between loops.

With the $\mathcal{O}(n^4)$ complexity of the design space and the fact that, compiling and synthesizing a single design is a time-consuming task, we conclude that examining all possibilities is not a viable approach even using "smart selection"-based DSE.

To evaluate which directives are the most important we present a bibliographic review highlighting the most common HLS optimizations impacts.

We focus the reviews on loop related directives since they are the most impactful ones in the HLS context (Huang *et al.* 2013), what is caused since optimizing loops means to optimize the computation bulk of most codes (as observed in Section 2.1).

Loop Unrolling: Allows parallel implementation on non-dependent loop operations (Buyukkurt, Guo and Najjar 2006), improve data locality (So, Hall and Diniz 2002), and have resources increase almost linear with the unroll factor (Buyukkurt, Guo and Najjar 2006). Hardware resources usage does not increase linearly due to extra control creation (Cardoso and Diniz 2004). Loop unrolling is one of the most important transformations since they allow to adjust the loop body size accordingly to the FPGA resources capabilities, as well the design throughput and data consumption rate (Cardoso and Diniz 2004).

Loop Pipelining: This transformation reorders the loop inner computations to reuse data in a straight forward flow. By reordering the loop dependencies, each loop iteration can be started every few clock cycles, reducing the number of cycles to complete the loop execution (Gokhale and Stone 1998). This transformation changes the loop schedule, strongly impacts the number of cycles for the loop computation, and lightly impacts the design metrics.

Loop Tiling: replicates the instances of a loop Basic Block (BB) to increase functional parallelism, but requires Multiplexerss (MUXs) and De-Multiplexerss (DEMUXs) to separate and assemble the tiles inputs and outputs. The extra control needed for the tiles usually introduces a smaller overhead than the one introduced by loop unrolling (Saggese *et al.* 2003). This transformation impacts the data consumption rate and throughput directly, allowing to adjust the design accordingly to the FPGA resources capabilities.

Loop Merging: merges two loops with the same iteration space, respecting data dependencies. It can improve temporal locality and allow to create loops with more appropriated size for accelerators (Devos *et al.* 2007, Wolf and Lam 1991).

Loop Interchange: changes nested loops iteration orders to improve temporal locality (McKinley, Carr and Tseng 1996) reducing the memory bandwidth needed, saving extra control

and logic (Devos *et al.* 2007). This transformation benefits the designs and helps to save FPGA resources and to improve the throughput. Devos *et al.* 2007 uses the suggestions for Locality Optimization tool (Beyls and D'Hollander 2006) to get suggestions of which loop transformations should be applied and in which order to achieve better performance for GPP executions.

Loop Dissevering: creates partitions of a large loop which would not fit a single FPGA (Cardoso 2003), similar to loop distribution. The partitions are loaded using partial reconfiguration. As for drawbacks, dissevering prevents the creation of pipelines and adds the reconfiguration time overhead.

Bandwidth Reduction: Cong, Zhang and Zou 2012 presents a complete study on loop transformation to reduce bandwidth inside loops, reducing the design internal buffers. Loops are represented in the polyhedral model which is used to calculate the Data Reuse Graph, and a solution is calculated by an Integer Linear Program (ILP). Results show a reduction in Block RAMs (BRAMs) usage without performance impact. Furthermore, the reduced BRAM usage implies an energy consumption reduction.

From the loop directives above mentioned, loop unrolling and pipelining are generic and known by its impact in the hardware metrics McFarland, Parker and Camposano 1990. The bandwidth reduction is also generic and can be applied to any loop, but the optimization is not implemented in any of the available HLS tools. Loop tilling and dissevering cannot be applied for some data-flows, and loop merging and interchange depend on the loop bounds and data dependencies.

Next, Figure 4 presents a small test case to exemplify the time to compile `C` code to HDL using the design space formed by loop unrolling and pipelining for the `adder-chain` benchmark[1] and LUP as HLS tool.

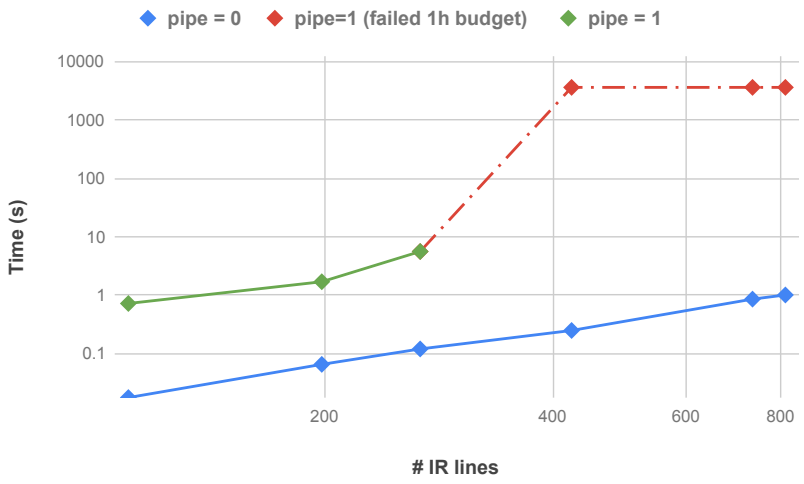
The configuration space is formed by $pipe = \{0,1\}$, representing the loop pipelining optimization turned off or on, respectively, and $uf = \{1,2,3,5,9,10\}$, representing the loop unroll factor. Figure 4 presents the total compilation time in function of the number of LLVM IR code lines (in a log-log scale). The red points indicate that the 1-hour budget was expired before concluding the compilation.

The behavior presented in Figure 4 can be explained as follows:

- The loop unrolling optimizations copies the instructions in the loop body $uf\times$, making the number of IR code lines to grow linearly with uf.

[1] Available at <http://legup.eecg.utoronto.ca/getstarted.php>

Figure 4 – Computation time to compile the `adder-chain` benchmark combining loop unrolling and pipelining.



Source: Research data.

- The compilation without loop pipeline uses the System of Difference Constraints (SDC) scheduler (Canis *et al.* 2013), which solves a constant number of optimization problems, each one with a polynomial time scaling with the number of IR code lines.
- The compilation with loop pipelining uses the SDC Modulo Scheduler (Canis, Brown and Anderson 2014), an iterative algorithm which solves a number of optimization problems that scales with the number of IR code lines.

As such, we can see that the current loop pipelining algorithms are a bottleneck that makes the DSE not viable, since the time to create a pipeline (especially when combined with loop unrolling) is larger than the time needed for hardware metrics estimations, or even design synthesis.

3.2. Final Remarks on Design Space Quantization

This section presented a directives leverage for the mainstream HLS tools. The design space size scaling shows that the local DSE problem is non-trackable when all directives are considered.

As motivated by the global DSE review (Section 2.2) we focus on loop optimization directives, as they are the most impactful ones and are not compiler/platform specific. However, loop pipelining is shown to be a bottleneck in the compilation steps.

As such, Chapter 4 presents our contributions towards speeding up the loop pipelining optimization.

CHAPTER

4

MODULO SCHEDULERS

As noted in Section 3, loop pipelining dominates the HLS compilation time, especially when combined with loop unrolling. Furthermore, loop pipelining is a crucial optimization to achieve high-performance accelerators.

The importance of loop pipelining is given that a loop, without pipeline, takes $total_{cycles} = tc * latency$ clock cycles to complete its execution, where tc is the loop trip count, and $latency$ is defined as how many cycles for one iteration to finish (not to be confused with accelerator latency). A pipelined loop takes $total_{cycles} = II * (tc - 1) + latency$ clock cycles to complete its execution, where II is the initiation interval and $II << latency$.

Loop pipelines can be created using "modulo scheduler" algorithms, and "modulo scheduling" will be used as the synonym for "loop pipelining". The modulo scheduling problem can be formally described as follows:

Definition: We define the number of instructions in the loop body IR code as "`loop size`". We also define the number of variables and constraints in an optimization problem formulation as "`problem size`".

The loop pipelining problem general form includes not only finding a feasible schedule and resource allocation for each instruction in the IR code, but also finding the minimum possible II, since this is the main acceleration factor.

To find the minimum II, current schedulers begin by trying to find a solution for the minimum candidate II, and, in case they fail, the candidate II is incremented and the search repeats. The lower II bound is the maximum value between the recurrence minimum II (*RecMII*) and the resource-constrained minimum II (*ResMII*) (Canis, Brown and Anderson 2014).

The modulo scheduling problem can be divided into the scheduling and allocation parts. The scheduling part of the problem consists of defining the starting time t_i for all instructions i in the loop code, in a way that no dependencies are violated. The congruence class m_i of

instruction i is defined as $m_i = t_i \% II$. The allocation part of the problem consists of choosing which congruence class, and which resource instance, defined as variables m_i and r_i respectively, instruction i will be executed. This information is represented through a Module Reservation Table (MRT), which is a matrix with II rows and $\sum_{k \in R} a_k$ columns, where a_k is the number of instances of resource type k and R is the set of all resources types.

Definition: An MRT (an allocation) is said "valid" if all positions (slots) are assigned to, at maximum, one instruction.

We will consider the modulo scheduling problem for a given II, assuming an outer loop searching for the possible II, as is the case in all state-of-art works.

The rest to this sections is organized as follows:

Section 4.1 presents the SDC Modulo Scheduler, (SDCS) introduced by Canis, Brown and Anderson 2014, which is used by the main HLS tools. Section 4.2 presents the ILP Modulo scheduler (ILPS), introduced by Oppermann *et al.* 2016, which focus on guaranteeing optimal solutions. Section 4.3 presents the Swing Modulo Scheduler (SMS), introduced by Llosa *et al.* 2001, which is a representative of the old list-based modulo schedulers.

Section 4.4 presents the new proposed SDC formulation which is explored by the proposed GA Modulo Scheduler (GAS) and Non-Iterative Modulo Scheduler (NIS) to speed-up the modulo scheduling problem.

Section 4.7 presents results comparing the modulo schedulers and Section 4.8 concludes this chapter.

4.1. SDC Modulo Scheduler (SDCS)

The SDCS algorithm is based on SDC problems, which are a special class of ILP that have a Totally Uni-Modular (TUM) constraints matrix (Camion 1965). Integer problems with TUM matrices are desirable in the optimization field since the associated linear problem solution is also a solution for the integer problem (Dyer and Frieze 1994). As such, an SDC problem is solvable in polynomial time, while generic ILP are solvable in exponential time.

Nevertheless, SDCS is still the HLS bottleneck, as pointed in Chapter 3. The source of the excessive time taken by SDCS is not caused by the SDC problems per-se, but by the iterative nature of the SDCS algorithm, which solves a large number of SDC problems during its execution. This iterative approach is necessary since the SDC formulation cannot represent the allocation part of the problem maintaining the TUM constraints matrix.

An algorithm analysis is carried out next to highlight the number of SDC problems solved by SDCS.

SDCS Algorithm Analysis

Algorithms 1 and 2 present the SDCS algorithm proposed by Canis, Brown and Anderson 2014. Firstly, in Algorithm 1, the non-resource constrained schedule is calculated once, resulting in the As Soon As Possible (ASAP) schedule (line 1), which serves as base values for the instructions time. All instructions are put in a queue to be processed in order (lines 2 to 4). The number of iterations is constrained by the `budget` (line 3), which is defined as $budget = budgetRatio * n$, where n loop size and $budgetRatio$ is a user defined constant. This limit makes the worst-case independent of the instructions queue size,

For each iteration, `solver` is invoked once before backtracking (line 11) and once after the backtracking (line 17). Thus, the number of times the `solver` is invoked within the backtracking function dictates the number of SDC problems solved.

Algorithm 1: Modulo SDC Scheduler(II, Budget) presented in Canis, Brown and Anderson 2014

```
1  Schedule without resource constraints to get ASAP times;
2  SchedQueue ← all resource constrained instructions;
3  while schedQueue not empty and budget ≥ 0 do
4      I ← pop schedQueue;
5      time ← time of I from SDC schedule;
6      if scheduling I at time has no resource conflicts then
7          Add SDC constraint: t_I = time;
8          Update MRT and prevSched for I;
9      else
10         Constrain SDC with GE constraint: t_I ≥ time + 1;
11         Attempt to solve SDC scheduling problem;
12         if LP sover finds feasible schedule then
13             Add I to schedQueue;
14         else
15             Delete new GE constraint;
16             // See Algorithm 2
17             BackTracking(I, time);
18             Solve the SDC scheduling problem;
19     budget ← budget − 1;
20 return success if schedQueue is empty, otherwise fail
```

In Algorithm 2, the input instruction is attempted to be scheduled from its ASAP scheduler time, until its current scheduled time (lines 2 to 4). In the worst-case, the ASAP time will be 0 and its current scheduled time will be the whole schedule length, which, in the worst-case, is a totally sequential path. Therefore, in the worst-case, **solver** will be invoked $\sum_{k=0}^{n} l_k$ times, where l_k is the latency of instruction k. For simplicity, assuming all instructions have a maximum latency of l_{max}, it results in $n \times l_{max}$ **solver** invocations on the backtracking.

Thus, the total number of solver calls is:

$$\mathcal{O}(budgetRatio \times n(2 + n \times l_{max})) = \mathcal{O}(budgetRatio \times n^2) \tag{4.1}$$

Algorithm 2: BackTracking(*II*, *time*) algorithm for SDC scheduler in Algorithm 1 presented in Canis, Brown and Anderson 2014

```
1  BackTracking (II, time)
2  for minTime = ASAP time of I to time do
3      SDC schedule with I at minTime ignoring resources;
4      break if LP solver finds feasible schedule;
5  preveSched ← previous scheduled time for I;
6  if no prevSched or minTime ≥ prevSched then
7      evictTime ← minTime;
8  else
9      evictTime ← minTime + 1;
10 if resource conflict scheduling I at evictTime then
11     evictinst ← instruction at evictTime%II in MRT;
12     Remove all SDC constraints for evictInst;
13     Remove evictInst from MRT;
14 if Dependecy conflict schedulint I at evictTime then
15     forall S in already scheduled instructions do
16         Remove all SDC constraints for S;
17         Remove S from MRT;
18         Add S to schedQueue;
19 Add SDC constraint: t_I = evictTime;
20 Update MRT and prevSched for I;
```

Although, if not constrained by the budget, in the very worst case, to schedule the first instruction, all instructions would be scheduled and unscheduled, leading the backtracking to execute n times. For each attempt to schedule the second instruction ($n-1$ attempts), the first instruction will have to be reschedule, resulting in $n \times (n-1)$ attempts. Continuing this, the worst number of calls `backtracking()` is given by $\sum_{i=1}^{n} \frac{n!}{(n-1)!} = \sum_{i=0}^{n-1} \prod_{j=0}^{i} (n-j)$. Resulting in a number of `solver` calls complexity of:

$$\mathcal{O}\left((2 + n \times l_{max}) \times \left(\sum_{i=1}^{n} \frac{n!}{(n-1)!}\right)\right)$$

According to (Wall 1948), this can be rewritten as:

$$\Rightarrow \mathcal{O}\left((2 + n \times l_{max}) \times (e \times n \times \Gamma(n,1))\right)$$

$$\Rightarrow \mathcal{O}\left(n^2\Gamma(n,1)\right)$$

Where,

$$\Gamma(n,x) = (n-1)!\left(1 - \mathrm{e}^{-x}\sum_{k=0}^{n-1}\frac{x^k}{k!}\right)$$

$$\Rightarrow \Gamma(n,1) = (n-1)!\left(1 - \mathrm{e}^{-1}\sum_{k=0}^{n-1}\frac{1}{k!}\right)$$

For sufficiently large values of n, $\left(\mathrm{e}^{-1}\sum_{k=0}^{n-1}\frac{1}{k!}\right)$ converges to 1. Then, $\lim_{n\to\infty}\Gamma(n,1)$ is a not defined limit of the form $0\times\infty$.

But, for complexity evaluation purposes, we can safely assume that $\left(1 - \mathrm{e}^{-1}\sum_{k=0}^{n-1}\frac{1}{k!}\right) < 1$, and, consequently $\Gamma(n,1) = (n-1)!\left(1 - \mathrm{e}^{-1}\sum_{k=0}^{n-1}\frac{1}{k!}\right) < (n-1)!$.

Thus, the SDCS number SDC problems solved complexity would be $\mathcal{O}(n^2(n-1)!) = \mathcal{O}(n\times n!)$ if no *budget* constraints were set.

It is important to notice that, SDCS algorithm iteratively adds constraints to the SDC problem (line 10 on Algorithm 1 and line 10 on Algorithm 2) in an attempt to force the solution to have a feasible MRT. In this sense, we say that the algorithm makes an implicit division of the problem into its scheduling and allocation parts once it changes the scheduling problem part in order to search through different allocations until it finds a valid MRT. This number of problems solved is effectively the main source of its inefficiency.

Formulation 4.1 presents the SDC problem formulation. The constraints (line 3) capture that instruction I_j should start after instruction I_i, where D_i is the delay of instruction I_i, $b_{i,j}$ measures the intra-loop dependencies and is represented as a back-edge in the Data-Flow Graph (DFG), such as $b_{i,j} > 0$ for back-edges between I_i and I_j, and $b_{i,j} = 0$ for forward edges.

Formulation 4.1 – SDC problem.

(1)	**minimize:**	$\sum_i t_i$
(2)	**subject to:**	
(3)	$t_i - t_j$	$\leq -D_i + b_{i,j}II$

4.2. ILP Modulo Scheduler (ILPS)

Oppermann *et al.* 2016 presents an alternative formulation for the module scheduling problem based on an ILP formulation, which is motivated by the fact that the SDC formulation cannot represent resource constraints.

As such, ILPS formulates both schedule and allocation constraints, and with all the information in a single problem, it guarantees to find the optimal solution and eliminates the necessity of an iterative search. On the one hand, ILPS solves only one optimization problem. On the other hand, the optimization problem is a general ILP, which is solvable in exponential time.

The typical formulation of a resource constraint scheduling problem includes time decisions variables for each resource instance and for each time slot, resulting in large `problem sizes`. To ease this burden, the ILPS uses overlap variables as suggested in Venugopalan and Sinnen 2015.

The decision variables included are m_i and r_i, which give the congruence class of instruction i in the schedule and its resource index, where $i = 0,..,n-1$ are the instructions that are processed on resource-constrained units.

The overlap variables included are the binary μ_{ij}, which is 1, if the congruence class of i is smaller than j's, and ε_{ij}, which is 1 if the resource designed to instruction i has an index smaller than j's. With these variables, it is possible to create constraints that guaranty MRT validity.

The ILP formulation is presented in Problem 4.2, with domains described in Chart 4.

Chart 4 – Problem signature from the ILP module scheduling described in Oppermann *et al.* 2016.

Input	
II	Iteration Interval
$O = \{0,\ldots,n-1\}$	Operations
$D_i,\ i \in O$	Latency of operation i
$E = \{(i \rightarrow j;b)\} \subseteq O \times O \times 0,1$	dependency edges on data-path. $b = 1 \Leftrightarrow backedge$
$R = mem, dsp$	Resource types
$A = a_k \mid k \in R$	instance of resource k
$L_k \subseteq O$	resource limited operations of type k
$L = U_{k \in R} L_k$	union of all resource limited operations
Output	
$t_i,\ i \in O$	start time of operation i
$r_i, i \in L$	resource instance used by operation i
Decision Variables	
$t_i,\ i \in O$	start time of operation i
$r_i,\ i \in L$	resource instance used by operation i
$m_i,\ i \in L$	congruence class of operation i
$y_i,\ i \in L$	helper variable of operation i
$\varepsilon_{ij},\ \forall k \in R : \forall i,j \in L_k, i \neq j$	$= 1$ if $r_i < r_j$, $= 0$ otherwise
$\mu_{ij},\ \forall k \in R : \forall i,j \in L_k, i \neq j$	$= 1$ if $m_i < m_j$, $= 0$ otherwise

Source: Oppermann *et al.* 2016.

From overlap variables number grows with complexity $\mathcal{O}(n^2)$, where n is the `loop size`. As such, the ILPS overall complexity is $\mathcal{O}\left(e^{n^2}\right)$. However, the fact that the ILP problem needs

Formulation 4.2 – ILP formulation for module scheduling presented in Venugopalan and Sinnen 2015

(1) minimize	$\sum_{i \in O} t_i$		
(2) subject to:	$t_i + D_i$	$\leq t_j + bII$	$\forall i \rightarrow j; b \in E$
(3)	$\varepsilon_{ij} + \varepsilon_{ji}$	≤ 1	$\forall k \in R : \forall i, j \in L_k, i \neq j$
(4)	$r_j - r_i - 1 - (\varepsilon_{ij} - 1)a_k$	≥ 0	$\forall k \in R : \forall i, j \in L_k, i \neq j$
(5)	$r_j - r_i - \varepsilon_{ij} a_k$	≤ 0	$\forall k \in R : \forall i, j \in L_k, i \neq j$
(6)	$\mu_{ij} + \mu_{ji}$	≤ 1	$\forall k \in R : \forall i, j \in L_k, i \neq j$
(7)	$m_j - m_i - 1 - (\mu_{ij} - 1)II$	≥ 0	$\forall k \in R : \forall i, j \in L_k, i \neq j$
(8)	$m_j - m_i - \mu_{ij} II$	≤ 0	$\forall k \in R : \forall i, j \in L_k, i \neq j$
(9)	$\varepsilon_{ij} + \varepsilon_{ji} + \mu_{ij} + \mu_{ji}$	≥ 1	$\forall k \in R : \forall i, j \in L_k, i \neq j$
(10)	t_i	$= y_i II + m_i$	$\forall i \in L$
(11)	r_i	$\leq a_k - 1$	$\forall i \in L_k$
(12)	m_i	$\leq II - 1$	$\forall i \in L_k$
(13)	$t_i + D_i$	$\leq SL_{max}$	$\forall i \in O$
(14)	t_i	$\in N$	$\forall i \in O$
(15)	r_i	$\in N$	$\forall i \in L$
(16)	y_i	$\in N$	$\forall i \in L$
(17)	m_i	$\in N$	$\forall i \in L$
(18)	ε_{ij}	$\in \{0,1\}$	$\forall k \in R : \forall i, j \in L_k, i \neq j$
(19)	μ_{ij}	$\in \{0,1\}$	$\forall k \in R : \forall i, j \in L_k, i \neq j$

to be solved only once per candidate II counterbalances its exponential solving time complexity when compared with the SDCS.

First of all, a comparison between the ILPS and SDCS is presented in Oppermann *et al.* 2016. However, the following SDCS changes were made by the authors in Oppermann *et al.* 2016:

- *RecMII* is calculated as proposed in Venugopalan and Sinnen 2015,
- The *budget* to increment II is changed for a time budget,
- The topological order used by SDC is changed for a depth branch first order,
- The CPLEX solver is used instead of the LPSolver.

Note that, with these changes, if an infeasible II is given to SDCS, it will spend all the budget time trying to solve the problem. With these changes, the SDCS and ILPS show similar efficiency to find a schedule to CHStone and MachSuite benchmarks (Venugopalan and Sinnen

2015). Furthermore, ILPS does not consider other constraints that model HLS optimizations, as chaining operations, which are implemented in LUP.

We implemented the ILPS on LegUP by adding the overlap constraints and variables on a typical SDC formulation, what gives support for all HLS optimization implemented by LUP at formulation level. The ILPS *II* candidates are the same as the SDCS ones, and we keep SDCS default topological order and budget.

4.3. List-Based Modulo Schedulers

Opposing the SDCS and ILPS, the list-based modulo schedulers family do not solve any optimization problem. These schedulers have two main steps. In the first step, all instructions are sorted according to a heuristic. In the second part, the instructions are allocated in the sorted order, allowing to calculate their scheduling time. Once an instruction is allocated, its scheduled time is calculated. As such, the following instruction needs to be allocated and scheduled according to the previous instructions. This "one-at-a-time" scheduling and allocation is what characterizes the list-based scheduler family.

Contrasting with list-based schedulers, an optimization problem calculates and allocates all instructions at the same time, considering all combinations. As consequence, list-based schedulers are usually fast, but often fail to find a schedule due to the restricted number of scheduling and allocations explored, when compared to schedulers based on optimization problems (Codina, Llosa and González 2002).

We choose the Swing Modulo Scheduler (SMS) (Llosa *et al.* 2001) to be implemented and tested against the modulo schedulers presented in this chapter since it presents the best results when compared to other list-based schedulers by Codina, Llosa and González 2002.

SMS has two main parts: The first part focuses on sorting the instructions such that the distance between dependencies is minimized, aiming to reduce the register pressure. The second part allocates the MRT instructions to calculate a scheduled time for them.

Since the list-based schedulers are not the base for our proposed modulo schedulers, and for conciseness purposes, we refer to Llosa *et al.* 2001 for SMS implementation details.

4.4. Explicitly Scheduling and Allocation Separation

Differently from the previous approaches, we propose a new approach that separates the ILP formulation scheduling and allocation parts, which allows to use an SDC formulation to solve the scheduling part and another method to explicitly traverse through valid MRT (i.e. the allocation space). This approach differs from (Canis, Brown and Anderson 2014), which modifies the SDC problem (scheduling part) by adding constraints trying to force its solution to

have a valid MRT indirectly.

This section presents how we transform Formulation 4.2 to allow the scheduling and allocation to be separately explored. Furthermore, this section presents a relaxation for the proposed formulation, that is useful due to its feasibility properties.

On Formulation 4.2, the overlap variables (lines 5 to 11) guarantee that two instructions are not allocated to the same MRT slot, and thus ensuring a valid MRT. In the proposed approach, the allocation part will ensure a valid MRT construction, and as such the overlap variables are eliminated from the formulation. The rest of the problem formulation has: dependency constraints (line 4), other HLS constraints (line 15), and the linker constrains between t_i and m_i (line 12). As such, the remaining problem can be rewritten as:

$$\begin{aligned} &\textbf{minimize:} && \textstyle\sum_{i \in O} t_i \\ &\textbf{subject to:} && \\ &t_i - t_j && \leq b_{ij} II - D_i \\ &t_i && = m_i + y_i II \end{aligned}$$

Substituting t_i by $m_i + y_i \times II$ in all constraints and objective function, t_i is eliminated from the problem, and y_i is promoted to a decision variable, resulting in Formulation 4.3. Note that t_i values can be calculated with the m_i values and the solution y_i.

Formulation 4.3 – SDC problem for scheduling and allocation separation

$$\begin{aligned} (1)\quad &\textbf{minimize:} && \textstyle\sum_{i \in O} y_i * II + m_i \\ (2)\quad &\textbf{subject to:} && \\ (3)\quad &y_i - y_j && \leq \left\lfloor \frac{b_{ij} II - D_I - (m_i - m_j)}{II} \right\rfloor \end{aligned}$$

Other HLS constraints in the SDC can also be added in Formulation 4.3 in a similar way.

Assuming a valid MRT has been produced by the allocation stage, the m_i and r_i are known, and by solving the SDC Formulation 4.3, the y_i values are calculated in polynomial time, leading to the t_i values. Furthermore, the solution latency is used as an MRT measure of quality, since solving Formulation 4.3 for different MRT results in schedules that differ only in latency, which is the only non-constant factor in $total_{cycles} = l + II \times (tc - 1)$, for a given II.

Summarizing, formulation 4.3 allows us to divide the modulo scheduling problem into the allocation and scheduling parts explicitly. The proposed method traverses the MRT space and produces possible allocations, which then drive the scheduling part through the SDC Formulation 4.3. The obtained solutions latency guides the MRT traversing space.

The MRT space traversing can be performed by various methods such as a heuristic, meta-heuristic, evolutionary algorithm, machine learning techniques, while the scheduling part can be optimally solved using Formulation 4.3, in polynomial time.

Nevertheless, Formulation 4.3 is not feasible for any MRT construction.

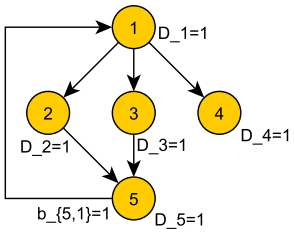
Definition: We define "feasible MRT" and "infeasible MRT" as MRT to which Formulation 4.3 is feasible and infeasible, respectively.

Formulation 4.3 feasibility is demonstrated by the Data-Flow Graph (DFG) example in Figure 5a. Let's assume that we only have 2 resource instances to execute instructions $I_i = \{I_1, I_2, I_3, I_4, I_5\}$, thus $minII = 3$ and the MRT has 3 rows and 2 columns. Figure 5b and 5e show the Non-Resource Constrained As Soon As Possible (NRCASAP) schedule and the optimal MRT, respectively.

Figure 5c shows a feasible MRT, with scheduling times $\mathbf{t} = \{0,1,1,3,2\}$, and the back-edge constraint given as $t_5 - t_1 \leq b_{5,1} \times II - D_5 \Rightarrow t_1 \geq 0$, which is satisfied since $t_1 = 0$. Figure 5d shows an infeasible MRT, with scheduling times $\mathbf{t} = \{0,3,1,1,5\}$, and the back-edge constraint given as $t_5 - t_1 \leq b_{5,1} \times II - D_3 \Rightarrow t_1 \geq 3$, which is not satisfied since $t_1 = 0$.

Figure 5 – Example of different MRT for a DFG.

(a) DFG example.



(b) NRCASAP for ex. 5a.

m	–	–	–
0	i_1	–	–
1	i_2	i_3	i_4
2	i_5	–	–

(c) A feasible MRT for ex. 5a.

m	a_0^0	a_0^1
0	i_1	i_4
1	i_2	i_3
2	i_5	–

(d) An infeasible MRT for ex. 5a.

m	a_0^0	a_0^1
0	i_1	i_2
1	i_4	i_3
2	i_5	–

(e) Optimal MRT for ex. 5a.

m	a_0^0	a_0^1
0	i_1	–
1	i_2	i_3
2	i_5	i_4

As such, the allocation space exploration needs to also consider infeasible MRT, for which the obtained solution quality (i.e. latency) is not available. As the MRT is infeasible, the operations cannot be scheduled, and as such, there is no quality metric (i.e. schedule latency) associated with the MRT.

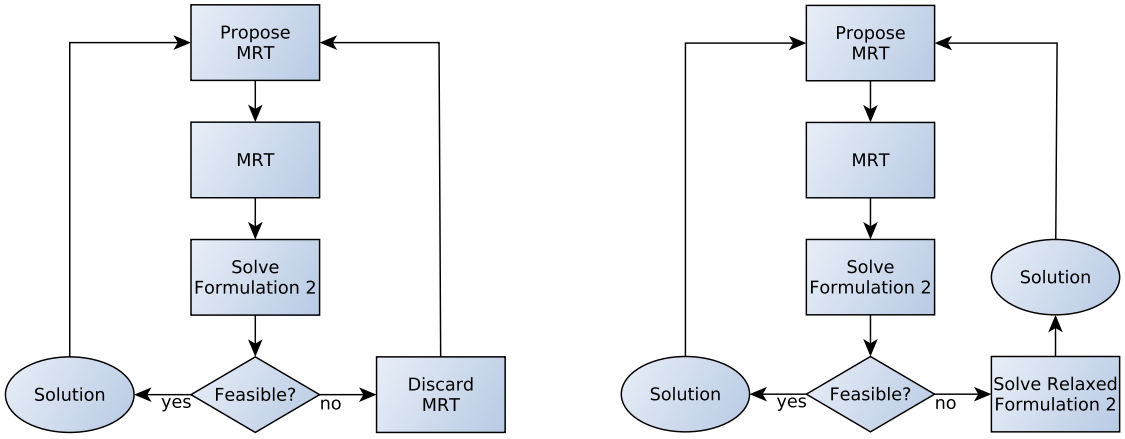
In this fashion, a possible solution is to discard the infeasible MRT and keep generating MRT and analysing only the feasible ones, as summarized in the flow presented in Figure 6a. However, experimental work indicated that finding a feasible MRT through the above algorithm is rare. More specifically, when using benchmark **cp** (described in Section 4.7.1), 145872.23 random MRT on average (50 repetitions) need to be created before finding a feasible one.

Instead of discarding infeasible MRT, we propose a new approach that allows the association of an infeasible MRT with a latency value, and as such would allow its incorporation

in the MRT exploration stage. The proposed approach is based on relaxing Formulation 4.3. The relaxed formulation solution results in a schedule for the loop with an associated latency, even though this may not be an actual solution for the non-relaxed problem. However, it can be used to compare infeasible MRT with feasible ones, as illustrated in the flow in Figure 6b.

Figure 6 – Different flows to handle infeasible MRT.

(a) Straight forward. (b) Using relaxation.



Source: Elaborated by the author.

Next, we present a relaxation for Formulation 4.3 that is guaranteed to be feasible even for infeasible MRT. Formulation 4.4 relaxes Formulation 4.3 by renaming $y_i II = x_i$.

Formulation 4.4 – SDC problem relaxation for scheduling and allocation separation

$$
\begin{aligned}
&(1) && \textbf{minimize:} \\
&(2) && \textstyle\sum_{i \in O} x_i + m_i \\
&(3) && \textbf{subject to:} \\
&(4) && x_i - x_j \quad \leq -D_i + b_{ij} II - (m_i - m_j)
\end{aligned}
$$

In Formulation 4.4, there is no guarantee that x_i values are multiples of II, and as such, y_i is not guaranteed to be integer. Thus, there is no guarantee that $t_i \% II = (x_i + m_i) \% II = m_i$. In other words, the MRT that corresponds to the solution (calculated as $t_i \% II$) is not guaranteed to be the same valid MRT used to construct the problem in Formulation 4.4, hence not guaranteed to be valid. This has no consequences for the flow proposed in Figure 6b, since the schedule values (t_i) are only used for comparison.

In the following paragraphs, it is proved that if a feasible MRT exists for Formulation 4.3, then Formulation 4.4 is always feasible (regardless of the MRT). As such, by employing Formulation 4.4, the proposed methodology ensures that a schedule can be derived that can be used to guide the DSE process.

Notation: let δx_{ab} represent the $x_a - x_b$.

For the following proof, the following bounds need to be introduced:

1. $-(II-1) \leq \delta m_{ij} \leq (II-1)$

2. $0 \leq D_i \leq (II-1)$

3. $0 \leq (II-1) - D_i \leq (II-1)$

Bound **1** is a consequence that $0 \leq m_i \leq (II-1)$ for any instruction I_i by definition. Bound **2** implies that instruction I_i delay is smaller than II, which is valid for functional units, which are not pipelined themselves. Thus, it is imperative that every instruction has to finish its execution before being executed again. Bound **3** is a consequence of bound **2**.

The first consideration to make is that the dependencies created by the forward edges $(I_i \rightarrow I_j)$ can always be satisfied regardless of the MRT. This is because the Right-Hand Side (RHS) in Formulation 4.3 can be only 0 or -1, since $b_{ij} = 0$, and bounds **1** and **2** make $-(II-1) - (II-1) \leq -D_i - \delta m_{ij} \leq 0 + (II-1)$. The forward edge constraints are also always valid for Formulation 4.4 since it is a relaxation of Formulation 4.3. Thus, we will only consider the back-edges constraints from now on.

Using bound **1**, we can expand the RHS of Formulation 4.3 to enclose all possible values of the MRT, including the ones that would make it infeasible, resulting in Inequality inq.1.

$$\delta y_{ij} \leq \left\lfloor \frac{b_{ij}II - D_i - \delta m_{ij}}{II} \right\rfloor \leq \left\lfloor \frac{b_{ij}II - D_i + (II-1)}{II} \right\rfloor \qquad \text{(inq.1)}$$

In the same fashion, for Formulation 4.4 to be feasible for any MRT, is equivalent to rewrite the RHS on Formulation 4.4 as Inequality inq.2. That is, δx_{ij} has to be smaller than the lower bound of $b_{ij}II - D_i - \delta m_{ij}$.

$$\delta x_{ij} \leq b_{ij}II - D_i - (II-1) \leq b_{ij}II - D_i - \delta m_{ij} \qquad \text{(inq.2)}$$

Thus, the condition for the feasible space of Formulation 4.4 to encompass all values of MRT is if Inequality inq.1 is encompassed by Inequality inq.2, resulting in Inequality inq.3.

$$\left\lfloor \frac{b_{ij}II - D_i + (II-1)}{II} \right\rfloor \leq b_{ij}II - D_i - (II-1) \qquad \text{(inq.3)}$$

If $II = 1$, Inequality inq.3 always holds. Thus, we only need to analyse the case when $II > 1$.

Using bound **3**, we can see that the Left-Hand-Side of Inequality inq.3 is equal to b_{ij}, implying that the condition in Inequality inq.3 can be rewritten as Inequality inq.4.

$$\begin{aligned} & b_{ij} && \leq b_{ij}II - D_i - (II-1) \\ \Rightarrow\ & b_{ij} && \leq b_{ij} + b_{ij}(II-1) - D_i - (II-1) \\ \Rightarrow\ & b_{ij} && \geq \left\lfloor \frac{D_i + (II-1)}{(II-1)} \right\rfloor \end{aligned} \tag{inq.4}$$

Finally, using bound **3**, and that $b_{ij} \geq 1$ by definition, we conclude that Inequality inq.4 always holds, meaning also that Inequality inq.3 always holds.

As such, if Formulation 4.3 is feasible for an MRT, then Formulation 4.4 is always feasible regardless the MRT.Next, we present a relaxation for Formulation 4.3 that is guaranteed to be feasible even for infeasible MRT. Formulation 4.4 relaxes Formulation 4.3 by renaming $y_i II = x_i$.

4.5. SDC-Based Genetic Algorithm (GAS)

As noted in Section 4.4, any method of choice can explore the allocation space as long valid and feasible MRT are produced by the method, then Formulation 4.3 can be used to solve the scheduling problem. We chose a GA since it is known to be applicable for problems without information about the solution structure (Mitchell 1996).

The first step is the GA chromosomes definition. Formulations 4.3 and 4.4 allow us to select the MRT as chromosomes (individuals), what allows us to reduce the number of problem variables (described in lines 5 to 11 on Formulation 4.2) by capturing these constraints with the GA chromosome encoding. As will be explained later, the chromosome evolution guarantees a valid MRT, and as such, lines 5 to 11 are removed (with the overlap variables).

The GA purpose is to evolve MRT (individuals) according to the final schedule latency (fitness). That is, since II is given, and tc is constant, the GA evolves individuals to reduce the solution latency. MRT are implemented as lists of (m_i, r_i, I_i) triplet, where (m_i, r_i) correspond instruction I_i MRT slot, which is a common way to represent sparse matrices such as the MRT.

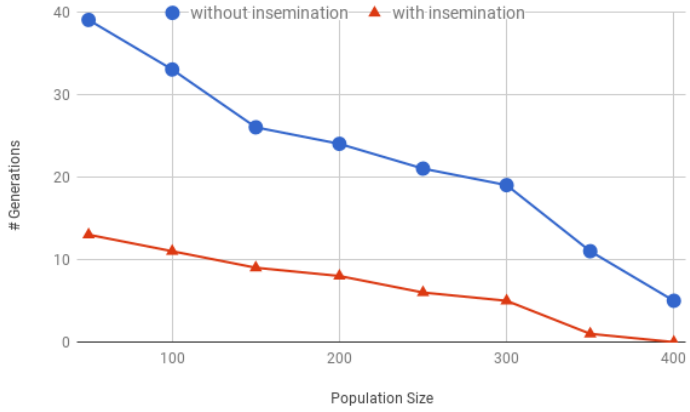
The proposed GA algorithm is described in Algorithm 3. The parameters impacting GAS are the population size *nPop*, the insemination rate i_r, the number of new individuals created at each generation *offspringSize*, the minimum number of generation *minGen*, and the mutation probability *mutationProb*.

The first step is the creation of a random population (lines 1 to 4). The initial population is inseminated with individuals created by randomly legalizing the MRT associates with the

NRCASAP schedule (lines 5 and 11) using the legalization process (described in details in Algorithm 5). Since the NRCASAP schedule is the unreachable ideal, we expect that its MRT values contain "good genes" and the optimal to be close to a slight modification of its MRT.

The population insemination introduces some knowledge about the solution structure into the GA, which is shown to improve GA results as shown in (Rosa *et al.* 2016). Figure 7 shows a typical GAS execution number of generations needed to find a feasible MRT in function of the population size, with and without the insemination. For Figure 7, we used the benchmark **complex** (described in details in Section 4.7.1).

Figure 7 – Average number of generations needed to find a feasible MRT with and without the insemination process.



Source: Research data.

Consider the example in Figure 5a, with $a_0 = 2$ resource instances and $MII = 3$. Figure 5b presents the NRCASAP MRT, which has three instruction with congruence class $m = 1$. The legalization process can generate the MRT presented in Figures 5c, 5d, or 5e, which will be used to inseminate the initial population.

The population is then evolved for *minGen* generations (lines 13 to 23). The evolution process consists of creating *offspringSize* new individuals, called cubs, (lines 14 to 22), from two different individuals randomly selected from the population (line 15) through a cross-over process (described in details in Algorithm 6). The mutation occurs by attributing a new random empty MRT slot to the instructions in the newly created MRT (lines 18 to 20). Then, the cubs are added to the population (lines 21 and 22), what results in a $nPop + offspringSize$ individuals population. Selection is made by ordering and selecting the *offspringSize* best population individuals (lines 23 and 24).

Finally, the algorithm does not stop until it reaches the minimum number of generations *minGen*, or until it finds a feasible MRT (line 27). Furthermore, we define $budget$ as a maximum number of generations in case a feasible MRT is not found (lines 25 and 26), which causes GAS to return a failure for the given II meaning that the candidate II should be increased.

Algorithm 3: Genetic Algorithm to calculate a schedule for a given II.

input : II and GA parameters
output : A schedule for II

1 /* Initialize population step */
2 **for** $i \leftarrow 0$ **to** $nPop - 1$ **do**
3 create a valid MRT_i by assigning each instruction to a random empty slot;
4 $fitness_i \leftarrow$ evaluateIndividual(MRT_i);
5 add $(MRT_i, fitness_i)$ to $population$;
6 /* Insemination Process */
7 Calculate the non-resource constrained ASAP schedule;
8 Calculate the non-valid MRT_{ASAP};
9 **for** $i \leftarrow 0$ **to** $i_r * nPop$ **do**
10 $MRT_i \leftarrow MRT_{ASAP}$;
11 validateIndividual(MRT_i);
12 $fitness_i \leftarrow$ evaluateIndividual(MRT_i);
13 add $(MRT_i, fitness_i)$ to $population$;
14 /* Population evolution */
15 $gen \leftarrow 0$;
16 **do**
17 /* Offspring creation */
18 **for** $i \leftarrow 0$ **to** $\frac{offspringSize-1}{2}$ **do**
19 $(p1, p2) \leftarrow$ different random individuals in the population;
20 $(cub1, cub2) \leftarrow$ crossOver(p1, p2);
21 **for** $cub \in (cub1, cub2)$ **do**
22 /* mutation */
23 **for** *instruction* $I \in MRT_{cub}$ **do**
24 **if** *random probability* $>$ *mutationProb* **then**
25 reallocate I in a random available slot in the MRT_{cub};
26 $fitness_i \leftarrow$ evaluateIndividual(MRT_{cub});
27 add $(MRT_{cub}, fitness_{cub})$ to $population$;
28 /* selection */
29 sort $population$ according to increasing $fitness$ order;
30 $population \leftarrow nPop$ first individuals of $population$;
31 **if** $gen > budget \times minGen$ **then**
32 fail to schedule with the given II;
33 **while** $!(gen \geq minGen \;\&\&\; foundFeasible)$;

Algorithm 4 describes the evaluation function, which considers two cases. The first case is when the MRT is feasible, and thus we have a solution to the problem. The second case is when the MRT is infeasible, and thus Formulation 4.4 is used to calculate the fitness with a penalty given by how many conflicts there were in the solution of Formulation 4.4.

First, the algorithm tries to solve Formulation 4.3 (line 1), and the resulting schedule latency is returned if it is feasible (lines 2 to 5).

If Formulation 4.3 is infeasible for the given MRT, Formulation 4.4 (line 7) is solved, which is guaranteed to be feasible. The MRT corresponding to the relaxed problem solution $MRT_{relaxed}$ is calculated using $m_i^{relaxed} = t_i \% II$ (line 9), where $t_i = (x_i + m_i)$. The latency is calculated using t_i values (line 9).

Since $MRT_{relaxed}$ is not guaranteed to be valid, we modify it using the validation function (line 11), which return how many resource conflicts the non-valid MRT had and randomly fixes the MRT (as described in Algorithm 5).

Finally, the function returns the latency l for the relaxed schedule plus a penalty according to the number of resource conflicts the MRT had times II (line 12). This penalty captures that the relaxed schedule can be a solution for the non-relaxed one, which happens when no resource conflicts occur, and also punishes relaxed schedules with conflicts, making them less likeable to survive through generations.

Algorithm 4: Function for individual evaluation for the GA presented in Algorithm 3.

```
   input  : MRT
   output : Fitness for the MRT
1  success ← solve Formulation 4.3 for MRT;
2  if success then
3      Calculate the solution latency l;
4      return l + II * tc;
5      foundFeasible ← true
6  else
7      solve Formulation 4.4 for MRT;
8      Calculate the non valid MRT_relaxed;
9      Calculate the solution latency l;
10     MRT ← MRT_relaxed;
11     n_outs ← validateIndividual(MRT);
12     return l + II * n_outs;
```

As example, consider Figure 5c, which is feasible and Formulation 4.3 returns the schedule $\mathbf{t}_{5c} = \{0,1,1,2,3\}$, with latency $l_{5c} = 4$, resulting in $fitness_{5c} = 4$.

Now, consider Figure 5d, which is infeasible, and solving Formulation 4.4 results in $\mathbf{x}_{5d} = \{0,1,0,0,0\}$, thus $\mathbf{t}_{5d} = \{0,1,1,1,2\}$. Note that $MRT_{relaxed}$ has instructions $\{I_2, I_3, I_4\}$ in the same congruence class, thus it is not valid, and the validation function will return the latency $l_{5d} = 3$ and the number of conflicts $n\_outs = 1$, resulting in $fitness_{5d} = 3 + 1 * 3 = 6$. Furthermore, $MRT_{relaxed}$ needs to be validated, before being added back to the population.

Algorithm 5 describes the function used to modify non-valid MRT into valid ones, randomly handling conflicts, and also counting how many conflicts there were in the MRT. This function is necessary for the insemination and individual evaluation processes since invalid MRT can be generated in both procedures.

Each resource constrained instruction I is attempted to be allocated in its current MRT slot (lines 2 and 3), and if there is already another instruction allocated to that slot, I is attempted to be scheduled in another resource instance, with the same congruence class (lines 4 to 8). This solves conflicts where two instructions are assigned for the same resource instance, but there are still other available instances in the same congruence class.

If all resources are busy, a randomly allocated instruction I_r in the congruence class is selected (line 10), and the number of conflicts is incremented (line 16). Finally, both instruction dispute a coin toss to define which one will be allocated to a random MRT empty slot (lines 11 to 15). This random selection is made as there is no information on which allocation is the best to be performed.

Algorithm 5: Function for counting and fixing MRT conflicts for the GA presented in Algorithm 3.

```
   input  : MRT
   output : New valid MRT and n_outs
 1 n_outs ← 0;
 2 for each instruction I ∈ MRT do
 3     if MRT slot is not available then
 4         m → I congruence class;
 5         r → I resource instance;
 6         r_try ← any available resource instance in the congruence class m;
 7         if ∃ r_try then
 8             allocate I in slot (m, r_try);
 9         else
10             I_r ← a random allocated instruction of congruence class m;
11             /* 50 % chance for each case                                  */
12             if random(0,1) ≥ 0.5 then
13                 reallocate I to an available random MRT slot (new random (m, r)
                    values);
14             else
15                 allocate I in I_r slot in the MRT;
16                 reallocate I_r to an available random MRT slot;
17             n_outs = n_outs + 1;
18 return n_outs;
```

To illustrate the validation process, consider Figure 5b. First I_1, I_2, and I_3 will successfully be allocated in their congruence class. Then I_4 will fail to be allocated with $m_4 = 1$. A random allocated instruction with $m = 1$ is chosen, which can be I_2 or I_3. Let's assume that I_2 is chosen, then a random selection between I_2 and I_4 will be performed. If I_2 is selected, it remains as it is, and I_4 is randomly allocated into an empty slot, which can result to MRT depicted in Figure 5c or Figure 5e. If I_4 is selected, I_2 is randomly allocated which leads to the MRT depicted in Figure 5d.

Algorithm 6 describes the single point cross-over function used. The cross-over point is a randomly chosen number of instructions (line 1), which are copied to cub_1 and cub_2 respectively from parents p_1 and p_2 (lines 2 to 4). The rest of the instructions are copied from the opposite parent (lines 5 to 7). This simple process might cause MRT conflicts since an instruction of the first part of p_1 can occupy the same slot as an instruction in the second part of p_2. Possible conflicts are solved using the validation process described in Algorithm 5.

Algorithm 6: Cross-over function for the GA presented in Algorithm 3.

```
   input  : Parents p1 and p2
   output : Offspring cub1 and cub2
 1 crossPoint ← rand(1, nInsts − 1);
 2 /* copy first part of chromosome from corresponding parent      */
 3 for I_i ← 0 to crossPoint − 1 do
 4 |   cub1 :: (m_i, r_i, I_i) ← p1 :: (m_i, r_i, I_i);
 5 |   cub2 :: (m_i, r_i, I_i) ← p2 :: (m_i, r_i, I_i);
 6 /* copy second part of chromosome from the other parent         */
 7 for I_i ← crossPoint to nInsts − 1 do
 8 |   cub1 :: (m_i, r_i, I_i) ← p2 :: (m_i, r_i, I_i);
 9 |   cub2 :: (m_i, r_i, I_i) ← p1 :: (m_i, r_i, I_i);
10 /* fix possible conflicts in the cubs                           */
11 validateIndividual(MRT_cub1);
12 validateIndividual(MRT_cub2);
```

To illustrate the cross-over, consider Figures 5c and 5d, which are represented by the lists $p_1 = MRT_{5c} = \{(0,0,I_1),(1,0,I_2),(1,1,I_3),(0,1,I_4),(2,0,I_5)\}$ and $p_2 = MRT_{5d} = \{(0,0,I_1),(2,0,I_2),(1,1,I_3),(1,0,I_4),(0,1,I_5)\}$, respectively. The cross-over point is a random number between 1 and 4. Supposing the cross-over point is 2, the resulting cubs would be $cub_1 = \{(0,0,I_1),(1,0,I_2),(1,1,I_3),(1,0,I_4),(0,1,I_5)\}$ and $cub_2 = \{(0,0,I_1),(2,0,I_2),(1,1,I_3),(0,1,I_4),(2,0,I_4)\}$.

Finally, the total number of individuals evaluated by GAS is $totalIndividuals = nPop \times (1 + inseminationRate) + maxGen \times offspringSize$. Under the proposed formulation, for each individual a maximum of two SDC problems are solved. As such, in order for GAS to be faster than SDCS, the $totalIndividuals$ need to be less than the number of problems solved by SDCS.

Let $nPop = offspringSize = \alpha \times n$ and $maxGen = \beta$, where α and β are parameters defined by the user, and n is the loop size. Thus, $totalIndividuals = n \times (\alpha(1 + inseminationRate) + \beta)$. Since, in worst-case, 2 SDC problems are solved per individual (Formulations 4.3 and 4.4), the number SDC problems solved by GAS scales linearly with the `loop size`.

The conditions for GAS to be faster than the SDCS worst-case are:

$$\begin{aligned} & 2\times(\alpha n(1+i_r)+\beta(\alpha n)) && < budgetRatio\times n^2 \\ \Rightarrow \quad & 2\alpha(\beta+1+i_r)1 && < budgetRatio\times n \\ \Rightarrow \quad & \tfrac{2\alpha(\beta+1+i_r)}{budgetRatio} && < n \end{aligned} \tag{4.2}$$

Inequality 4.2 is a sufficient condition for GAS to be faster than SDCS worst-case. We will come back to the impact of the above condition in the performance evaluation section when the parameters α and β are tuned.

4.6. Non-Iterative Modulo Scheduler (NIS)

This section presents the studies made to answer two questions that naturally arise on the genetic algorithm presented in section 4.5: "what makes an individual to have a good fitness?" and "what makes an MRT infeasible?". Such questions allowed us to implement a data-flow-based method to create a valid and feasible MRT, with latency minimization. The creation of such MRT allows us to avoid GAS and SDCS iterative searches.

Thus, unifying the MRT construction method presented in this section with Formulation 4.3 creates a non-iterative approach that avoids solving several SDC problems to find a valid MRT (as SDCS), avoids solving the scheduling and allocation parts (as ILPS) simultaneously, avoids evolving several MRT to find a feasible one (as in GAS), and evaluates several schedules for the given MRT (opposing to SMS).

The proposed NIS focus on two objectives during MRT construction, the MRT feasibility and the schedule latency reduction. Next, we analyse both objectives and present the proposed heuristic to construct feasible and valid MRT.

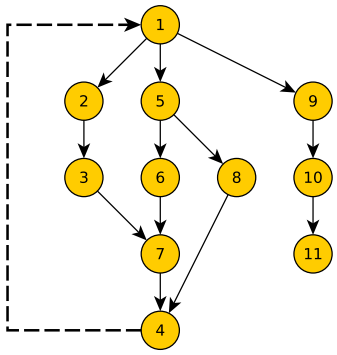
Objective 1: MRT Feasibility

Consider all cycles ψ in a DFG, which is a list of edges starting with the back-edge destination and ending with the back-edge source. Figure 8 presents a simple example with 3 cycles: $\psi_0 = \{I_1, I_5, I_6, I_7, I_4\}$, $\psi_1 = \{I_1, I_2, I_3, I_7, I_4\}$, and $\psi_2 = \{I_1, I_5, I_8, I_4\}$.

To analyse Formulation 4.1 feasibility, we need to consider all constraints related to instructions belonging to cycles, once the other instructions do not render Formulation 4.1 infeasible (Rosa, Bouganis and Bonato 2018).

Writing all constraints for all instruction in a cycle $\psi = \{i_\alpha, i_{\alpha+1}, \ldots, i_{\omega-1}, i_\omega\}$ results

Figure 8 – A simple DFG example.



Source: Elaborated by the author.

in Inequalities 4.3.

$$\begin{cases} t_\alpha - t_{\alpha+1} & \leq -D_\alpha \\ \dots & \\ t_\omega - t_\alpha & \leq -D_\omega + b_{\omega,\alpha} II \end{cases} \tag{4.3}$$

Inequality 4.4 results from summing up all Inequalities 4.3, and is defined as the "slack" of cycle ψ.

$$s_\psi = l_b^\psi - l_f^\psi \geq 0 \tag{4.4}$$

Where $l_f^\psi = \sum_{i \in \psi} D_i$ defines the cycle forward path length in a non-resource constrained schedule, and $l_b^\psi = b_{\omega,\alpha} II$ is defined as the cycle back-edge length.

Considering the resources constraints, an instruction I_i is associated with its congruence class m_i in the MRT, which can create a delay to the instruction execution defined as $0 \leq g_i \leq II - 1$.

Then, Equation 4.4 is redefined in Inequality 4.5 as the effective slack, which represents the final schedule slack.

$$es_\psi = l_b^\psi - l_f^\psi - \sum_{i \in \psi} g_i = s_\psi - \sum_{i \in \psi} g_i \geq 0 \tag{4.5}$$

The condition $es_\psi \geq 0 \mid \forall \psi$ is necessary to satisfy all Inequalities 4.3. Thus, it is also a condition for the MRT to be feasible.

Even though g_i can be only calculated with the final schedule in place, Inequality 4.5 shows that a larger slack s_ψ means that $\sum_{i \in \psi} g_i$ can be larger without violating $es_\psi \geq 0$. In other words, a larger slack s_ψ means that the cycle ψ instructions can be more "re-arranged" in the

MRT without violating $es_\psi \geq 0$. As such, the proposed heuristic set instructions in DFG cycles with smaller s_ψ to be arranged in the MRT with a higher priority.

Objective 2: Latency Reduction

As noted in Section 4.6, the position of an instruction in the MRT can add delays g_i in the final schedule forward length es_f^ψ. This is caused when an instruction is scheduled with a congruence class different from $m_0 = t_0 \% II$, where t_0 is the time when all its dependencies are ready. Furthermore, whenever an instruction dependency is delayed due to MRT conflicts, the instruction itself should be delayed the same amount in the MRT, in order to match m_i and m_0.

Thus, we can conclude that the instructions should be arranged in the MRT such that all instruction dependencies are arranged before itself. In other words, the instructions should be arranged in topological order.

Proposed Heuristic

The proposed heuristic main idea is to find an ordering to the instructions for scheduling that explores Sections 4.6 and 4.6 priorities. Algorithm 7 implements such heuristic with the following three main steps.

The first step creates a list with all cycles ψ (line 1) and calculates all s_ψ values (line 2). Cycles are sorted according to s_k (line 3). Instructions in the cycles are parsed into a unique list (a list without repeated elements) (lines 4 and 5), eliminating double entries from instructions that appear in more than one cycle (as i_5 which is shared by ψ_0 and ψ_2 in Figure 8). Finally, the unique list is topologically ordered using a depth-first order (lines 6 to 9).

The second step creates the topological order of all instructions that are not contained in cycles. The length of instruction I_i is set as the maximum length of all paths containing I_i (lines 10 and 11). Then, the DFG edges whose I_i is source are sorted accordingly to destination edge length (lines 12 and 13). A depth-first search is performed in the ordered DFG, giving a list of instructions ordered accordingly to their length (line 15). Finally, this list is appended to the ordered list if all its predecessors are already in the list (lines 17 to 19).

The third step creates the MRT. The ASAP schedule is calculated by solving formulation 4.1, resulting in the base congruence class values $m_i^0 = t_i^{asap}$ (line 20). The base m_i^0 value is updated by propagating delays from its predecessors (line 22). Non-resource constrained instructions are allocated in lines 23 and 24. Resource-constrained instructions are attempted to be allocated to MRT slot $m = m_i^0$ (lines 28 to 31). However, in case no slot is available, the congruence class is incremented and we count an increment delay (lines 27 and 32). Finally, the delay increments are recursively passed to all instructions that have i as predecessor (line 33).

Following the example in Figure 8 for the first steps, a topological order with increasingly slack priority for cycles $\mathbf{s}_\psi = \{\psi_0, \psi_1, \psi_2\}$ would be $order = \{I_1, I_5, I_6, I_2, I_3, I_7, I_8, I_4\}$, and for the

Algorithm 7: NIS algorithm.

input : Data-Flow Graph, Constraints, II
output : Valid MRT

```
/* Topological-slack order                                              */
create the set Ψ containing all cycles;
calculate s_ψ ∀ψ ∈ Ψ;
sort Ψ increasingly according to s_ψ;
for ψ ∈ Ψ do
    add all instructions i to list if i ∉ list;
for i ∈ list do
    if ∀j ∈ dep(i), j ∈ oList && j ∈ list then
        add i to oList if ;
        recursively add all i successors k to oList if ∀j ∈ dep(k), j ∈ oList && j ∈ list ;
/* Topological-path length order                                        */
for i ∈ O do
    length_i ← maximum length between all paths containing i;
for i ∈ O do
    sort j ∈ uses(i) decreasingly according to length_j;
clean list;
Fill list with the Deep First Search order;
for i ∈ list do
    if ∀j ∈ dep(i), j ∈ oList then
        add i to oList if ;
        recursively add all i successors k to oList if ∀j ∈ dep(k), j ∈ ocList;
/* Filling the MRT with oList                                           */
calculate the ASAP schedule;
for i ∈ oList do
    m_0 ← (t_i^s %II + delay_i)%II;
    if i is not resource constrained then
        MRT(m_0, 0) ← i;
    else
        m ← m_0; inc ← 0;
        while i is not allocated do
            for r ∈ 0 to a_k | i is type k do
                if MRT(m, r) is available then
                    MRT(m, r) ← i;
                    set i as allocated;
            inc ← inc + 1; m ← m + 1;
        recursively propagate inc to all i successors;
```

second step, suppose that $\mathbf{l_f} = \{1,2,3\}$ and the path $\{I_1, I_9, I_{10}, I_{11}\}$ has length 4, a topological order with decreasingly path length priority would be $order = \{I_1, I_9, I_{10}, I_{11}, I_5, I_8, I_2, I_3, I_5, I_6, I_7, I_4\}$.

4.7. Modulo Schedulers Comparison

In this section we present a detailed comparison between SDCS, ILPS, SMS, GAS, and NIS.

Chart 5 presents a comparison between the SDCS, ILPS, GAS, and SMS. ILPS has the worse solving scalability, which also has the worse scaling with the `loop size`, n. The optimization problems solved by GAS and NIS have a better asymptotic scaling than the ones solved by SDCS, as GAS and NIS do not add allocation related constraints to its problems as SDCS does (Section 4.1).

Chart 5 – Comparison between the state-of-the-art and proposed modulo schedulers.

	SDCS	ILPS	GAS	NIS	SMS
Problem type	SDC	ILP	SDC	SDC	N/A
Solve complexity	Polynomial	Exponential	Polynomial	Polynomial	N/A
allocation related constraints	grows during execution	$\mathcal{O}(n^2)$	0	0	N/A
# Problems solved	$\mathcal{O}(n^2)$	1	$\mathcal{O}(n)$	2	N/A
Optimal II guarantee	no	yes	no	no	no
Optimal Latency guarantee	no	yes	no	no	no

Source: Elaborated by the author.

The main SDCS source of inefficiency lies on its iterative heuristic search that tries to find a solution with a valid MRT, and on its incapability to prove infeasibility for a given II. The ILPS main source of inefficiency lies on its quadratic scaling `problem size`, and its exponential solver complexity. In contrast, GAS search solves a linear number of problems, achieving better complexity than SDCS. Furthermore, GAS problems are in the SDC form, maintaining a polynomial solving time. Finally, NIS solves a constant number of SDC problems, reaching further improvements in speed compared to GAS.

Concerning the quality of the achieved solution, SDCS does not guarantee latency or II optimality due to its heuristic nature. If a solution cannot be found within a pre-allocated time budget, the II is increased and a new search is performed. On contrary, ILPS guarantees both latency and II optimality, since the solver guarantees to find the optimal solution for the problem, for a given II, or declare infeasibility. GAS and NIS do not guarantee latency or II optimality since the GA may fail to find a feasible MRT before it reaches its stop criteria, and NIS can simple create an infeasible MRT.

This section presents three different comparisons between the schedulers for a set of benchmarks presented in Section 4.7.1. First, Section 4.7.2 presents a speed-latency comparison, demonstrating the proposed schedules number of SDC solved problems reduction and the latency and $total_{cycles}$ impacts. Second, Section 4.7.3 presents results when loop unrolling is applied

together with the loop pipelining, demonstrating the achieved time scaling desired by the local DSE and the hardware metrics impacts of the proposed methods. Finally, Section 4.7.4 present results relating the scheduler latency and hardware metrics, demonstrating that the proper DSE can mitigate the proposed schedulers negative impact in the hardware metrics.

4.7.1. Benchmark Selection

Table 3 provides the characteristics, number of operations per type, resource constraints, back-edge distance, and minimum *II* estimation for the HLS benchmark set designed by Canis, Brown and Anderson 2014 and Liu, Bayliss and Constantinides 2015. The benchmarks are challenging scheduling problems due to the number of resource-constrained operations and the data-path structures.

Table 3 – Benchmark selection and characterization.

Name	Short Name	Loop size	Arrays Size	$\frac{\#Operations}{Constraints}$ +/*/%/f+/f*/f%/[]	RecMII/ ResMII/ MII
multipliers	mt	28	100	8/2/0/0/0/0/7 3/1/x/x/x/x/1	3/4/4
dividers	dv	72	100	12/0/4/0/0/0/11 3/x/2/x/x/x/2	3/6/6
faddtree	fat	81	100	7/0/0/21/0/0/22 3/x/x/1/x/x/2	27/11/27
add int	ai	82	100	0/0/0/21/9/0/12 x/x/x/3/3/x/2	1/12/12
complex	cp	98	100	21/7/2/0/0/0/25 3/3/1/x/x/x/2	20/13/20
adderchain	ac	92	100	48/0/0/0/0/0/24 3/x/x/x/x/x/2	3/12/12

Source: Research data.

All schedulers were implemented as part of LUP infrastructure, where the SDCS is natively implemented. LUP imposes that all loop bounds and array sizes be statically determined, only the innermost loops can be pipelined, and conditional paths have to be merged into a single basic block. The whole code is entirely “inlined” to allow the usage of local memories. It should be noted that the above restrictions are due to LUP infrastructure and not due to the proposed method.

The results were obtained on a computer with Ubuntu 14.04, 16 GB of Random Access Memory (RAM), and an Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz. All schedulers were implemented within LUP 4.0 as LLVM 3.5 opt passes, using **Gurobi** 7.5 (Optimization 2017) as solver. SDCS budget is set to 6 (LUP default) and $INCREMENTAL\_SDC = 1$ is used, ILPS time budget is set to 10 minutes, and GAS is set as presented in Rosa, Bouganis and Bonato 2018.

To evaluate the different schedulers effects on the hardware quality, all designs in the scaling tests have been implemented using Quartus II 16.1, targeting a Stratix V board, with `OPTIMIZATION_TECHNIQUE=speed`, `auto_dsp_recognition off`, `dsp_block_balancing "logic elements"`, and `max_balancing_dsp_blocks 0` to remove the usage of hardcore DSP, avoiding influences in the Adaptive Logic Modules (ALMs) and Registers usage during this comparison.

4.7.2. Generated Schedules Comparison

Table 4 shows the number of problems solved for each benchmark. NIS solves only 2 SDC problems per loop, while SDCS and GAS solve several problems. ILPS solves only one ILP problem per loop per candidate *II*. Table 4 also presents the reached *II* value, latency, total number of cycles for loop completion, and total computation time for all schedulers. Table 4 last column shows the row `geomean` normalized according to the SDCS `geomean`. All values in Table 4 are the average of 50 repetitions. The results for SDCS, ILPS, and NIS are integers since they are deterministic methods, while GAS has a random nature.

SDCS, ILPS, GAS, and NIS were able to find the minimum *II* for all benchmarks. SMS fails for all benchmarks apart from **cp**, for which it achieves a larger *II* than the other approaches. Further investigation on the failing cases reveals that SMS fails in its second heuristic (i.e. the one used to construct a schedule). This happens if back-edge instructions have conflicts in the MRT, making them be reallocated in a way that violates the back-edge constraints, forcing the algorithm to stop and return a `failure`.

The obtained SMS results are not unexpected since list-based schedulers do not explore multiple solutions, making them unable to handle loops with complex data-flows leading to no solutions or to a schedule with larger *II*s when compared with methods that explore multiple solutions as shown in (Codina, Llosa and González 2002).

GAS and NIS achieve 20% and 16% worse latency than SDCS, making the total number of cycles for loop completion to be affected by 5% and 1%, respectively. When considering the total computation time, NIS is 100× faster than SDCS, due to its constant number of SDC problems solved. GAS is 3.17 times faster than SDCS. SMS is more than 100× faster than NIS for **cp** benchmark.

Even though results on Table 4 indicate that the larger latencies achieved by NIS and GAS have their impact amortized in the $total_{cycles}$, it is expected that pipelines with larger latency will create deeper pipelines, which have larger register pressures, implying in need of more registers and multiplexers in the final hardware (Llosa *et al.* 2001).

Table 4 – Performance and computation time comparison between the state-of-the-art and proposed modulo schedulers (50 repetitions average).

Name		mt	dv	fat	ai	cp	ac	geo. rate
# problems solved	SDCS	40	201	175	155	380	661	1
	ILPS	1	1	1	1	8	1	0.01
	GAS	17	71	174	62	120	95.64	0.37
	NIS	2	2	2	2	2	2	0.01
	SMS	fail	fail	fail	fail	N/A	fail	N/A
Initiation interval	SDCS	4	6	27	12	20	12	1
	ILPS	4	6	27	12	20	12	1
	GAS	4	6	27	12	20	12	1
	NIS	4	6	27	12	20	12	1
	SMS	fail	fail	fail	fail	21	fail	N/A
Latency (cycles)	SDCS	24	72	101	95	127	25	1
	ILPS	23	71	101	91	122	12	0.86
	GAS	26 .00	84 .00	118 .38	138 .73	155 .29	28 .00	1.20
	NIS	28	73	120	102	141	36	1.16
	SMS	fail	fail	fail	fail	121	fail	N/A
$Total_{cycles}(10^3)$	SDCS	0.38	0.31	1.15	0.19	1.81	1.09	1
	ILPS	0.38	0.31	1.15	0.19	1.80	1.08	0.99
	GAS	0.38	0.32	1.17	0.23	1.84	1.10	1.05
	NIS	0.38	0.31	1.17	0.20	1.82	1.10	1.01
	SMS	fail	fail	fail	fail	1,82	fail	N/A
Time (s)	SDCS	1.34	12.35	17.08	18.23	35.61	32.92	1
	ILPS	1.79	3.42 e2	1.19 e3	1.14 e2	1.17 e3	1.13 e3	1.62 e3
	GAS	0.46	2.23	5.51	1.76	5.24	3.36	0.18
	NIS	0.07	0.08	0.07	0.06	0.12	0.11	0.01
	SMS	fail	fail	fail	fail	0.001	fail	N/A

Source: Research data.

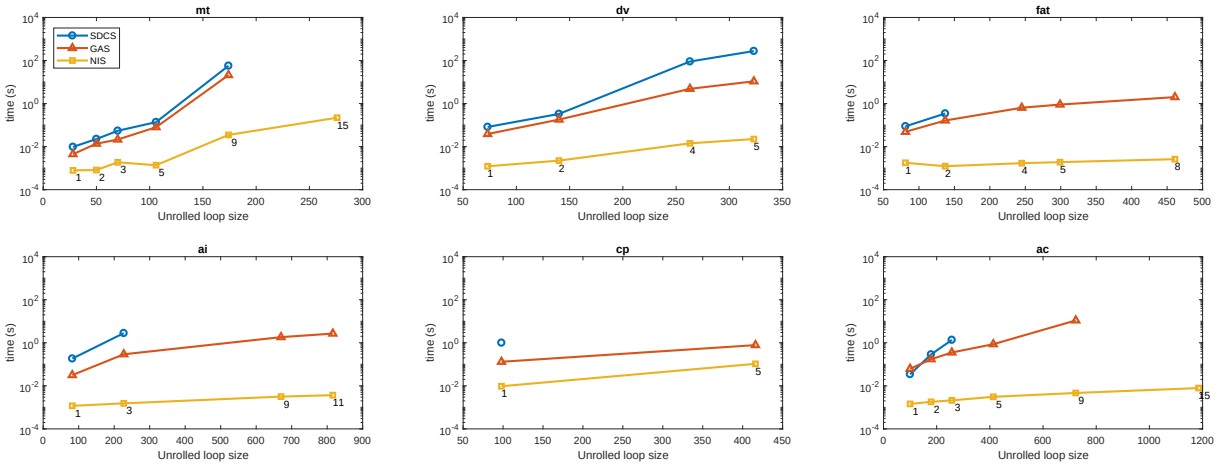
4.7.3. Scaling Comparison

To exemplify scalability, Figures 9 and 10 present the computation time and achieved *II*, respectively, as a function of the `loop size` when `Loop Unrolling` is applied (unrolled loop size) before pipelining. Local memories are utilized to avoid memory bottlenecks, improving performance. The unroll factor is presented besides NIS marks. Missing points mean that the scheduler fails to find a schedule in a 1-hour budget. ILPS results are not presented for this experiment given its poor scalability.

Figure 9 shows how the NIS gain in speed increases with the `loop size`, what is

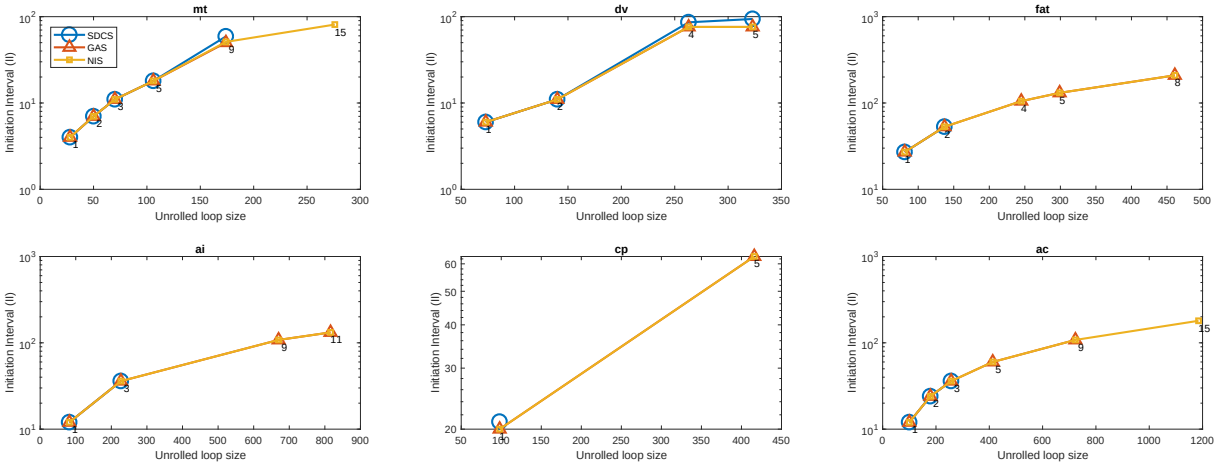
expected given its constant number of SDC problems solved. Figure 10 shows NIS and GAS can achieve better *II*s than SDCS for large loops, indicating that NIS and GAS can improve the $total_{cycles}$ when compared to SDCS for very large and complex loops. Figures 9 and 10 missing points indicate that GAS and NIS can explore the solution space better than SDCS.

Figure 9 – Computation time in function of the unrolled `loop size`.



Source: Research data.

Figure 10 – *II* obtained as a function of the unrolled `loop size`.

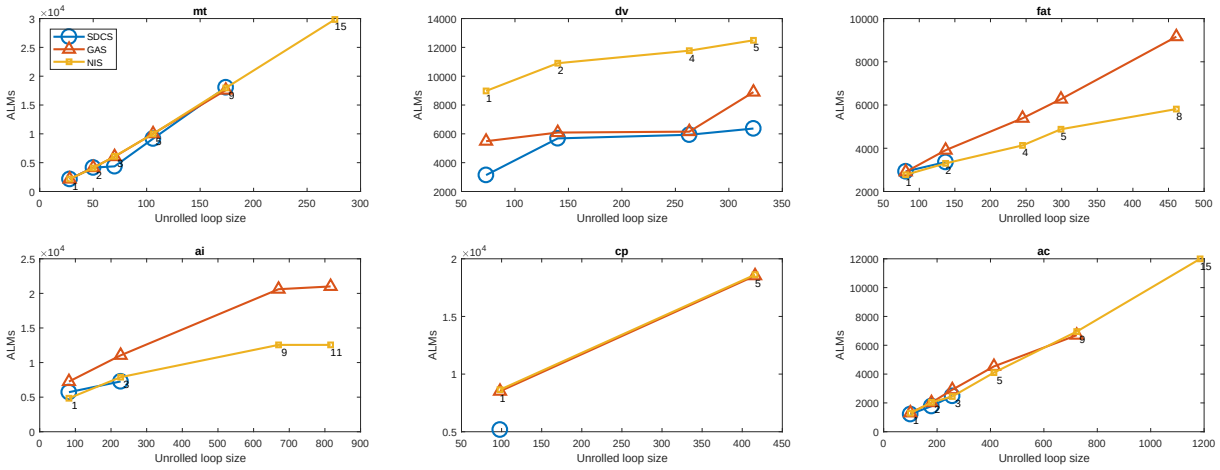


Source: Research data.

Figures 11, 12, and 13 present the number of ALM, registers usage and maximum frequency achieved by the designs according to the unrolled `loop size` (10 repetitions average). The differences in Figures 11, 12, and 13 are explained by the fact that the schedule impacts strongly the register pressure (Llosa *et al.* 1998), reflecting in the multiplexes network, maximum frequency, and possible optimizations performed by the synthesis tool.

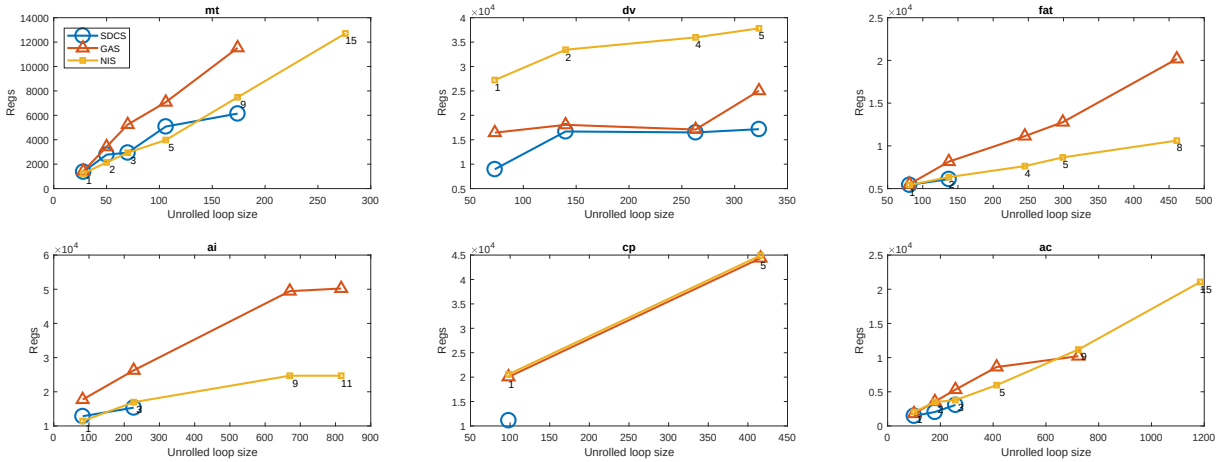
Figures 11 and 12 show that SDCS results in fewer hardware resources in most cases (when it can create a schedule), indicating a better register pressure than the other schedulers. NIS obtains results similar to SDCS in all benchmarks apart from **dv**, where its resources usage

Figure 11 – ALM usage as a function of the unrolled `loop size`.



Source: Research data.

Figure 12 – Registers usage as a function of the unrolled `loop size`.



Source: Research data.

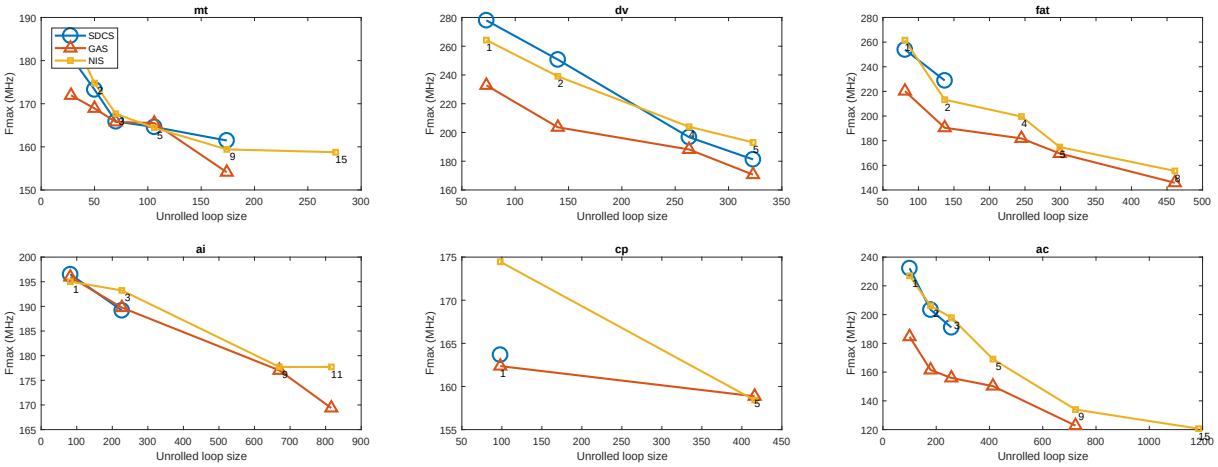
is 2× larger than the other schedulers. This occurs because NIS does not optimize for latency as much as GAS evolution or SDCS heuristic, resulting in a deeper pipeline which degrades increasing hardware resources usage (Dai *et al.* 2017).

GAS presents worse or similar results to NIS in all benchmarks apart from **dv**, what is explained by the fact that it quickly tries to find a valid and feasible solution, without optimizing the register pressure. NIS ordering heuristics allocates instructions and its dependencies close to each other, which reduces the register pressure (Llosa *et al.* 2001).

Figure 13 shows that NIS does not strongly impact the maximum frequency when compared to SDCS, while it also shows the negative impact on the maximum frequency caused by the GAS larger register pressure.

Even though unrolling the loop prior to pipelining might reduce the number of clock cycles required to complete the loop execution, Figures 11 and 12 show that the negative impact

Figure 13 – Achieved maximum frequency as a function of the unrolled `loop size`.



Source: Research data.

in hardware resources and frequency can restrain the overall performance. Nevertheless, the usage of unrolling together with pipelining in this paper has its purpose of demonstrating the NIS scalability and final hardware quality impact, as presented in Rosa, Bouganis and Bonato 2018.

4.7.4. Resources Constraints, Latency, and Hardware Metrics

In this section we explore the relation between schedule latency, resources constraints and hardware resources usage, which is important to evaluate in which conditions GAS and NIS do not have a significant negative impact in the final hardware resources usage and how this reflects in the local DSE.

In order to demonstrate the relation between schedule latency and hardware resources usage, we selected the benchmark **dv** as an example, which ALM and Registers usage strongly affected by the schedule latency according to empirical tests, and evaluate several schedules with different latencies. An unroll factor of 2 is used to make the problem complex enough for the schedulers to return significantly different scheduler.

To create schedules with different latencies, ILPS, SDCS, GAS and NIS were used with the configurations as described in Chart 6.

Chart 6 – Schedulers configurations to obtain schedules with different latencies.

ILP*	ILPS without time budget, which provides the optimal solution
ILP-10	ILPS with 10 minutes time budget;
GAS{-0.5,-1,-2,-3}	GAS with $\alpha = \{0.5, 1, 2, 3\}$, respectively
NIS	NIS with the topological order presented in Algorithm 7
NIS-DFS	NIS with a Deep-First Search topological order

Source: Elaborated by the author.

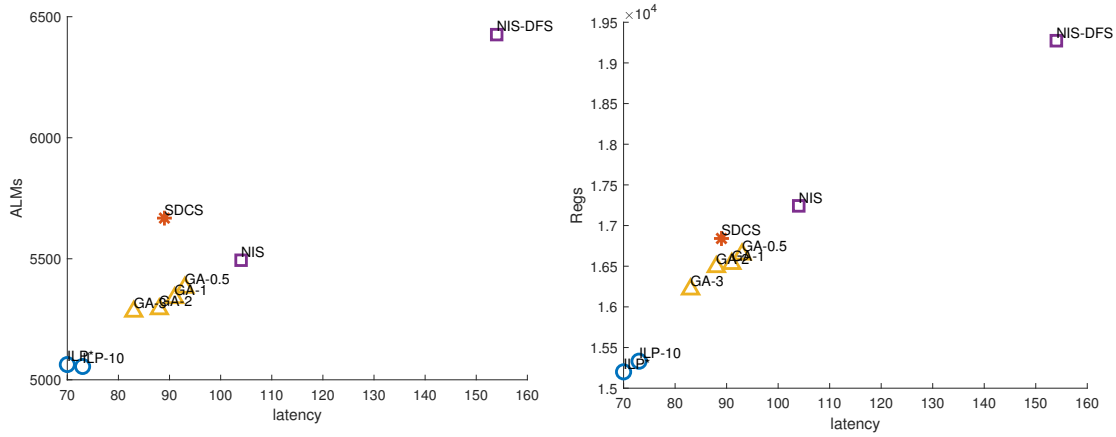
Figures 14a and 14b present the ALM and registers usage in function of the latency for each schedule. Even though the results in Table 4 show that the latency has a small impact in the $total_{cycles}$, Figure 14 indicates that it might affect strongly the hardware resources used, what makes the overall optimization a more complex problem.

These effects indicate that NIS is more adequate for DSE early stages, where NIS faster computation time benefits the comparison of many different designs configurations, while a more robust scheduler, as ILPS, should be used in a fine optimization later stage of hardware development.

Figure 14 – Hardware usage scaling with latency for benchmark **dv**.

(a) ALMs usage scaling with latency for benchmark **dv**.

(b) Register usage scaling with latency for benchmark **dv**.



Source: Research data.

The correlation between latency and hardware resource usage observed in Figures 14a and 14b does not imply that a smaller latency results in a smaller register pressure, as demonstrated by `SDCS` point which uses more resources while has larger latency than `GAS-1`, `GAS-0.5`, and `NIS` points.

It is important to notice that the resources constraints presented in Table 1 (the same as used by Canis, Brown and Anderson 2014 and Rosa, Bouganis and Bonato 2018), were fixed arbitrarily, implying that the extra resources usage due to schedules with larger latency in Figure 14 is purely an overhead caused by the larger register pressure.

In this scenario, it is desirable to increase the resource constraints, which adds more columns to the MRT. The consequences of having more resources available are: a reduction of conflicts during the scheduling, schedules with a shorter path (smaller latency), a possible *II* reduction (for *RecMII* constrained benchmarks). Thus, increasing the resources constraints can significantly improve the overall number of cycles for loop completion However, to choose the best number of resource constraints, a full resources constraints DSE must be performed.

To demonstrate that using the right set of resources constraints we can diminish the

schedulers hardware metrics impact we evaluate an exhaustive search over all combinations of resources constraints using ILPS, SDCS, GAS, and NIS. Then, we calculate the Euclidean distance between the hardware metrics obtained with SDCS, GAS, and NIS and the hardware metrics obtained with ILPS.

Figure 15 presents the histograms showing how many designs there are aggregated by the distance between the hardware metrics created using SDCS, GAS, and NIS and the hardware metrics created using ILPS (used as reference due its optimality) for the same resources constraints configuration.

We do not present results for **ai** benchmark since its configuration space has more than 100000 configurations, making unpractical to perform an exhaustive search.

Figure 15 shows that there are resources constraints configurations that result in small hardware metrics impacts for all schedulers, indicating that the hardware metrics impacts observed in Figures 11 and 12 can be mitigated with proper resources constraints values definition.

Furthermore, Figure 15 gives us an idea about the negative hardware metrics probability function for each scheduler (supposing random configurations), which resembles a half-Gaussian distribution. In this picture we can see how the standard deviation is higher for GAS and NIS. As such, the standard variation can be used as a metric to measure and compare the schedulers impact in the hardware metrics.

The DSE can be accelerated by exploring the design space with NIS (or SDCS and GAS). The resulting Pareto-optimal solutions configurations can be used with ILPS to create an Pareto-optimal approximation. To demonstrate the schedulers DSE impact in the final hardware metrics, we calculate the Pareto-optimal points for the exhaustive search using ILPS, which will serve as reference set. Then, we calculate Pareto-optimal points for the exhaustive search using SDCS, GAS, and NIS, and use the correspondent configurations to create designs using ILPS, resulting in the Pareto-optimal approximation for each scheduler. Finally, the two sets are compared with the Average Distance from Reference Set (ADRS).

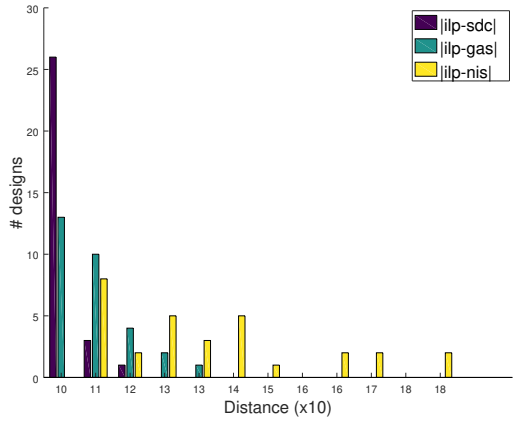
The ADRS is defined by Equation 4.6, where $\Gamma = \{\gamma_1, \ldots, \gamma_n\}$, and $\Omega = \{\omega_1, \ldots, \omega_m\}$ are the reference points and approximated sets, respectively. In Equation 4.6, first the distance from a point in the reference set is calculated with all approximated set points. The minimal result is accumulated for all reference set points and averaged. Thus, the closer ADRS is to 0, the closer the Pareto curves are from each-other. Note that points repeated in the same set reduce the ADRS. Thus, we measure the ADRS only between unique points in the reference and approximated sets.

$$ADRS(\Gamma,\Omega) = \frac{1}{|\Gamma|}\sum_{\gamma\in\Gamma}\min_{\omega\in\Omega}\left[\max\left\{\frac{a_\omega - a_\gamma}{a_\gamma}, \frac{l_\omega - l_\gamma}{l_\gamma}\right\}\right] \tag{4.6}$$

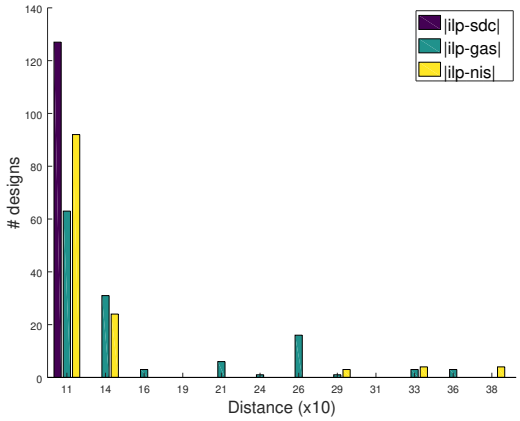
Table 5 presents the ADRS reference set (using ILPS) and the approximated sets (using

Figure 15 – Number of resources constraints configuration grouped by SDCS, GAS, and NIS hardware metrics distance from ILPS hardware metrics.

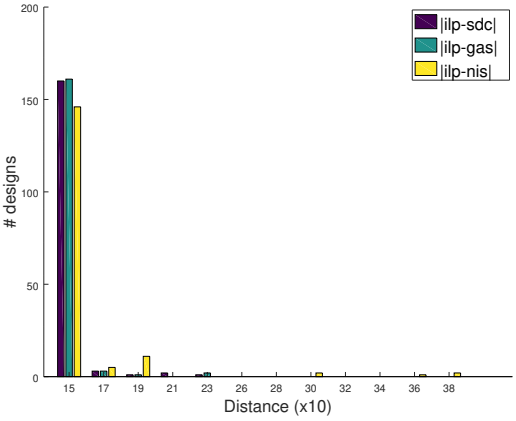
(a) **mt**.



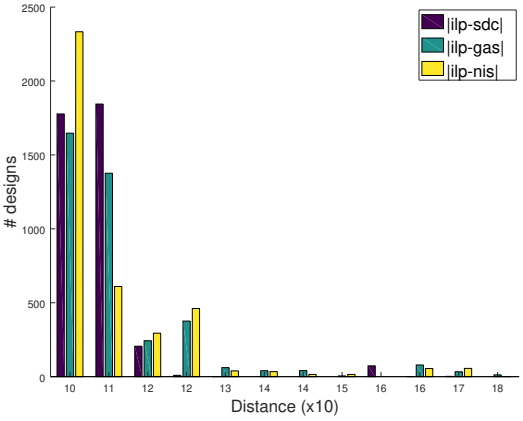
(b) **dv**.



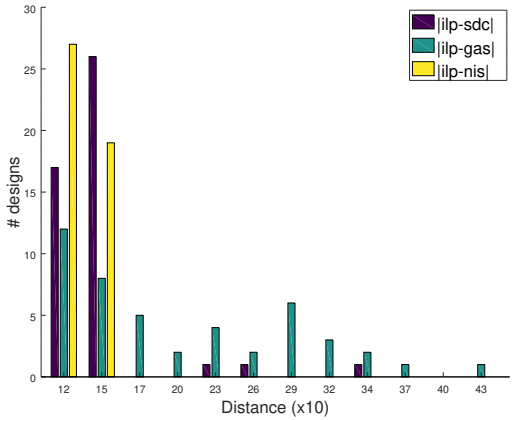
(c) **fat**.



(d) **cp**.



(e) **ac**.



Source: Research data.

SDCS, GAS, and NIS), showing SDCS, GAS, and NIS impact the design space by less than 1%. The average distance is measured as $avgDist = 100 - geomean_{i \in benchmarks}(100 - ADRS_i)$ since the *geomean* requires values greater than zero.

Table 5 – ADRS (%) between ILPS Pareto-optimal solutions and SDCS, GAS, and NIS Pareto-optimal solutions compiled with ILPS.

Benchmark	**mt**	**dv**	**fat**	**cp**	**ac**	*avgDist*
SDCS	0.000	0.957	0.597	0.322	0.000	0.376
GAS	0.170	0.584	0.308	1.026	1.887	0.797
NIS	1.481	1.505	0.704	0.467	0.804	0.993

Source: Research data.

Even though NIS results in an DSE ADRS 2.65× higher than SDCS, design space approaches consider $ADRS < 5\%$ as low values, and $ADRS < 1.7\%$ as extremely low values (Schafer 2016). Thus, we can safely assert the NIS adequacy for speeding-up the DSE due to its speed-up compared to the other schedulers and the less than 1% impact in the final hardware resources.

4.8. Final Remarks on Modulo Schedulers

The modulo scheduling methods in the literature are iterative approaches which dominate the HLS time complexity, becoming a local DSE bottleneck. This bottleneck comes from the excessive number of SDC problems solved by the methods, which were the acceleration target in this chapter.

The proposed methods explore the explicit separation between the problem scheduling and allocation parts created by the proposed SDC formulation. As such the proposed GAS and NIS achieve an asymptotic reduction the in loop pipelining computation time, removing it as a local DSE bottleneck.

Concerning the final implementation quality, the results demonstrate how latency and hardware resources usage relationship can impact the proposed methods usage. However we also demonstrate how this impact can be mitigated by defining proper resources constrains values, which also motivated the local DSE.

CHAPTER

5

LOCAL DESIGN SPACE EXPLORATION

Recapping from Chapter 3, having a fast local DSE method has been identified as a key to enable the local DSE, which was restricted by the loop pipelining computation time, especially when applied alongside loop unrolling. Chapter 4 presented two new approaches for reducing the loop pipelining scaling complexity, removing it as a DSE bottleneck. As such, we now focus this thesis back on the local DSE acceleration.

Section 5.1 presents in-depth state-of-the-art approaches and analyses the literature to identify its current gap, which show how the complex relationships between configuration and design spaces hurt the performance of the methods. Such analyses culminate in an individual directives study proposal, which allows mitigating the complex relationships between configuration and design spaces. As such, we propose to study loop pipelining, unrolling, and resources constraints, which are three of the most impactful and common directives. The rest of this chapter is organized as follows:

Sections 5.2, 5.3, and 5.4 present proposed "white-box" approaches to explore the resources constraints, loop pipelining and unrolling directives, respectively. Finally, Section 5.5 concludes this chapter.

5.1. Local DSE Inconsistencies Analysis

This section presents an in-depth analysis of the literature DSE methods aiming to highlight their inconsistencies sources and strengths. These information can be incorporated into our proposed approaches to obtain better results.

Recapping from Section 2.2, the local DSE methods can be categorized according to their DSE speed-up mechanics as "Smart Selection", "Modelling", and "Architecture Fitting" methods, which allowed us to conclude that both "Smart Selection" and "Modelling" are required to achieve practical local DSE times without compromising the method with non-portable architectures.

"Architecture Fitting" methods will be spared from further analysis due to their portabilities restrictions.

We separated the works presented on Section 2.2 in three groups, summarizing the speed-up, results quality, and explored directives. The directives acronyms are presented on Chart 7, and the three groups on Charts 8, 9, and 10.

Chart 7 – Directives acronyms for Charts 8, 9, and 10.

Acronym	Directive	Acronym	Directive
fi	function inline	rc	resources constraints
fb	function branch type	mg	memory ports
tb	tree balancing	mp	array partition
lu	loop unrolling	mm	array mapping
lp	loop pipelining	md	memory duplication
lf	loop fold	mr	memory reorganization
lt	loop tiling	mc	cache

Source: Elaborated by the author.

The works on Chart 8 are a continuous evolution of the Cyber compile DSE, focusing on reducing the number of evaluated designs, and also using ML techniques to create hardware metrics estimations. As such, these works present a progression being PMK (Schafer 2016) the one with better speed-up and smaller error.

The works on Chart 9 focus on creating hardware metrics estimations, which are used to speed up the local DSE through an exhaustive search. The estimations depend on specific computation models, hardware architectures, compilers or tools, and can only model a sub-set of HLS directives as loop unrolling, pipelining, data-flow and array partitioning (Ferretti, Ansaloni and Pozzi 2018). As such, these approaches fall out of this thesis scope alongside the hardware metrics estimations.

The works on Chart 10 use heuristics to reduce the number of evaluated designs, speeding-up de DSE. These works target miscellaneous compilers, and their portability is not constrained to specific platforms or environments.

The approaches on Charts 8, 9, and 10 present as motivation for the ML methods used the design space size non-trackability and how unpredictable are the relationships between configuration and design spaces. Such unpredictability is expected since many different directives are put together to form the design space without considerations about the effects of each directive in the code, design, and configuration space.

Between the presented approaches, only PMK (Schafer 2016) treats directives differently. The directives are divided into three knobs. However, this division is motivated by the way that the directives are applied in the code, and not their design impacts.

Chart 8 – Comparison Between Cyber compiler DSE methods.

Name	Type	Synthesis	Speed-up	Quality	Over	Directives
ASA	simulated annealing	yes	3.80×	+5% area, +8% latency	exhaustive	fi, fb, mm, lu, lf
MLP	GA + ML	yes	2.04×	0.62% dominance	ASA	
DC-ExpA	heuristic	yes	3.33×	0.88% dominance	ASA	
PMK	heuristic + swarm	yes	13×	$ADRS = 1.7\%$	ASA	
ERDSE	GA + ML	yes	9.8×	$ADRS = 10\%$	ASA	

Source: Elaborated by the author.

Chart 9 – Comparison Between estimation-only DSE methods (N/R - not reported).

Name	Compiler	Synthesis	Speed-up	Error	Over	Directives
PBDRO	Rose	yes	N/R	N/R	N/R	mc
Aladdin	ILDJIT	no	445×	6.5% error	synthesis	rc, lu, lp, mg
Lin-Analyzer	LLVM	no	123×	5.2%	synthesis	lu, lp, mp
MPSeeker	LLVM	no	600×	12.8%	synthesis	lt, lu, lp, mp
AutoAccel	Merlin	partial	N/R	13.5%	synthesis	lp, mc, mr
MKDSE	SDCAccel	N/R	N/R	N/R	N/R	lu, lp, mp

Source: Elaborated by the author.

Chart 10 – Comparison Between misc. DSE methods (N/R- not reported, N/A - not applicable).

Name	Compiler	Type	Synthesis	Speed-up	Quality	Over	Directives
SDSE	AUGH	heuristic	no	19×	+8.8% latency	(Corre *et al.* 2012)	rc, lu, mm
$(AS)^2$	Vivado	arch. matching	yes	N/R	N/R	hand coded design	md, mp, mm, tb
PSCEFP	N/R	heuristic	yes	2091×	+3.3% latency	exhaustive	PolyOpt
AO-DSE	Vivado	heuristic	yes	N/R	N/A	exhaustive	fi, mp, lp, rc
SPIRIT	Spark	heuristic	yes	200×	$ADRS = 2\%$	ASA	N/R
LBDSE	Vivado	lattice	yes	132×	$ADRS = 1\%$	exhaustive	lu, fi, mg

Source: Elaborated by the author.

Such unpredictable relation between the configuration and design spaces should be expected. Next, we present a list of unpredictability sources:

1. Directives act on different scopes. As an example, loop unrolling and pipelining target specific loops, while resources constraints affect the whole kernel code.

2. Local optimal directives are not optimal for the whole kernel. As an example, given a kernel with two loops, the optimal resources constraints configuration for each loop are (probably) different, but if both loops are implemented within the same kernel, another configuration will (probably) be optimal.

3. Directives scope intersection. As an example, memory architecture optimizations are kernel-wide, but they affect loops implementations if the memories are used within the loop body.

4. Directives are non-orthogonal, meaning that the same design can be obtained with two different sets of directives.

5. The directives application order impacts the final design implementation. By e.g., if applying loop interchange before unrolling and vice versa.

6. Treating different directives, in the same manner, creates false neighbourhood relations in the design space.

Summarizing, many different optimizations are mixed-in, creating a complex relationship between the configuration and design spaces, which is handled with “black-box” approaches, which are convenient due to its extendability and portability.

Going against such “black-box” approaches, we propose to consider each directive individually, analysing the relationships created between the configuration and design spaces. Such relations should not be unpredictable since all above-listed sources, apart from 2, come from mixing different directives. Furthermore, such analyses can unveil proper directive application orderings, handling the point 5.

As such, we expect that a DSE paradigm shift from “black-box” to a series of “white-box” approaches will enable to reach higher speed-ups and/or better accuracy in the local DSE methods.

Directives that are not yet or cannot be analysed with “white-box” approaches can still be explored with traditional “black-box” approaches separately, making the proposed paradigm shift not harmful to DSE approaches portability and extendability. Furthermore, interactions between different directives can be identified with a “white-box” analysis, allowing the exploration of such relations to improve the DSE. As such, for each directive that can be successfully analysed as a

"white-box", the configuration space is reduced combinatorially in size and an unpredictability source is removed.

Next, we go through the state-of-the-art approaches, highlighting where their strengths and inconsistencies are. Section 5.1.1 analyses approaches which take decisions based on design space gradients, which is a common practice in many approaches on Charts 8 and 10. Section 5.1.2 analyses the PMK, which presents the best results for the Cyber compiler. Finally, Section 5.1.3 analyses the LBDSE, which Lattice nature presents advantages when compared to the other heuristic based approaches.

5.1.1. Gradient-Pruning Based Approaches

Gradient-pruning based approaches are greedy heuristics that, given a DSE point, compute (compile, estimate, or measure) the hardware metrics for all its neighbours. Then, the point resulting in a better gradient is chosen as the next point and the search iterates. SDSE, AO-DSE, and ERDSE are examples of such approaches.

This type of approach is suitable when all points in the design space form a typical Pareto curve, where reducing the resources constraints results in smaller and slower designs. However, reducing the resources constraints in a loop can have different results, as achieving the same design (if the resource is not a bottleneck), achieving a smaller and slower design (if the resource is a bottleneck), or even resulting in a larger and slower design (due to increasing overhead, which will be discussed in Section 5.3).

Gradient-pruning based approaches make decisions based on weights given to each directive, which are defined by the hardware metrics gradient between the current design and its neighbours. The gradient function can be written as Equation 5.1, where, da_r and dt_r are area and time differences between the current design and the design after applying transformation T for each directive type $r \in \mathscr{R}$; K_{curr} is a constant calculated from the current design metrics; gw_T is a short-name for the global weight.

$$\nabla G = K_{curr} \left[\frac{dt}{da} \right] \tag{5.1}$$

In the literature, gw_T is calculated for all directives. However, this leads to two critical flaws.

The first flaw is that two configurations which vary only in one directive value are not necessarily neighbours. E.g., a design implementation with and without pipelining result in significantly different speeds and hardware resources usage, but they are considered neighbours by these approaches.

The second flaw appears when we consider Equation 5.1 and the possible da and dt results, as the following scenarios:

1. $da = 0$ **and** $dt = 0$: $\nabla G = \frac{0}{0}$;

2. $da = 0$ **and** $dt > 0$: $\nabla G = \infty$, and d_n dominates d_c;

3. $da = 0$ **and** $dt < 0$: $\nabla G = \infty$, and d_c dominates d_n;

4. $da > 0$ **and** $dt = 0$: $\nabla G = 0$, and d_n dominates d_c;

5. $da < 0$ **and** $dt = 0$: $\nabla G = 0$, and d_c dominates d_n;

6. $da > 0$ **and** $dt > 0$: $\nabla G > 0$, and d_n dominates d_c;

7. $da < 0$ **and** $dt < 0$: $\nabla G > 0$, and d_c dominates d_n;

8. $\frac{da}{|da|} = -\frac{dt}{|dt|}$: $\nabla G < 0$, d_n and d_c trade-off area and speed.

In scenarios 2 and 3, we have that transformations create a better, and a worse design, respectively, however, ∇G cannot tell the difference between designs due to the singularity. In scenarios 4 and 5, we have the same problem, but with $\nabla G = 0$. In scenarios 6 and 7, we have the same problem again, but ∇G cannot differ good and bad designs due to its maximizing nature and the division signal cancelling. Finally, in scenario 8, where different Pareto-points are reached, $\nabla G < 0$, making transformation T not likely to be applied.

To reduce the second flaw inconsistencies, SPIRIT (Xydis *et al.* 2015) uses a signal model, which is more robust than Equation 5.1. The signal $s[x]$ is defined as Equation 5.2, where $s[x]$ is a "signal" representing the area-time product, and x is the discrete vector formed by the resources constraints. As such, minimizing $s[x]$ tends to the Pareto-optimal curve direction.

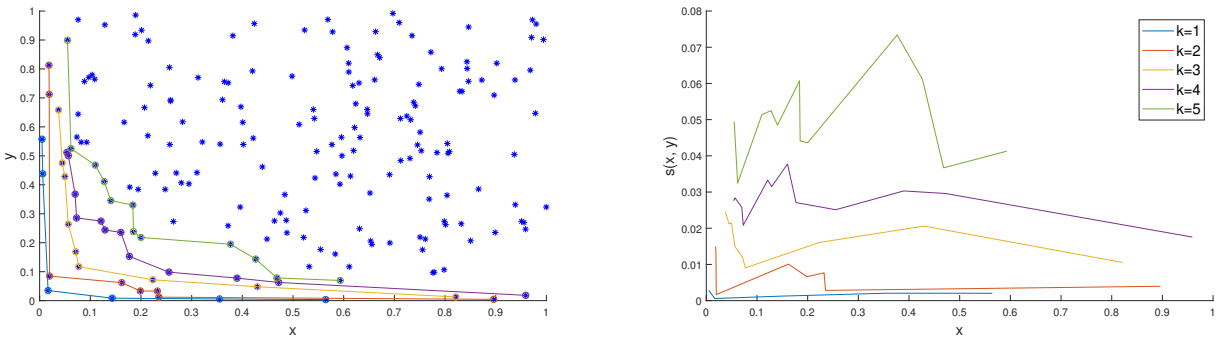
$$s[x] = Area(x) \times Cycles(T) \tag{5.2}$$

However, minimizing $s[x]$ does not imply in transversing all Pareto points. Figure 16 presents a random space example and its k-Pareto curves on the left, and the signal $s(x,y) = xy$ for the respective curves on the right.

Figure 16 presents the first 5 Pareto curves over a set of random points and their respective signal strengths, demonstrating the general trend of approaching the Pareto optimal curve by minimizing $s(x,y)$. Furthermore, $s(x,y)$ has many peaks, meaning that minimizing $s(x,y)$ does not imply in transversing a path between all Pareto-optimal points.

Furthermore, we can write the signal of the design obtained after applying a transformation T as Equation 5.3.

Figure 16 – Set of points and k-Pareto curves on the right. Corresponding signal $s(x)$ on the left.



Source: Elaborated by the author.

$$
\begin{aligned}
s[d_n] &= Area(d_n) \times Cycles(d_n) \\
&= (Area(d_c) - da) \times (Cycles(d_c) - dt) \\
&= s[d_c] - Area(x_c)dt - Cycles(x_c)da + dadt \qquad (5.3)
\end{aligned}
$$

Here we can see that, in scenario 1 the signal does not change, as expected. In scenarios 2, 4, and 6, $s[x_T]$ increases, and in scenarios 3, 5, and 7, $s[x_T]$ decreases, being consistent in differentiating better and worse points. However, in scenario 8 we again consider $Area(x_c) > da$ and $Cycles(x_c) > dt$, $s[x_T]$ is negative, and instead of reaching a possible new Pareto point, the $s[x_T] < 0$ inconsistently interprets the point as a worse configuration.

Scenario 8 is the most important scenario since it is the one which represents the finding of a new point "possibly" in the same Pareto curve. Furthermore, gradient-pruning based approaches also suffer from local traps in the design space.

5.1.2. The Probabilistic Approach

The PMK probabilistic model (Schafer 2016) results from the Cyber compiler DSE evolution, which is detailed in this section, alongside with its steps discussions and strengths highlighted. The PMK main idea is to divide the directives into three knobs, the local knob, #FUs, and global knob (already discussed in Section 2.2) depending on how they are applied.

The approach is divided in two parts. First, an Ant Colony Optimization (ACO) (Wang 2007) is used to explore the local knob. Each configuration is synthesized with minimum and maximum #FUs (D_{min} and D_{max}, respectively), which gives an idea about the design space region containing all designs generated by varying the number of FU. Then the ACO is performed returning a set of dominant local knob configurations to the second step.

Second, the probabilistic model is used to explore the #FUs knob, where each local knob configuration is associated with a probability function proportional to the area between D_{min} and

D_{max}. The probability function sets large areas with high exploration priority, and overlapping areas low with priority. Finally, to explore the designs in each area, the resources constraints are varied from D_{max} to D_{min} by 10% intervals, which has empirically shown to provide a good balance between exploration speed and results quality for the presented tests.

We refrain from presenting PMK full algorithms for conciseness reasons since we are interested only in the above observations.

This is the only approach in the literature that does not analyze all directives in the same manner. However, directives are classified according to their application scope and method, resulting in a "black-box" approach. Furthermore, PMK uses a simple linear distribution to explore the #FUs, which can lead to sub-optimal results since bottleneck balancing is not considered. However, it has shown to be efficient at avoiding local-optimal plateaus.

The promising results obtained by PMK indicate that a clear separation between directives help to improve the DSE methods performance, corroborating with our "white-box" paradigm proposition.

5.1.3. The Lattice-Based Approach

The LBDSE lattice-based approach (Ferretti, Ansaloni and Pozzi 2018) is the first to observe relationships between the design and configuration spaces. Through a Principal Component Analysis (PCA), it is empirically observed that Pareto-optimal points cluster in the configuration space, while they are spread in the design space. To explore such characteristic, the configuration points are normalized, and the lattice-based exploration presented in Algorithm 8 is realized over the design space.

Algorithm 8 first uses an U-distribution to select random configuration points (line 1), which are synthesized (line 2). This distribution makes that designs with more extreme values are more likely to be chosen. From this distribution, m individuals are sampled in the algorithm initialization part.

In the iterative part, the Pareto optimal points (between the points that have already been visited) are selected and have their σ-neighbours synthesized (lines 7 to 9). The selection and synthesis process is iterated until no new Pareto optimal points are found, or a budget is exceeded (lines 13 to 15). Finally, all compiled designs configurations and results are returned (lines 5, 6, 10, 12, and 16).

As such, LBDSE has two parameters to control the number compiled designs and the results quality trade-off: m, which controls the initial sampling number; and σ, which controls the how many neighbours will be explored per point. These parameters must be adjusted for the target compiler and directive sets.

By considering and exploring that Pareto-optimal points cluster in the configuration

Algorithm 8: Lattice-based DSE algorithm.

```
   input  : List of directives and possible values
   output : List of configurations and metrics
 1 configs ← m U-distribution random configurations;
 2 designs ← synthesize and annotate area and cycles;
 3 nextConfigs ← pareto(configs, designs);
 4 count ← size(configs);
 5 compiledConfigs ← configs;
 6 compileDesigns ← designs;
 7 do
 8     configs ← σ-neighbours of nextConfigs;
 9     designs ← synthesize and annotate area and cycles;
10     nextConfigs ← pareto(configs, designs);
11     append configs to compiledConfigs;
12     append designs to compileDesigns;
13     if count ≥ budget then
14         break;
15 while nextConfigs ≠ ∅;
16 return compiledConfigs and compileDesigns;
```

space, LBDSE is the first approach that explores the relationships between the configuration and design spaces in order to realize a local DSE speed-up method. This strongly motivates our proposed approach of studying and evaluating such relations for each directive individually.

The major flaw in this approach lies within its U-shaped distribution initialization, which tends to create fast-and-large and slow-and-small designs only, leaving gaps that balance the area and number of clock cycles reduction susceptible to local plateau traps.

5.2. Stand-alone Resources Constraints Exploration

Resources constraints are the most basic type of directives, which control how many functional units the design will implement, influencing area and speed (number of clock cycles) trade-off. However, this is not a straightforward relation since the resources constraints affect the instructions schedule, which affects the number of clock cycles and hardware resources usage.

Using the Section 5.1 observations, we propose two new approaches for exploring the resources constraints, the first explores neighbour relations in the configuration and design spaces, and the second is an LBDSE improvement, both aim to reduce the number of evaluated designs and while keeping comparable accuracy.

5.2.1. Proposed Approach

The main idea behind the proposed approach is that Pareto points will be close to each other if they are in the same design space plateau. However, different plateaus will vary in proportions between the resources constraints. As such, we proposed a neighbour search (as used by LBDSE) until a plateau is found, then we perform the 10% reduction rule (used by PMK) to escape such local optima traps.

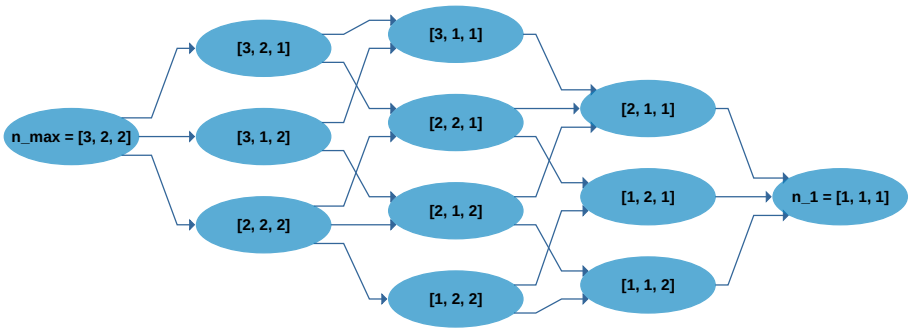
The proposed approach tries to find a path between two extreme points, one with no resources constraints, and another with FU constrained to 1. By evaluating points in this path, we expect to find Pareto-optimal designs in the same fashion as the gradient-pruned based searches. Moreover, by evaluating a whole path connecting these points, we expect to reach transverse through plateaus containing Pareto-optimal solutions.

We define the Design Space Graph (DSG) as a graph connecting the two extreme points as follows:

Give the set of resources to be constrained $K = \{add, mult, div, \ldots\}$, each one with maximum values a_k defined by the number of instructions in the code, we define the resources constraints graph $R = \{N, E\}$, where each node $n \in N$ is a unique array $e = \{e_1, e_2, \ldots\}$ with $1 \leq e_i \leq a_i$, and each edge $\exists\ e = \{n_i, n_j\}$ if, and only if $|n_1, n_2| = 1$. If $\exists\ e = \{n_i, n_j\}$, we say that n_i and n_j are neighbours. the graph has the nodes $n_{max} = \{a_1, a_2, \ldots, a_{|K|}\}$ and $n_1 = \{1, 1, \ldots, 1\}$.

Figure 17 presents a DSG example with $|K| = 3$ and $n_{max} = \{3, 2, 2\}$. Note that the graph has $|N| = \prod_{i \in 1}^{|K|} a_k$ nodes, representing the whole design space formed by the resources constraints.

Figure 17 – Example of resources constraints design space graph.



Source: Elaborated by the author.

Each unique resources constraints configuration is a DSG node connected to all its neighbours. The resources constraints DSE objective is to find a DSG path between the points n_{max} and n_1 visiting the maximum number of Pareto points as possible.

The DSG transversal is made by evaluating the current point neighbours to select the

next one, similarly to Gradient Pruning approaches, and to use a set of conditionals varying if $da, dt\{=,>,<\}0$, to realize if neighbour points are better, worse, or different Pareto points. By using the conditionals, we avoid the inconsistencies presented by the signal and gradient models, at the cost os 8 if-then-else statements.

To avoid local plateaus, we include the 10% variation rule from Schafer 2016, which will generate nodes with different resource constraints proportions. As demonstrated, these points have a higher probability of belonging to different plateaus.

Algorithm 9 presents the proposed DSG Path-based DSE (PBDSE) approach. First, n_{max}, n_1 and $diff10$ constants are defined, along with queues to annotate which designs were compiled and discarded (lines 1 to 6), and a queue to save which are the possible configurations to be searched next (line 7).

At this point, it is important to notice that the hardware resources usage and clock cycles can be obtained by synthesis, simulation, or estimation. We use hardware elaboration and behavioural simulation to obtain the hardware metrics, which is a relatively fast option, has good accuracy and is portable, since it depends only on the synthesis tools and not on the HLS environment. However, any method to obtain the hardware metrics can be used instead of compilation (lines 8 and 17).

The approach iterative part (lines 10 to 31) starts with search queue being popped to get the design which will be explored together with its neighbours which are created using the function presented in Algorithm 10 (lines 11 and 13). Algorithm 10 returns all neighbours that were not explored or discarded by the compilation function presented in Algorithm 11.

In case neighbours are found (lines 15 to 20), they are compiled using the Algorithm 11 (line 16), which iteratively compiles the neighbourhood designs until a dominant design or one with the same hardware metrics is found. As such, Algorithm 11 compiles only a neighbours subset. Then, dominated designs are excluded from the search using the 8 conditions presented in Section 5.1.1 (lines 17 to 20). The search queue is updated by removing dominated points it may contain (line 21). Removing dominated points from the search queue reduces the number of evaluated designs forcing the algorithm only to visit the most promising points.

If no neighbours were found, which can happen in local-optimal plateaus, we apply the PMK 10% reduction rule iteratively until we achieve a configuration different from n_1 that has not been compiled or discarded is found (lines 23 to 30).

Algorithm 10 presents the function to get neighbours used by Algorithm 9, where a new configuration is created by reducing 1 from each resource constraint in the input configuration independently (lines 2 to 6), regarding to not reduce the constraints to less than 1 (lines 4).

Algorithm 11 presents the function to compile the neighbours from a given design, which iteratively compiles one design at a time (lines 2 to 10). In case a neighbour is found to dominate or to result in the same hardware metrics as the current design, the compilation stops, and the

Algorithm 9: Proposed Path-based DSE algorithm 9.

input : Data-Flow Graph (DFG)
output : Resources constraints and evaluated points metrics

1 $n_{max} \leftarrow$ maximum resources;
2 $n_1 \leftarrow$ ones vector;
3 $diff10 \leftarrow \lceil 0.1 * (n_{max} - n_1) \rceil$;
4 $currConstr \leftarrow n_{max}$;
5 $compiled \leftarrow [n_{max}]$;
6 $discarded \leftarrow \emptyset$;
7 $searchQueue \leftarrow n_{max}$;
8 compile n_{max};
9 $currMetrics \leftarrow n_{max}$ metrics;
10 **do**
11 $curr \leftarrow searchQueue.pop()$;
12 $currMetrics \leftarrow curr$ metrics;
13 $neighbors \leftarrow getNeighbors(curr)$;
14 /* do not add points which are dominated by $curr$ */
15 **if** $neighbors \neq \emptyset$ **then**
16 $nMetrics \leftarrow compileNeighbors(curr)$;
17 calculate da and dt for each neighbour;
18 **for** $n \in neighbors$ **do**
19 **if** $(da > 0 \,||\, dt > 0) \,||\, (da == 0 \,\&\&\, dt == 0)$ **then**
20 $searchQueue.insert(n)$;
21 $searchQueue \leftarrow pareto(searchQueue)$;
22 /* apply the 10% rules */
23 **if** $searchQueue == \emptyset \,\&\&\, n_1 \notin compiled$ **then**
24 $newConfig \leftarrow curr - diff10$;
25 change to 1 where $newConfig \leq 0$;
26 **while** $newConfig \in compiled \,||\, newConfig \in discarded \,\&\&\, newConfig \neq n_1$ **do**
27 $newConfig \leftarrow curr - diff10$;
28 change to 1 where $newConfig \leq 0$;
29 $searchQueue.insert(newConfig)$;
30 **while** $searchQueue \neq \emptyset$;

un-compiled designs are discarded (lines 7 to 10).

The proposed PBDSE approach (Algorithm 9) handles the flaws observed in the gradient-pruning based approaches and PMK. The flaw presented in LBDSE can be handled by a simple extension of Algorithm 8. The idea behind this extension is to use the PBDSE results instead of the U-shaped distribution in Algorithm 8. As such, with an initialization closer to the Pareto, we can avoid the local plateaus problems and speed-up the convergence.

Algorithm 12 presents the proposed expansion in the input line list and lines 1 to 4 which should substitute line 1 in Algorithm 8. This combination will be referred to as Path Seed

Algorithm 10: *getNeighbors*() function used bt Algorithm 9.

```
   input  : Design configuration curr.
   output : Neighbors configurations rSet.
1  rSet ← ∅;
2  for a ∈ curr do
3      newConfig ← curr;
4      newConfig(a) ← max(newConfig(a), 1);
5      if newconfig ∉ compiled && newconfig ∉ discarded then
6          rSet.insert(newConfig);
```

Algorithm 11: *compileNeighbors*() function used bt Algorithm 9.

```
   input  : Current design curr and its list of neighbours neighbors.
   output : Metrics for the compiled designs rMetrics.
1  cp ← ∅;
2  for n ∈ neighbors do
3      compile n;
4      calculate da and dt;
5      add n to cp;
6      if da ≥ 0 && dt ≥ 0 then
7          break;
8  /* compiled and discarded queues in Algorithm 9.                    */
9  append cp to compiled;
10 append neighbors/cp to discarded;
```

Lattice-Based DSE (P+LBDSE).

Algorithm 12: Lattice-based DSE algorithm accepting seed configurations.

```
   input  : List of directives and possible values, seeds
1  if seeds ≠ ∅ then
2      configs ← seeds
3  else
4      configs ← m U-distribution random configurations;
```

5.2.2. Resource Constraints Exploration Results

To evaluate how close from the estimated Pareto curve are from the real Pareto curve, we calculate the ADRS (Equation 4.6) between the exhaustive search Pareto-optimal points (reference set) and the PBDSE, LBDSE, and P+LBDSE (approximate sets) Pareto-optimal points; Note that a fair comparison with the literature is not trivial since other approaches consider more directives than just the resource constraints and also the whole code is targeted by the DSE, while we focus on loops only. Thus, we do not present results for PMK and gradient-pruning based approaches.

Table 6 presents the number of synthesized configurations for PBDSE, LBDSE and P+LBDSE, without loop pipelining ("No-pipe" for short) and using NIS and ILPS to create loop pipelines. Empirical tests have shown that $(m, \sigma) = (0.1, 0.25)$ are adequate to explore the configuration space formed by the resources constraints in our environment. The experiments presented in Ferretti, Ansaloni and Pozzi 2018, Ferretti, Ansaloni and Pozzi 2018 use $(m, \sigma) \geq (0.2, 0.5)$ to achieve an $ADRS < 1\%$. The necessity of a larger σ is caused by a more complex and unpredictable design space created by different optimizations, while our tests consider only the resource constraints.

Table 6 – Number of compiled configurations and ADRS obtained by PBDSE (P), LBDSE (L), and P+LBDSE (50 repetitions average).

		No-pipe			NIS			ILPS		
		P	L	P+L	P	L	P+L	P	L	P+L
# configs	**mt**	12	6.00	12	11	6.00	11	11	6.00	11
	dv	30	31.92	32	21	28.68	23	27	33.30	32
	fat	36	137.88	114	35	105.16	79	40	101.16	111
	cp	35	842.30	79	86	2011.3	473	74	1180.30	352
	ac	35	46	52	28	32.60	49	30	34.22	55
ADRS	**mt**	0	0.99	0	0	0.65	0	0	0.63	0
	dv	0.095	0.011	0.095	0.86	0.82	0.86	0.41	0.38	0.41
	fat	0.24	0.023	0.22	0.22	0.46	0.22	0.20	0.14	0.18
	cp	0.16	0.0036	0.15	0.38	0	0.033	0.30	0.031	0.11
	ac	0	0	0	0	0.38	0	0	0	0

Source: Research data.

First, we can not notice any substantial difference between using NIS and ILPS in Table 6, allowing to save computation time when compared with other module schedulers.

Table 7 shows the design space size and computation time for the exhaustive search realized to obtain the Pareto-optimal points. Note that the hours-scale **DSE** times are due to the usage elaboration and simulation to obtain the hardware metrics, and the time differences between ILPS and NIS should be more pronounced were faster estimation methods used. It is important to notice that loops without pipeline are scheduled as sequential code, by solving a constant number of SDC problems (Canis *et al.* 2013), resulting in computation times comparable to NIS, which also solves a constant number of SDC problems.

As such, the NIS small ADRS and large time savings corroborate with its adequacy for exploring design spaces in the early phases of a project.

Table 6, also shows that the ADRS is always larger when loop pipelining is used, what is caused since the pipelined design space has a clustering effect and creates larger overheads than the non-pipelined design space (will be explored in details in Section 5.3).

Table 6 shows that, for all benchmarks apart from **mt**, the PBDSE evaluates less points

Table 7 – Design space size and full exhaustive DSE time without pipeline and with NIS and ILPS to create pipelines.

		mt	**dv**	**fat**	**cp**	**ac**
Design Space Size		30	128	168	3920	48
time (hours)	No-pipe	1.5	6.4	6.4	84	2.4
	NIS	1.6	6.5	6.5	98	2.5
	ILPS	1.9	8.3	8.4	251	10,1

Source: Research data.

than LBDSE. However, PBDSE reaches the $ADRS < 1\%$ condition for which LBDSE is tuned to. P+LBDSE number of evaluated designs include the designs evaluated by PBDSE, and when compared to LBDSE show little improvement for all benchmarks apart from **cp**, indicating that the benefits of seeding the LBDSE are more significant for larger and more complex designs.

Furthermore, P+LBDSE can not degrade the PBDSE ADRS, making the exploration to be another option to balance the DSE speed-up and accuracy.

5.3. Loop Pipelining Exploration

Pipelining is one of the most important optimizations since it improves the throughput in exchange for increasing the hardware resources usage. This section presents studies on relations between the design space formed by the FU without and with loop pipelining, which allows to perform the DSE in one space and use its information to quickly explore the second one, reducing the number of evaluated designs. As such, we first analyse the loop pipelining effects on the design space, which allow us to propose a practical measure of similarity between the design spaces without and with pipelining. Finally, we propose a new "white-box" approach to explore both design spaces efficiently.

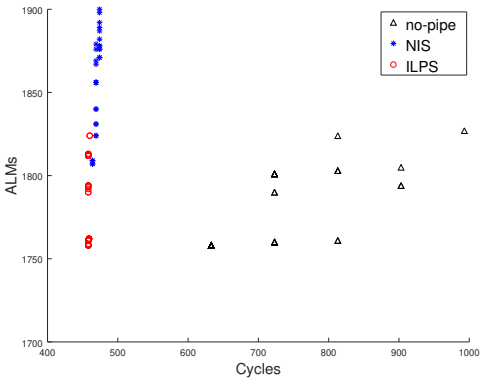
5.3.1. Loop Pipelining Effects on the Design Space

The goal of this section is to explicitly analyse the loop pipelining effects in the design space formed by the number of FU. To do so, Figure 18 presents the complete FU design space for the benchmarks used in Canis, Brown and Anderson 2014 without loop pipelining and when NIS and ILPS are used to create the pipelines.

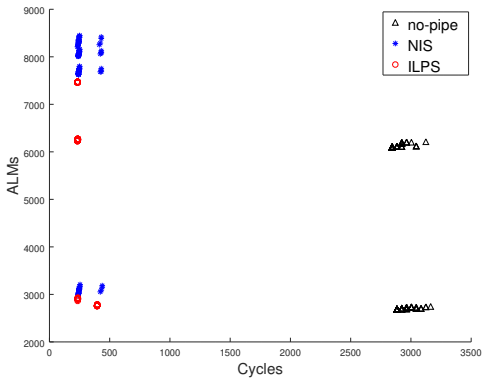
As expected, Figure 18 shows that all points in the design space have their number of clock cycles reduced when pipelining is applied, while the hardware resources usage increases. We also can observe that NIS points use more hardware resources than ILPS, which is expected according to Section 4.7.4. Furthermore, we can observe that the pipelined design spaces are clustered in the `cycles`-axis. This clustering happens since the II is the major contribution for pipelined designs cycles, which varies less than the loop latency.

Figure 18 – Design spaces without and with NIS and ILPS loop pipelining.

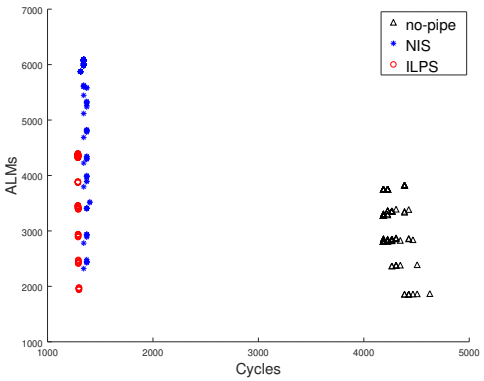
(a) **mt**



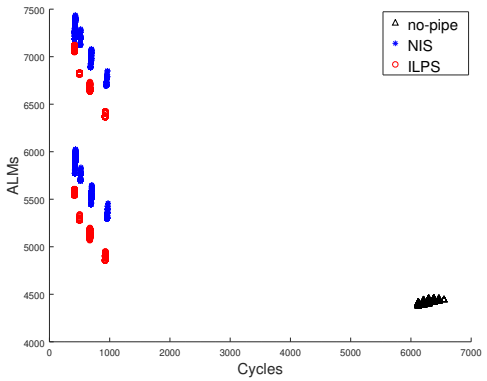
(b) **dv**



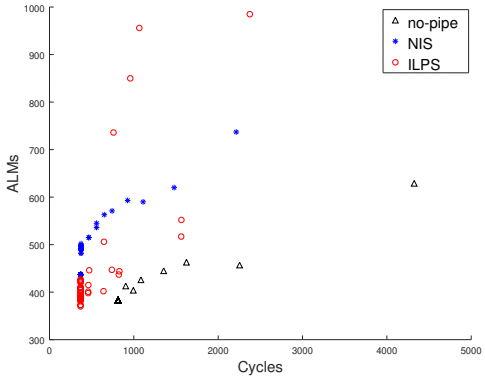
(c) **fat**



(d) **cp**



(e) **ac**



Source: Research data.

Figure 18 shows a clear separation between the design spaces without and with pipelining, which can be explored to improve the DSE performance. At the same time, this separation is an unpredictability source for "black-box"-based methods.

As mentioned before, "Treating different directives in the same manner, creates false neighbourhood relations in the design space.", which is the problem here.

As example, consider a configuration space for a design with only one type of resources

and two instructions $C_{init} = \{c_1 = [1], c_2 = [2]\}$. The literature DSE methods add a Boolean variable to represent when loop pipelining is `on` or `off` (let it be represented by 1 and 0 respectively), resulting in the new configuration space: $C_{pipe} = \{c_1 = [1,0], c_2 = [2,0], c_3 = [1,1], c_4 = [2,1]\}$.

In this case, points c_1 and c_3 are neighbours (same for c_3 and c_4). However, as demonstrated by Figure 18, points c_1 and c_3 are probably not close from each other in the combined design space C_{pipe}. This discrepancy is an uncertainty source while relating the configuration and design spaces, which prejudices typical DSE algorithms.

To avoid this situation, the straightforward idea is to explore both design spaces separately and merge the resulting Pareto estimations for each space to form a final estimation. As such, it solves the unpredictability source and achieves more accurate results. However, in the next section, we will explore relationships between the design spaces without and with pipelining. The unveiling of such relations can allow avoiding exploring both design spaces thoroughly as in the straightforward approach, achieving further speed-ups.

5.3.2. Practical Measure to Compare Design Spaces

This section proposes a practical measure that can be used to highlight relationships between design spaces without and with pipelining. The indication of similarities between the design spaces can be used to speed-up the DSE with the approach proposed in Section 5.3.3.

Let O be the set of DFG instructions with size $n = |O|$; let $l_{i=1,l\ldots,k}$ be the set composed by the instructions type; and let $a_k \geq 1$ be the number of instructions of type k. A configuration is represented by the vector c_i, which element represent the resources constraints for resource type k. As such, the total number of possible configurations is $p = \prod_{i=1}^{k} a_i$. The set of all configurations is defined as:

$$C = \left\{ c_i = \left[\begin{array}{c} c_i^1 \\ \vdots \\ c_i^k \end{array} \right] \middle| \, 1 \leq c_i^j \leq a_k \; \forall \; i = \{1,\ldots,p\}, \; j = \{1,\ldots,k\} \right\} \tag{5.4}$$

For each compiled design using configuration c_i, we obtain a set of metrics, represented by an m-array, such as $m \geq 2$, since we measure the number of clock cycles for the loop completion and at least one type of hardware resources usage. Thus, we write the set of metrics for the compiled designs without pipelining as:

$$B = \left\{ b_i = \left[\begin{array}{c} b_i^1 \\ \vdots \\ b_i^m \end{array} \right] \middle| \, 1 \leq b_i^j \; \forall \; i = \{1,\ldots,p\}, \; j = \{1,\ldots,m\} \right\} \tag{5.5}$$

Similarly, the set of metrics for the compiled designs with pipelining is written as:

$$A = \left\{ a_i = \left[\begin{array}{c} a_i^1 \\ \vdots \\ a_i^m \end{array} \right] \middle| 1 \leq a_i^j \; \forall \; i = \{1,\ldots,p\}, \; j = \{1,\ldots,m\} \right\} \tag{5.6}$$

Let also $f, g : C \rightarrow B, A$ be two surjective functions such as $b_i = f(c_i)$ and $a_i = g(c_i)$, respectively. As this point, we want to relate the spaces B and A to speculate if there is a relation between Pareto points in B and A. The motivation behind this is, the closer the B Pareto points are from the ones in A, less exploration is necessary to find the A Pareto points (and vice-versa).

We propose a measure to estimate how close A and B are from each other. As such, when applied over a set of representative benchmarks, the measure gives an idea about how the design spaces created by a compiler behave. The measure can be used with any number of points evaluated with and without loop pipelining and represents the value for a given benchmark, which opens the possibility to be used dynamically. However, this is not explored in this thesis.

The measure is constructed based on the relative position between pairs of points in both spaces. Formally, we define that two configurations (c_{i1}, c_{i2}) maintain their relative positions in spaces B and A if the relations $b_{i1}^j \; (< \mid > \mid =) \; b_{i2}^j \Leftrightarrow a_{i1}^j \; (< \mid > \mid =) \; a_{i2}^j \; \forall \; j = \{1,\ldots,m\}$.

Consider the $2-D$ example presented in Figure 19, where Figure 19a presents two points in a reference space, Figure 19b presents two correspondent points in another space that maintain the same relative position as in Figure 19a, while Figures 19c, 19d, and 19e present points that do not maintain the same relative position as in Figure 19a.

Figure 19 – Relative position in two different sets example. B in orange and A in green.

(a) Reference points. (b) Same relative positions. (c) Different relative positions. (d) Different relative positions. (e) Different relative positions.

Source: Elaborated by the author.

We define the Set Relative Position Distance (SRPD), which is an average based measure that indicates how many points in two sets maintain their relative positions, as Equation 5.7. The term $(a_i^k - a_j^k)(b_i^k - b_j^k) \geq 0$ is a logical comparison, returning 1 when it is true and 0 otherwise.

$$SRPD = \frac{2}{p(p-1)} \sum_{i=1}^{p} \sum_{j=i+1}^{p} \prod_{k=1}^{m} \left[(a_i^k - a_j^k)(b_i^k - b_j^k) \geq 0 \right] \tag{5.7}$$

Equation 5.7 measures the average number of pairs-of-points that maintain their relative positions in two spaces A and B. For each distinct pair of points i and j, 1 is added to the measure if their relative positions hold in all dimensions, represented by the $\prod$ term. Then, the number of pairs that hold their relative positions is summed up before being averaged by the term $\frac{2}{p(p-1)}$. Note that the logical comparison $\geq$ means that, if the pair of points coincide in both A or B, we consider that the points maintain their relative positions.

Equation 5.7 converges to 1 or 0 if all pairs-of-points maintain or not their relative positions, respectively. This is a useful tool that can be used to empirically study and compare the design spaces with and without pipelining (varying the resources constraints) for any compiler or set of codes. By defining and elaborating the design space for a set of benchmarks with and without loop pipelining, SRPD can be calculated. If $SRPD \approx 1$ for all benchmarks, then one can conclude that the design spaces are similar and only one needs to be effectively explored.

Table 8 presents the SRPD evaluation over the loop pipelining benchmarks used in Section 4, and the SRPD considering only the *cycles* and *ALMs* SRPD components independently.

Table 8 – SRPD for design spaces without and with NIS and ILPS loop pipelining.

SRPD	Scheduler	**ac**	**mt**	**dv**	**fat**	**cp**
total	ILPS	0.878	0.729	0.488	0.547	0.232
	NIS	0.922	0.837	0.502	0.689	0.232
cycles	ILPS	0.937	0.978	0.923	0.937	0.897
	NIS	0.937	0.978	0.937	0.963	0.799
ALMs	ILPS	0.923	0.849	0.674	0.684	0.448
	NIS	0.924	0.849	0.536	0.723	0.497

Source: Research data.

Table 8 shows that SRPD is small for the compiler used (LUP), however, the independent hardware metrics shows that the SRPD cycles component is close to 1, and the small SRPD is caused by the small SRPD ALM component. Each SRPD metric component shows the designs likelihood to keep their relative position for that metric. As such, we can conclude that a faster design in a space without pipeline will probably result in a faster design when pipelined.

We expect that the low SRPD ALM component value is due to designs fluctuating in-between the small local clusters in Figure 18; however, this subject needs further evaluation.

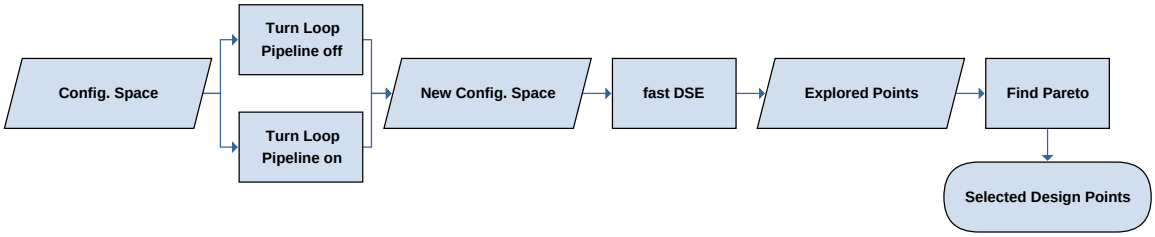
Furthermore, Table 8 shows no significant difference regarding the loop pipelining method used, what corroborates with the NIS usage during the project DSE phase.

5.3.3. Proposed Approach for Exploring Loop Pipelines

In this section presents a new method to explore the design space formed by the number of FU and loop pipelining based on the design spaces separation and SRPD results.

The literature DSE approaches combine the configuration space considering the loop pipelining turned `on` and `off` with the number of FU, resulting in an effective twice as large configuration space, on which the DSE is performed. This flow is presented in Figure 20. We will use the terminology $B \bigcup A$ to refer to the combined design space approach used in the literature.

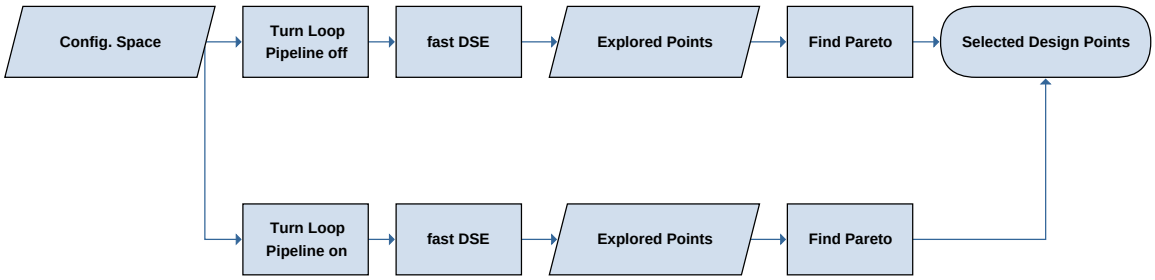
Figure 20 – Typical DSE flow considering resources constraints and loop pipelining.



Source: Elaborated by the author.

As noted in Section 5.3.1 exploring B and A separately and merging the results is a straightforward manner to increase the exploration precision. This flow is presented in Figure 21. We will use the terminology $B + A$ to refer to the proposed separate design space exploration.

Figure 21 – Proposed DSE pipeline independent flow considering resources constraints.



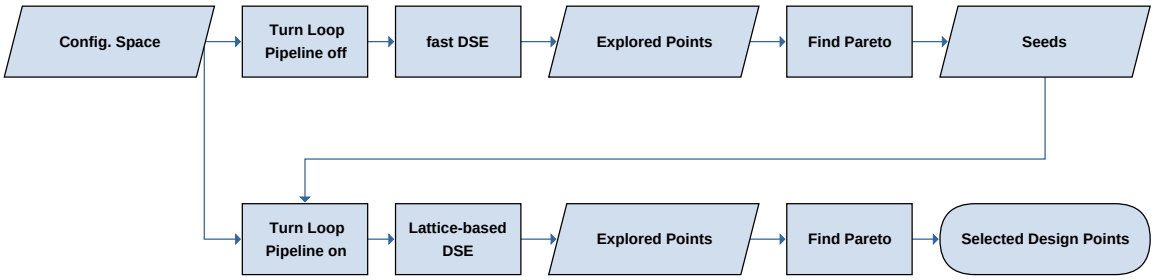
Source: Elaborated by the author.

Next, we present another approach aiming to achieve further DSE improvements by exploring the SRPD observations presented in Section 5.3.2.

The main idea of this second proposed approach is to perform the DSE in space B and use its results as seeds to explore the space A with P+LBDSE. The reasoning for this is that since the points maintain their relative positions in the `cycles` axes, the Pareto points from B will contain a sample of each II cluster observed in Figure 18. Furthermore, the SRPD ALM component indicates that more than half of the pairs-of-points maintain their relative positions in this direction, indicating a more than 50% chance of B Pareto points to be also A Pareto points. As such, we expect to diminish A exploration using the fact that B Pareto points should be close to A Pareto points.

We choose to use B results to seed A since the DSE in B results in a generally smaller ADRS, as presented in Table 6. Empirical results also have shown that the "B seeds A" achieves better results than vice-versa. The proposed seeding-based flow is presented in Figure 22. We will use the terminology $B \rightarrow A$ to refer to the proposed seeding-based DSE and Figure 22 flow usage.

Figure 22 – Proposed DSE seeding-based flow considering resources constraints and loop pipelining.



Source: Elaborated by the author.

The "`fast DSE`" box in Figures 20, 21, and 22 represents any method which aims to perform an accelerated DSE, e.g., PBDSE, LBDSE, or P+LBDSE.

5.3.4. Loop Pipelining Exploration Results

In this section, we will evaluate the three DSE flows considering the number of FU and loop pipelining presented in Section 5.3.3.

Tables 9 and 10 present the number of compiled designs and ADRS for the $B \rightarrow A$, $B+A$, and $B \bigcup A$ flows. Results are obtained using PBDSE, LBDSE and P+LBDSE as DSE methods. Tables 9 and 10 use NIS and ILPS to create the loop pipelines, respectively.

Tables 9 and 10 shows that $B \bigcup A$ achieves $ADRS > 1\%$ for the $(m, \sigma) = (0.05, 0.25)$ configuration, which was adequate to achieve $ADRS < 1\%$ for the designs spaces independently (Section 5.2.2). Increasing the configuration to $(m, \sigma) = (0.2, 0.5)$ (recommended by Ferretti, Ansaloni and Pozzi 2018) increases the number of explored designs achieves $ADSR < 1$. This demonstrate that combining the two designs spaces generates uncertainties, degrading the traditional "black-box"-based approaches performance, making necessary to explore more points to achieve comparable ADRS. As such, we have demonstrated that the "black-box" DSE flaws analyses (Section 5.3.1) reflect on the actual data.

Comparing $B+A$ and $B \bigcup A$, we cannot observe a significant variation in the number of evaluated designs. However, we can observe an ADRS reduction in all cases, as predicted in Sections 5.3.1 and 5.3.3.

Comparing $B \rightarrow A$ and $B+A$, we can observe that the seeding approach further reduces

Table 9 – Number of compiled configurations and ADRS using the $B \rightarrow A$, $B+A$, and $B \bigcup A$ flows, when PBDSE (P), LBDSE (L) and P+LBDSE (P+L) are used as `fast DSE` and NIS to create the loop pipelines.

NIS		$B \rightarrow A$			$B+A$			$B \bigcup A$			$B \bigcup A$	
	(m, σ)	$(0.05, 0.25)$			$(0.05, 0.25)$			$(0.05, 0.25)$			$(0.2, 0.5)$	
	DSE	P	L	$P+L$	P	L	$P+L$	P	L	$P+L$	L	$P+L$
# designs	**mt**	12	2	12	23	4	23	11	3	11	20.04	19
	dv	21	12.18	24	39	18.96	44	18	19.84	24	154.28	212
	fat	117	146.76	171	70	179.38	192	36	132.86	157	307.22	360
	cp	313	429.64	357	348	1153.54	1377	314	1111.68	1369	5102.16	4761
	ac	48	52	58	63	71.98	101	35	68.38	81	87.22	91
ADRS (%)	**mt**	0.90	2.37	0.90	0.00	1.90	0.00	0.00	3.43	0.00	0.26	0.00
	dv	2.31	4.76	2.31	0.70	3.16	0.70	2.64	3.24	2.64	0.33	0.01
	fat	1.40	0.35	1.40	1.32	0.84	1.32	2.44	3.78	1.81	0.14	0.00
	cp	0.02	0.02	0.02	0.06	0.00	0.02	0.96	0.01	0.02	0.00	0.00
	ac	0.00	0.00	0.00	0.00	2.52	0.00	22.22	2.56	0.00	0.25	0.00

Source: Research data.

Table 10 – Number of compiled configurations and ADRS using the $B \rightarrow A$, $B+A$, and $B \bigcup A$ flows, when PBDSE (P), LBDSE (L) and P+LBDSE (P+L) are used as `fast DSE` and ILPS to create the loop pipelines.

ILPS		$B \rightarrow A$			$B+A$			$B \bigcup A$			$B \bigcup A$	
	(m, σ)	$(0.05, 0.25)$			$(0.05, 0.25)$			$(0.05, 0.25)$			$(0.2, 0.5)$	
	DSE	P	L	$P+L$	P	L	$P+L$	P	L	$P+L$	L	$P+L$
# designs	**mt**	12	2	12	23	4	23	11	3	11	20.14	19
	dv	21	12.98	24	48	22.24	58	31	21.96	39	209.26	184
	fat	185	412.28	229	157	731.8	490	125	737.5	497	276.68	321
	cp	131	176.42	193	71	193	226	33	119.22	135	4935.72	3343
	ac	43	48	54	61	78.98	107	35	59.9	87	73.04	97
ADRS (%)	**mt**	0.90	2.86	0.90	0.00	1.70	0.00	0.00	3.46	0.00	0.36	0.00
	dv	0.97	4.19	0.97	0.39	0.92	0.39	0.95	1.49	0.95	0.11	0.31
	fat	1.12	0.35	1.12	0.26	0.06	0.09	0.28	0.06	0.09	0.33	0.05
	cp	0.27	0.28	0.27	0.25	0.19	0.22	3.85	0.89	0.49	0.00	0.00
	ac	0.00	0.00	0.00	3.65	0.00	0.00	44.69	1.10	0.00	0.00	0.00

Source: Research data.

the number of evaluated designs at the cost of a larger ADRS for most cases, indicating the SRPD-based supposition correctness. In other words, the results indicate that most points hold their relative positions. Hence Pareto points in B are close to A Pareto points.

The **ac** benchmark, using the PBDSE, is a special case. We can see that $B \bigcup A$ results in very large ADRS, which is reduced by $B + A$ (as expected), but not enough to achieve $ADRS < 1\%$, which is achieved by $B \rightarrow A$. Further analysis shows that the $ADRS > 1$ is caused by Pareto points not found in the pipelined space, thus using the estimated B Pareto points as seeds effectively results in good seeds for A exploration, resulting in the $ADRS < 1$. This corroborates with the adequacy of using the estimated B Pareto points as a seed to explore A.

Tables 9 and 10 are not easy to analyse since there are many trade-offs between the number of evaluated designs and ADRS.

Table 11 presents the average speed-up and *ADRS* values (geomean), summarizing the results presented by Tables 9 and 10, which indicates that $B \rightarrow A$, $B + A$, and $B \bigcup A$ have decreasing efficiency exploring the design spaces, and that increasing (m, σ) degrades further the $B \bigcup A$ exploration efficiency.

Table 11 – Speed-up and *ADRS* average values for the results presented in Tables 9 and 10.

		$B \rightarrow A$			$B + A$			$B \bigcup A$			$B \bigcup A$	
	(m, σ)	$(0.05, 0.25)$			$(0.05, 0.25)$			$(0.05, 0.25)$			$(0.2, 0.5)$	
	DSE	P	L	$P+L$	P	L	$P+L$	P	L	$P+L$	L	$P+L$
NIS	# designs	6.15	8.66	5.20	4.90	5.10	2.70	8.69	5.78	3.85	1.56	1.44
	ADRS (%)	0.93	1.52	0.93	0.42	1.71	0.41	6.06	2.60	0.90	0.20	0.00
ILPS	# designs	6.82	8.44	5.63	5.53	5.23	3.01	9.54	6.45	4.35	1.56	1.61
	ADRS (%)	0.65	1.55	0.65	0.92	0.58	0.14	12.08	1.41	0.31	0.16	0.07

Source: Research data.

Furthermore, Table 11 shows that using ILPS improves the DSE efficiency over NIS. However, this efficiency is measured as the number of evaluated designs and does not consider the time taken by each scheduler, which can make the ILPS usage impracticable, as presented in Section 5.2.2. For comparison, using the **cp** benchmark, 200 designs evaluated with a 10 minutes budget ILPS take $1,39$ days for the loop pipelining, while 400 designs evaluates with NIS take 44 seconds for the loop pipelining. Thus, we conclude that it is more time-efficient to use NIS and explore more designs to improve the ADRS than to use ILPS, corroborating with the NIS usage for the DSE and ILPS usage for the final implementation stages.

5.4. Loop Unrolling Design Space Completeness

Loop unrolling can expose data reuse and parallelism, enabling to improve the hardware implementations further. Furthermore, in some cases, it is interesting to fully unroll small nested loops in order to enable the creation of large outer-loops pipelines (Zuo *et al.* 2015).

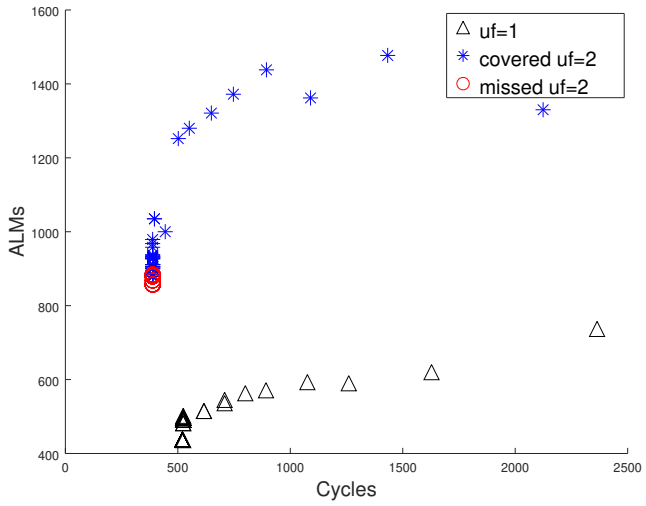
Since the literature DSE methods handle the optimizations as a “black-box”, they ignore the effects that directives have in the designs space, leading to incomplete results for loop unrolling. These approaches combine the unroll factor in the design space, forming a new, larger design space. As e.g. consider a configuration space for a design with only one type of resources constraints and two instructions $C_{init} = \{c_1 = [1], c_2 = [2]\}$ and maximum unroll factor $uf_{max} = 2$. Combining loop unrolling will extend the design space as $C_{unr}^{black-box} = \{c_1 = [1,1], c_2 = [2,1], c_3 = [1,2], c_4 = [2,2]\}$.

However, when loop unrolling is applied, the number of instructions in the code are multiplied according to the unrolling factor $uf \in 1, \ldots, tc$, becoming $uf \times a_k$. Thus the correct design space is $C_{unr}^{white-box} = \{c_1 = [1,1], c_2 = [2,1], c_3 = [1,2], c_4 = [2,2], c_5 = [3,2], c_6 = [4,2]\}$.

Thus, combining the resources constraints between 1 and a_k and unrolling factors uf do not cover the whole design space. Hence the “black-box” approaches analyse an incomplete design space, which might not cover the actual optimal solution.

Figure 23 presents an example using benchmark **ac** and maximum unroll factor $uf_{max} = 2$, highlighting the design space points with $uf = 1$, the points with $uf = 2$ which are covered by the literature approaches (“covered $uf = 2$”), and the points that are missed by such approaches (“missed $uf = 2$”).

Figure 23 – Benchmark **ac** design space using unroll factor $uf = 2$.



Source: Research data.

Figure 23 shows that the “black-box” approaches search a non-complete design space, what can make them not to find the actual best solutions.

PMK (Schafer 2016) is the only approach that explores loop unrolling and the resources constraints with different approaches, and state:

"*Step 2 [Probability Function Computation Prob(M_i)]:* All designs generated by exploring the *min* and *max* number of FUs for each micro-architecture [...]" (Schafer 2016, p.5).

As such, we interpreted that PMK evaluates the whole design space correctly.

This demonstrates that the proposed "white-box" DSE also allows observing inconsistencies that would arise without analysing the configuration and design spaces relations.

5.5. Final Remarks on Local Design Space Exploration

All approaches in the literature agree in the existence of an unpredictable relationship between the configuration and design spaces, which motivates the elaborated "black-box"-based methods used.

In this chapter, we propose to negate such unpredictability and demonstrate how the "black-box" assumption is the source of such complex relations, which hurts the methods speed and accuracy.

As such, we proposed "white-box" approaches for the resources constraints, loop pipelining and unrolling, which demonstrates that the explicit knowledge on each directive can lead to faster and more accurate methods. Even though it might not be practical to cover all compilers directives, the proposed paradigm shift can be elaborated for the most common or important directives, while using a "black-box" approach to complete the exploration of all directives.

CHAPTER

6

CONCLUSIONS

The introduction of high-level synthesis has impulsed the FPGA usage for accelerators creation. However, due to the high and low-level languages discrepancies, the creation of efficient hardware accelerators still requires expertise to map the application into efficient designs. As such, high-level synthesis was conceived as a tool to improve the hardware designers productivity.

Nowadays, high-level synthesis developments are attracting the attention of non-hardware-experts, which creates the necessity of methods to implement efficient accelerators automatically. To do so, the designer has to choose not only between many directives but also to smartly select how to partition the application code.

Towards code partitioning, the literature shows a general local DSE avoidance trend, which is caused by the lack of fast and accurate kernel performance models. Hence, we conclude that the local design space exploration is a primordial key to enable a better code partition. As such, this thesis focuses its efforts in the local design space acceleration and improvement, which consists of choosing a set of directives and its values to obtain the best hardware metrics.

The local DSE literature shows that it is necessary to associate hardware metrics estimation with smart selection methods to achieve practical exploration times. Hardware estimation metrics can be classified into two main streams, the faster but more limited HLS estimation and the slower but more complex and precise HDL estimation. Even though such estimations are out of this thesis scope, our tests show that HLS estimations are too imprecise, making its use a systematic error source. Thus, we decided to use HDL estimations to avoid invalidating the methods and results in this thesis due to such errors.

A first design space evaluation shows its non-trackable size due to a large number of available directives and their combinatorial combination possibilities. As such, we decided focusing on loop directives, as loop pipelining and unrolling since they are not constrained to specific compilers or platforms and are indicated to be the most impactful and common optimizations to control the hardware micro-architecture by the literature. However, first tests

show that the loop pipeline creating is a time-consuming process, being the local DSE bottleneck, especially when considered alongside loop unrolling. In some cases, loop pipelining could take more time than the hardware elaboration and synthesis. As such, speeding-up the loop pipelining becomes an essential step towards speeding up the local DSE.

An in-deep loop pipelining methods (modulo schedulers) study identifies SDCS and ILPS as the two state-of-the art methods for creating loop pipelines. SDCS inefficiency comes from the $\mathcal{O}(n^2)$ SDC problems solved to find a solution for the problem, while ILPS inefficiency comes from $\mathcal{O}(e^{n^2})$ solving complexity, with n being the `loop size`.

Towards accelerating the loop pipelining, our contributions lay as follows:

- We proposed a new SDC optimization formulation which allows the explicit separation between the scheduling and allocation parts of the problem.
- We use a GA that uses the new formulation and its relaxation to evaluate individuals and is accelerated by a proposed insemination process. The GA is shown to reduce the asymptotic number of solved SDC problems $\mathcal{O}(n)$, while allowing to control the balance between exploration time and solution quality.
- We proposed a non-iterative approach that reduces further the number of solved SDC problems to exactly 2, removing loop pipelining as the local DSE bottleneck.

The proposed methods improve the number of SDC problems solved in exchange of pipelines with larger latency, which has shown to impact by approximately 1% the total number of cycles. However, the results also demonstrate that the pipeline latency impacts the final hardware resources usage. By performing an exhaustive resources constraints DSE, we were able to demonstrate that the proposed schedulers larger latency impact in the hardware metrics can be diminished depending on the resources constraints configuration. As such, we conclude that the extra hardware resources usage is a consequence of inappropriately selected resources constraints. Furthermore, results show a less than 1% ADRS impact when using NIS, demonstrating its adequacy to speed-up the DSE.

With the loop pipelining bottleneck problem solved, we focus on the local DSE acceleration per se. An in-depth analysis on the local DSE literature reveals a common sense about the existence on an unpredictable relationship between the configuration and design spaces, which motivates the usage of “black-box” approaches, as heuristics, meta-heuristics, and ML.

However, an in-depth analysis of the works shows that considering the configuration-design space relation as a “black-box” is one of the main unpredictability sources, or being inconsistent. As such, we proposed an exploration paradigm shift to a “white-box”-based approach, where we analyse each directive and its effects on the configuration and design spaces. The goal of such stand-alone analysis is to unveil characteristics, relations, and features that

allow improving the DSE speed and accuracy. To demonstrate the proposed paradigm shift capabilities, we developed "white-box" approaches for the resources constraints, loop pipelining and unrolling directives.

As such, our contributions towards accelerating the local DSE are summarized as follows:

- We proposed a "black-to-white box" local DSE paradigm shift, showing the inconsistencies and unpredictability sources in the "black box"-based approaches from the literature.
- For resources constraints exploration:
  - We proposed a "white-box" path-based DSE (PBDSE), which avoids the gradient-based methods inconsistencies and incorporates mechanics to escape local traps.
  - We proposed an **LBDSE** improvement (P+LBDSE), which allows the introduction of search seeds, improving the method convergence rate.
- For loop pipelining exploration:
  - We proposed a straightforward separation approach ($B + A$) based on the design spaces without and with pipelining separability observations, which improves the exploration accuracy.
  - We proposed a practical measure (SRPD), which can be used over benchmarks to indicate the existence of relations between the design spaces without and with loop pipelining.
  - We proposed a seeding-based approach, which uses the results of exploring the design space without pipelining as seeds to explore the design space with pipelining ($B \rightarrow A$), which takes advantage of the relations pointed out by the SRPD measure to reduce the number of evaluated designs, improving the exploration speed.
- For loop unrolling exploration:
  - We demonstrate how the "black-box" approaches generate incomplete design spaces, which can prevent the methods to find the actual design optimal.

The PBDSE results demonstrate that the design space composed just by the resources constraints does not suffer from unpredictable relations by itself. Further analysis demonstrates how the insertion of other directives make the design space more "unpredictable", as loop pipelining introduces false-neighbourhood relations, and loop unrolling creates incomplete design spaces. As such, we have strong evidence that the configuration and design spaces relationships are not unpredictable as claimed in the literature, and that the "fuzziness" is caused by the "black-box" approach that treats directives with different effects as equals.

Even though analysing each directive is not a practical and portable solution (directives might behave uniquely in each environment), we argue that the most common and impactful solutions can be analysed as "white-boxes", while the other ones can still be evaluated with traditional "black-box"-based approaches.

Given all results, we can conclude that the proposed "white-box" paradigm shift allows to speed-up de design space exploration, improve its accuracy, and solve its inconsistencies. As such, the proposed paradigm is a first step to handle the configuration and design spaces relations unpredictability, which has been holding back the local design space true potential.

We conclude this thesis highlighting that the proposed approaches pave the future DSE techniques to enable efficient and generic global design space explorations, which oaths to facilitate the FPGA designs and to boost its adoption among non-hardware experts.

Future Works

The following modulo schedulers improvements can increase the proposed methods results in quality and speed:

- An As-Late-As-Possible schedule can be used to reduce the delays in Algorithm 7 (line 30);
- The implementation of alternative instruction orderings may improve the produced pipelines quality for NIS;
- Modifying the SDC formulation to optimize for register pressure following the work presented by Sittel *et al.* 2018;
- NIS MRT can be used as a seed for GAS, improving its convergence rate and results quality.
- Hierarchically DFG sub-graphs selection allows creating small modulo scheduling problems, which solution can be used to compose the whole graph scheduler, as demonstrated by Oppermann *et al.* 2018.

The following local DSE acceleration improvements can be made to increase the proposed approaches results quality and speed:

- The proposed PBDSE does not incorporate any mechanic to trade-off the number of explored designs and accuracy. A back-tracking approach could handle allow this balance capability.
- Formal loop pipelining modelling might give more insights between the design spaces with and without pipeline relations, which can improve the speed-up and solutions quality.

- Further studies on loop unrolling can reveal conditions that create bottlenecks in unrolled loops, which prevents further improvements in speed and hardware resources usage. Leveraging such conditions can allow proposing methods to prune the loop unroll factor, improving the DSE speed.
- We also point out that the are other directives can be explored as “white-boxes”, each one contributing with the local design space exploration.

BIBLIOGRAPHY

Agarwal, Ng and Arvind 2010 AGARWAL, A.; NG, M. C.; ARVIND. A Comparative Evaluation of High-Level Hardware Synthesis Using Reed Solomon Decoder. **IEEE Embedded Systems Letters**, v. 2, n. 3, p. 72–76, Sept 2010. ISSN 1943-0663. Cited on page 44.

Aung, Lam and Srikanthan 2015 AUNG, Y. L.; LAM, S. K.; SRIKANTHAN, T. Rapid estimation of DSPs utilization for efficient high-level synthesis. In: **2015 IEEE International Conference on Digital Signal Processing (DSP)**. [S.l.: s.n.], 2015. p. 1261–1265. ISSN 1546-1874. Cited on page 41.

Azarkhish *et al.* 2015 AZARKHISH, E.; ROSSI, D.; LOI, I.; BENINI, L. High performance AXI-4.0 based interconnect for extensible smart memory cubes. In: EDA CONSORTIUM. **Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition**. [S.l.], 2015. p. 1317–1322. Cited on page 39.

Beyls and D'Hollander 2006 BEYLS, K.; D'HOLLANDER, E. H. Intermediately Executed Code is the Key to Find Refactorings That Improve Temporal Data Locality. In: **Proceedings of the 3rd Conference on Computing Frontiers**. New York, NY, USA: ACM, 2006. (CF '06), p. 373–382. ISBN 1-59593-302-6. Available: <http://doi.acm.org/10.1145/1128022.1128071>. Cited on page 53.

Bondhugula *et al.* 2008 BONDHUGULA, U.; HARTONO, A.; RAMANUJAM, J.; SADAYAPPAN, P. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In: **Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation**. New York, NY, USA: ACM, 2008. (PLDI '08), p. 101–113. ISBN 978-1-59593-860-2. Available: <http://doi.acm.org/10.1145/1375581.1375595>. Cited on page 35.

Buyukkurt, Guo and Najjar 2006 BUYUKKURT, B.; GUO, Z.; NAJJAR, W. A. **Impact of loop unrolling on area, throughput and clock frequency in ROCCC: C to VHDL compiler for FPGAs**. Berlin, Heidelberg, 2006. 401–412 p. ISBN 978-3-540-36863-2. Available: <http://dx.doi.org/10.1007/11802839_48>. Cited on page 52.

Camion 1965 CAMION, P. Characterization of totally unimodular matrices. **Proceedings of the American Mathematical Society**, JSTOR, v. 16, n. 5, p. 1068–1073, 1965. Cited on page 56.

Canis, Brown and Anderson 2014 CANIS, A.; BROWN, S. D.; ANDERSON, J. H. Modulo SDC scheduling with recurrence minimization in high-level synthesis. In: **2014 24th International Conference on Field Programmable Logic and Applications (FPL)**. [S.l.: s.n.], 2014. p. 1–8. ISSN 1946-147X. Cited on pages 19, 54, 55, 56, 57, 58, 62, 78, 84, and 103.

Canis *et al.* 2013 CANIS, A.; CHOI, J.; ALDHAM, M.; ZHANG, V.; KAMMOONA, A.; CZAJKOWSKI, T.; BROWN, S. D.; ANDERSON, J. H. LegUp: An Open-source High-level Synthesis Tool for FPGA-based Processor/Accelerator Systems. **ACM Trans. Embed. Comput. Syst.**, ACM, New York, NY, USA, v. 13, n. 2, p. 24:1–24:27, Sep. 2013. ISSN 1539-9087. Available: <http://doi.acm.org/10.1145/2514740>. Cited on pages 44 and 102.

Canis *et al.* 2013 CANIS, A.; CHOI, J.; FORT, B.; LIAN, R.; HUANG, Q.; CALAGAR, N.; GORT, M.; QIN, J. J.; ALDHAM, M.; CZAJKOWSKI, T.; BROWN, S.; ANDERSON, J. From Software to Accelerators with LegUp High-level Synthesis. In: **Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems**. Piscataway, NJ, USA: IEEE Press, 2013. (CASES '13), p. 18:1–18:9. ISBN 978-1-4799-1400-5. Available: <http://dl.acm.org/citation.cfm?id=2555729.2555747>. Cited on page 54.

Cardoso 2003 CARDOSO, J. M. P. Loop dissevering: a technique for temporally partitioning loops in dynamically reconfigurable computing platforms. In: **Parallel and Distributed Processing Symposium, 2003. Proceedings. International**. [S.l.: s.n.], 2003. p. 8 pp.–. ISSN 1530-2075. Cited on page 53.

Cardoso and Diniz 2004 CARDOSO, J. M. P.; DINIZ, P. C. **Computer Systems: Architectures, Modeling, and Simulation: Third and Fourth International Workshops, SAMOS 2004, Samos, Greece, July 21-23, 2004 and July 19-21, 2004. Proceedings**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. 224–233 p. ISBN 978-3-540-27776-7. Available: <http://dx.doi.org/10.1007/978-3-540-27776-7_24>. Cited on page 52.

Castro *et al.* 2015 CASTRO, P. D. O.; AKEL, C.; PETIT, E.; POPOV, M.; JALBY, W. CERE: LLVM-Based Codelet Extractor and REplayer for Piecewise Benchmarking and Optimization. **ACM Trans. Archit. Code Optim.**, ACM, New York, NY, USA, v. 12, n. 1, p. 6:1–6:24, Apr. 2015. ISSN 1544-3566. Available: <http://doi.acm.org/10.1145/2724717>. Cited on page 34.

Che, Sheaffer and Skadron 2011 CHE, S.; SHEAFFER, J. W.; SKADRON, K. Dymaxion: Optimizing Memory Access Patterns for Heterogeneous Systems. In: **Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis**. New York, NY, USA: ACM, 2011. (SC '11), p. 13:1–13:11. ISBN 978-1-4503-0771-0. Available: <http://doi.acm.org/10.1145/2063384.2063401>. Cited on page 43.

Chen *et al.* 2006 CHEN, G.; OZTURK, O.; KANDEMIR, M.; KARAKOY, M. Dynamic Scratch-pad Memory Management for Irregular Array Access Patterns. In: **Proceedings of the Conference on Design, Automation and Test in Europe: Proceedings**. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2006. (DATE '06), p. 931–936. ISBN 3-9810801-0-6. Available: <http://dl.acm.org/citation.cfm?id=1131481.1131741>. Cited on page 43.

Cilardo and Gallo 2015 CILARDO, A.; GALLO, L. Interplay of loop unrolling and multidimensional memory partitioning in HLS. In: **2015 Design, Automation Test in Europe Conference Exhibition (DATE)**. [S.l.: s.n.], 2015. p. 163–168. ISSN 1530-1591. Cited on page 43.

Codina, Llosa and González 2002 CODINA, J. M.; LLOSA, J.; GONZáLEZ, A. A Comparative Study of Modulo Scheduling Techniques. In: **Proceedings of the 16th International Conference on Supercomputing**. New York, NY, USA: ACM, 2002. (ICS '02), p. 97–106. ISBN 1-58113-483-5. Available: <http://doi.acm.org/10.1145/514191.514208>. Cited on pages 62 and 79.

Cong *et al.* 2018 CONG, J.; FANG, Z.; HAO, Y.; WEI, P.; YU, C. H.; ZHANG, C.; ZHOU, P. Best-Effort FPGA Programming: A Few Steps Can Go a Long Way. **arXiv preprint arXiv:1807.01340**, 2018. Cited on page 39.

Cong *et al.* 2017 CONG, J.; WEI, P.; YU, C. H.; ZHOU, P. Bandwidth Optimization Through On-Chip Memory Restructuring for HLS. In: **Proceedings of the 54th Annual Design Automation Conference 2017**. New York, NY, USA: ACM, 2017. (DAC '17), p. 43:1–43:6. ISBN 978-1-4503-4927-7. Available: <http://doi.acm.org/10.1145/3061639.3062208>. Cited on page 40.

Cong *et al.* 2018 CONG, J.; WEI, P.; YU, C. H.; ZHANG, P. Automated Accelerator Generation and Optimization with Composable, Parallel and Pipeline Architecture. In: **Proceedings of the 55th Annual Design Automation Conference**. New York, NY, USA: ACM, 2018. (DAC '18), p. 154:1–154:6. ISBN 978-1-4503-5700-5. Available: <http://doi.acm.org/10.1145/3195970.3195999>. Cited on page 39.

Cong, Zhang and Zou 2012 CONG, J.; ZHANG, P.; ZOU, Y. Optimizing Memory Hierarchy Allocation with Loop Transformations for High-level Synthesis. In: **Proceedings of the 49th Annual Design Automation Conference**. New York, NY, USA: ACM, 2012. (DAC '12), p. 1233–1238. ISBN 978-1-4503-1199-1. Available: <http://doi.acm.org/10.1145/2228360.2228586>. Cited on page 53.

Corre *et al.* 2012 CORRE, Y.; HOANG, V.; DIGUET, J.; HELLER, D.; LAGADEC, L. HLS-based fast design space exploration of ad hoc hardware accelerators: A key tool for MPSoC synthesis on FPGA. In: **Proceedings of the 2012 Conference on Design and Architectures for Signal and Image Processing**. [S.l.: s.n.], 2012. p. 1–8. Cited on page 91.

Czajkowski *et al.* 2012 CZAJKOWSKI, T. S.; AYDONAT, U.; DENISENKO, D.; FREEMAN, J.; KINSNER, M.; NETO, D.; WONG, J.; YIANNACOURAS, P.; SINGH, D. P. From opencl to high-performance hardware on FPGAS. In: **22nd International Conference on Field Programmable Logic and Applications (FPL)**. [S.l.: s.n.], 2012. p. 531–534. ISSN 1946-147X. Cited on page 42.

Dai *et al.* 2017 DAI, S.; LIU, G.; ZHAO, R.; ZHANG, Z. Enabling adaptive loop pipelining in high-level synthesis. In: **2017 51st Asilomar Conference on Signals, Systems, and Computers**. [S.l.: s.n.], 2017. p. 131–135. Cited on page 82.

Darte 2000 DARTE, A. On the complexity of loop fusion . **Parallel Computing**, v. 26, n. 9, p. 1175 – 1193, 2000. ISSN 0167-8191. Available: <http://www.sciencedirect.com/science/article/pii/S016781910000034X>. Cited on page 43.

Deng *et al.* 2011 DENG, L.; SOBTI, K.; ZHANG, Y.; CHAKRABARTI, C. Accurate Area, Time and Power Models for FPGA-Based Implementations. **Journal of Signal Processing Systems**, v. 63, n. 1, p. 39–50, 2011. ISSN 1939-8115. Available: <http://dx.doi.org/10.1007/s11265-009-0387-7>. Cited on page 42.

Devos *et al.* 2007 DEVOS, H.; BEYLS, K.; CHRISTIAENS, M.; CAMPENHOUT, J.; D'HOLLANDER, E. H.; STROOBANDT, D. **Transactions on High-Performance Embedded Architectures and Compilers I**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. 159–178 p. ISBN 978-3-540-71528-3. Available: <http://dx.doi.org/10.1007/978-3-540-71528-3_11>. Cited on pages 52 and 53.

Dwivedi *et al.* 2006 DWIVEDI, B. K.; KEJARIWAL, A.; BALAKRISHNAN, M.; KUMAR, A. Rapid Resource-Constrained Hardware Performance Estimation. In: **Seventeenth IEEE International Workshop on Rapid System Prototyping (RSP'06)**. [S.l.: s.n.], 2006. p. 40–46. ISSN 1074-6005. Cited on page 41.

Dyer and Frieze 1994 DYER, M.; FRIEZE, A. Random walks, totally unimodular matrices, and a randomised dual simplex algorithm. **Mathematical Programming**, Springer, v. 64, n. 1-3, p. 1–16, 1994. Cited on page 56.

Enzler *et al.* 2000 ENZLER, R.; JEGER, T.; COTTET, D.; TRÖSTER, G. High-Level Area and Performance Estimation of Hardware Building Blocks on FPGAs. In: HARTENSTEIN, R. W.; GRÜNBACHER, H. (Ed.). **Field-Programmable Logic and Applications: The Roadmap to Reconfigurable Computing**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000. p. 525–534. ISBN 978-3-540-44614-9. Cited on page 42.

Escobar, Chang and Valderrama 2016 ESCOBAR, F. A.; CHANG, X.; VALDERRAMA, C. Suitability analysis of FPGAs for heterogeneous platforms in HPC. **IEEE Transactions on Parallel and Distributed Systems**, IEEE, v. 27, n. 2, p. 600–612, 2016. Cited on page 29.

Fernando *et al.* 2015 FERNANDO, S.; WIJTVLIET, M.; NUGTEREN, C.; KUMAR, A.; CORPORAAL, H. (AS)2: Accelerator Synthesis Using Algorithmic Skeletons for Rapid Design Space Exploration. In: **Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition**. San Jose, CA, USA: EDA Consortium, 2015. (DATE '15), p. 305–308. ISBN 978-3-9815370-4-8. Available: <http://dl.acm.org/citation.cfm?id=2755753.2755821>. Cited on page 36.

Ferretti, Ansaloni and Pozzi 2018 FERRETTI, L.; ANSALONI, G.; POZZI, L. Cluster-Based Heuristic for High Level Synthesis Design Space Exploration. In: . IEEE, 2018. PP, p. 1–1. ISSN 2168-6750. Available: <https://ieeexplore.ieee.org/document/8259330/>. Cited on pages 39 and 102.

Ferretti, Ansaloni and Pozzi 2018 ____. Lattice-Traversing Design Space Exploration for High Level Synthesis. In: . IEEE, 2018. p. 210–217. ISBN 978-1-5386-8477-1. ISSN 2576-6996. Available: <https://ieeexplore.ieee.org/document/8615690/>. Cited on pages 39, 90, 96, 102, and 109.

Fowers *et al.* 2012 FOWERS, J.; BROWN, G.; COOKE, P.; STITT, G. A Performance and Energy Comparison of FPGAs, GPUs, and Multicores for Sliding-window Applications. In: **Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays**. New York, NY, USA: ACM, 2012. (FPGA '12), p. 47–56. ISBN 978-1-4503-1155-7. Available: <http://doi.acm.org/10.1145/2145694.2145704>. Cited on page 29.

Giefers *et al.* 2016 GIEFERS, H.; STAAR, P.; BEKAS, C.; HAGLEITNER, C. Analyzing the energy-efficiency of sparse matrix multiplication on heterogeneous systems: A comparative study of GPU, Xeon Phi and FPGA. In: IEEE. **2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)**. [S.l.], 2016. p. 46–56. Cited on page 29.

Gokhale and Stone 1998 GOKHALE, M. B.; STONE, J. M. NAPA C: compiling for a hybrid RISC/FPGA architecture. In: **FPGAs for Custom Computing Machines, 1998. Proceedings. IEEE Symposium on**. [S.l.: s.n.], 1998. p. 126–135. Cited on page 52.

Guo *et al.* 2005 GUO, Z.; BUYUKKURT, B.; NAJJAR, W.; VISSERS, K. Optimized Generation of Data-Path from C Codes for FPGAs. In: **Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1**. Washington, DC, USA: IEEE Computer Society, 2005. (DATE '05), p. 112–117. ISBN 0-7695-2288-2. Available: <http://dx.doi.org/10.1109/DATE.2005.234>. Cited on page 34.

Henkel and Ernst 1998 HENKEL, J.; ERNST, R. High-level estimation techniques for usage in hardware/software co-design. In: **Proceedings of 1998 Asia and South Pacific Design Automation Conference**. [S.l.: s.n.], 1998. p. 353–360. Cited on page 42.

Huang *et al.* 2013 HUANG, Q.; LIAN, R.; CANIS, A.; CHOI, J.; XI, R.; BROWN, S.; ANDERSON, J. The Effect of Compiler Optimizations on High-Level Synthesis for FPGAs. In: **2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines**. [S.l.: s.n.], 2013. p. 89–96. Cited on page 52.

Jang *et al.* 2011 JANG, B.; SCHAA, D.; MISTRY, P.; KAELI, D. Exploiting Memory Access Patterns to Improve Memory Performance in Data-Parallel Architectures. **IEEE Transactions on Parallel and Distributed Systems**, v. 22, n. 1, p. 105–118, Jan 2011. ISSN 1045-9219. Cited on page 43.

Jiang, Tang and Banerjee 2006 JIANG, T.; TANG, X.; BANERJEE, P. Macro-models for high-level area and power estimation on FPGAs. **International Journal of Simulation and Process Modelling**, Inderscience Publishers, v. 2, n. 1-2, p. 12–19, 2006. Cited on page 42.

Kenter, Vaz and Plessl 2014 KENTER, T.; VAZ, G.; PLESSL, C. Partitioning and Vectorizing Binary Applications for a Reconfigurable Vector Computer. In: GOEHRINGER, D.; SANTAMBROGIO, M.; CARDOSO, J.; BERTELS, K. (Ed.). **Reconfigurable Computing: Architectures, Tools, and Applications**. Springer International Publishing, 2014, (Lecture Notes in Computer Science, v. 8405). p. 144–155. ISBN 978-3-319-05959-4. Available: <http://dx.doi.org/10.1007/978-3-319-05960-0_13>. Cited on page 34.

Kulkarni *et al.* 2002 KULKARNI, D.; NAJJAR, W. A.; RINKER, R.; KURDAHI, F. J. Fast area estimation to support compiler optimizations in FPGA-based reconfigurable systems. In: **Field-Programmable Custom Computing Machines, 2002. Proceedings. 10th Annual IEEE Symposium on**. [S.l.: s.n.], 2002. p. 239–247. Cited on pages 42 and 44.

Kulkarni *et al.* 2006 ____. Compile-time Area Estimation for LUT-based FPGAs. In: . New York, NY, USA: ACM, 2006. v. 11, n. 1, p. 104–122. ISSN 1084-4309. Available: <http://doi.acm.org/10.1145/1124713.1124721>. Cited on pages 42 and 44.

Liu and Schafer 2016 LIU, D.; SCHAFER, B. C. Efficient and reliable high-level synthesis design space explorer for FPGAs. In: IEEE. IEEE, 2016. p. 1–8. ISBN 978-2-8399-1844-2. Available: <https://ieeexplore.ieee.org/document/7577370/>. Cited on page 38.

Liu and Carloni 2013 LIU, H.-Y.; CARLONI, L. P. On Learning-based Methods for Design-space Exploration with High-level Synthesis. In: **Proceedings of the 50th Annual Design Automation Conference**. New York, NY, USA: ACM, 2013. (DAC '13), p. 50:1–50:7. ISBN 978-1-4503-2071-9. Available: <http://doi.acm.org/10.1145/2463209.2488795>. Cited on page 35.

Liu, Bayliss and Constantinides 2015 LIU, J.; BAYLISS, S.; CONSTANTINIDES, G. A. Offline Synthesis of Online Dependence Testing: Parametric Loop Pipelining for HLS. In: **2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines**. [S.l.: s.n.], 2015. p. 159–162. Cited on pages 43, 44, and 78.

Llosa *et al.* 2001 LLOSA, J.; AYGUADE, E.; GONZALEZ, A.; VALERO, M.; ECKHARDT, J. Lifetime-sensitive modulo scheduling in a production environment. **IEEE Transactions on Computers**, v. 50, n. 3, p. 234–249, Mar 2001. ISSN 0018-9340. Cited on pages 56, 62, 79, and 82.

Llosa *et al.* 1998 LLOSA, J.; VALERO, M.; AGYUADE, E.; GONZALEZ, A. Modulo scheduling with reduced register pressure. **IEEE Transactions on Computers**, v. 47, n. 6, p. 625–638, June 1998. ISSN 0018-9340. Cited on page 81.

Marugán, González-Bayón and Espeso 2011 MARUGáN, P. G. de A.; GONZáLEZ-BAYóN, J.; ESPESO, P. S. Hardware performance estimation by dynamic scheduling. In: **FDL 2011 Proceedings**. [S.l.: s.n.], 2011. p. 1–6. ISSN 1636-9874. Cited on page 42.

Matai *et al.* 2016 MATAI, J.; LEE, D.; ALTHOFF, A.; KASTNER, R. Composable, Parameterizable Templates for High-level Synthesis. In: **Proceedings of the 2016 Conference on Design, Automation & Test in Europe**. San Jose, CA, USA: EDA Consortium, 2016. (DATE '16), p. 744–749. ISBN 978-3-9815370-6-2. Available: <http://dl.acm.org/citation.cfm?id=2971808.2971979>. Cited on page 34.

McFarland, Parker and Camposano 1990 MCFARLAND, M. C.; PARKER, A. C.; CAMPOSANO, R. The high-level synthesis of digital systems. **Proceedings of the IEEE**, IEEE, v. 78, n. 2, p. 301–318, 1990. Cited on page 53.

McKinley, Carr and Tseng 1996 MCKINLEY, K. S.; CARR, S.; TSENG, C.-W. Improving Data Locality with Loop Transformations. **ACM Trans. Program. Lang. Syst.**, ACM, New York, NY, USA, v. 18, n. 4, p. 424–453, Jul. 1996. ISSN 0164-0925. Available: <http://doi.acm.org/10.1145/233561.233564>. Cited on page 52.

MIT 2016 MIT. **Gaussian elimination with pivoting**. 2016. Available: <http://web.mit.edu/course/10/10.001/OldFiles/Web/Course_Notes/main.html>. Accessed: 2016-06-16. Cited on page 44.

Mitchell 1996 MITCHELL, M. An Introduction to Genetic Algorithms. The MIT Press, 1996. Cited on page 67.

Mittal and Vetter 2014 MITTAL, S.; VETTER, J. S. A Survey of Methods for Analyzing and Improving GPU Energy Efficiency. **ACM Comput. Surv.**, ACM, New York, NY, USA, v. 47, n. 2, p. 19:1–19:23, Aug. 2014. ISSN 0360-0300. Available: <http://doi.acm.org/10.1145/2636342>. Cited on page 29.

Murphy and Kogge 2007 MURPHY, R. C.; KOGGE, P. M. On the Memory Access Patterns of Supercomputer Applications: Benchmark Selection and Its Implications. **IEEE Transactions on Computers**, v. 56, n. 7, p. 937–945, July 2007. ISSN 0018-9340. Cited on page 43.

Nane *et al.* 2016 NANE, R.; SIMA, V.-M.; PILATO, C.; CHOI, J.; FORT, B.; CANIS, A.; CHEN, Y. T.; HSIAO, H.; BROWN, S.; FERRANDI, F. *et al.* A survey and evaluation of FPGA high-level synthesis tools. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, v. 35, n. 10, p. 1591–1604, 2016. Cited on page 29.

Nayak *et al.* 2002 NAYAK, A.; HALDAR, M.; CHOUDHARY, A.; BANERJEE, P. Accurate Area and Delay Estimators for FPGAs. In: **Proceedings of the Conference on Design, Automation and Test in Europe**. Washington, DC, USA: IEEE Computer Society, 2002. (DATE '02), p. 862–. ISBN 0-7695-1471-5. Available: <http://dl.acm.org/citation.cfm?id=882452.874410>. Cited on page 42.

Nugteren and Corporaal 2014 NUGTEREN, C.; CORPORAAL, H. Bones: An Automatic Skeleton-Based C-to-CUDA Compiler for GPUs. **ACM Trans. Archit. Code Optim.**, ACM,

New York, NY, USA, v. 11, n. 4, p. 35:1–35:25, dec 2014. ISSN 1544-3566. Available: <http://doi.acm.org/10.1145/2665079>. Cited on page 36.

Nurvitadhi *et al.* 2016 NURVITADHI, E.; SHEFFIELD, D.; SIM, J.; MISHRA, A.; VENKATESH, G.; MARR, D. Accelerating binarized neural networks: Comparison of FPGA, CPU, GPU, and ASIC. In: IEEE. **2016 International Conference on Field-Programmable Technology (FPT)**. [S.l.], 2016. p. 77–84. Cited on page 29.

Nurvitadhi *et al.* 2017 NURVITADHI, E.; VENKATESH, G.; SIM, J.; MARR, D.; HUANG, R.; HOCK, J. O. G.; LIEW, Y. T.; SRIVATSAN, K.; MOSS, D.; SUBHASCHANDRA, S.; BOUDOUKH, G. Can fpgas beat gpus in accelerating next-generation deep neural networks? In: **Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays**. New York, NY, USA: ACM, 2017. (FPGA '17), p. 5–14. ISBN 978-1-4503-4354-1. Available: <http://doi.acm.org/10.1145/3020078.3021740>. Cited on page 29.

Okina *et al.* 2016 OKINA, K.; SOEJIMA, R.; FUKUMOTO, K.; SHIBATA, Y.; OGURI, K. Power Performance Profiling of 3-D Stencil Computation on an FPGA Accelerator for Efficient Pipeline Optimization. **SIGARCH Comput. Archit. News**, ACM, New York, NY, USA, v. 43, n. 4, p. 9–14, Apr. 2016. ISSN 0163-5964. Available: <http://doi.acm.org.ez67.periodicos.capes.gov.br/10.1145/2927964.2927967>. Cited on page 42.

O'Neal and Brisk 2018 O'NEAL, K.; BRISK, P. Predictive Modeling for CPU, GPU, and FPGA Performance and Power Consumption: A Survey. In: IEEE. **2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)**. [S.l.], 2018. p. 763–768. Cited on page 42.

Oppermann and Koch 2016 OPPERMANN, J.; KOCH, A. Detecting kernels suitable for C-based high-level hardware synthesis. In: IEEE. **2016 Intl IEEE Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld)**. [S.l.], 2016. p. 1157–1164. Cited on page 34.

Oppermann *et al.* 2016 OPPERMANN, J.; KOCH, A.; REUTER-OPPERMANN, M.; SINNEN, O. ILP-based Modulo Scheduling for High-level Synthesis. In: **Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems**. New York, NY, USA: ACM, 2016. (CASES '16), p. 1:1–1:10. ISBN 978-1-4503-4482-1. Available: <http://doi.acm.org/10.1145/2968455.2968512>. Cited on pages 17, 56, 59, 60, and 61.

Oppermann *et al.* 2018 OPPERMANN, J.; REUTER-OPPERMANN, M.; SOMMER, L.; SINNEN, O.; KOCH, A. Dependence Graph Preprocessing for Faster Exact Modulo Scheduling in High-level Synthesis. 2018. Cited on page 118.

Optimization 2017 OPTIMIZATION, G. Inc.,"gurobi optimizer reference manual," 2017. **URL: http://www.gurobi.com**, 2017. Cited on page 78.

Pilato and Ferrandi 2012 PILATO, C.; FERRANDI, F. Bambu: A Free Framework for the High Level Synthesis of Complex Applications. **University Booth of DATE**, v. 29, p. 2011, 2012. Cited on page 50.

Pouchet *et al.* 2013 POUCHET, L.-N.; ZHANG, P.; SADAYAPPAN, P.; CONG, J. Polyhedral-based Data Reuse Optimization for Configurable Computing. In: **Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays**. New York,

NY, USA: ACM, 2013. (FPGA '13), p. 29–38. ISBN 978-1-4503-1887-7. Available: <http://doi.acm.org/10.1145/2435264.2435273>. Cited on page 35.

Prost-Boucle, Muller and Rousseau 2014 PROST-BOUCLE, A.; MULLER, O.; ROUSSEAU, F. Fast and standalone Design Space Exploration for High-Level Synthesis under resource constraints. **Journal of Systems Architect**, n. 1, p. 79 – 93, 2014. ISSN 1383-7621. Available: <http://www.sciencedirect.com/science/article/pii/S1383762113001938>. Cited on pages 36, 37, and 44.

Roozmeh and Lavagno 2018 ROOZMEH, M.; LAVAGNO, L. Design space exploration of multi-core RTL via high level synthesis from OpenCL models. **Microprocessors and Microsystems**, v. 63, p. 199 – 208, 2018. ISSN 0141-9331. Available: <http://www.sciencedirect.com/science/article/pii/S0141933118300760>. Cited on page 39.

Rosa *et al.* 2016 ROSA, L.; DELBEM, A.; TOLEDO, C.; BONATO, V. Design and analysis of evolutionary bit-length optimization algorithms for floating to fixed-point conversion. **Applied Soft Computing**, v. 49, p. 447 – 461, 2016. ISSN 1568-4946. Available: <http://www.sciencedirect.com/science/article/pii/S156849461630429X>. Cited on page 68.

Rosa, Bouganis and Bonato 2018 ROSA, L. de S.; BOUGANIS, C. S.; BONATO, V. Scaling Up Loop Pipelining For High-Level Synthesis: A Non-Iterative Approach. **The 2018 International Conference on Field-Programmable Technology**, p. 8, 2018. Cited on page 83.

Rosa, Bouganis and Bonato 2018 ____. Scaling Up Modulo Scheduling For High-Level Synthesis. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, p. 1–1, 2018. ISSN 0278-0070. Cited on pages 73, 78, and 84.

Saggese *et al.* 2003 SAGGESE, G. P.; MAZZEO, A.; MAZZOCCA, N.; STROLLO, A. G. M. **Field Programmable Logic and Application: 13th International Conference, FPL 2003, Lisbon, Portugal, September 1-3, 2003 Proceedings**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003. 292–302 p. ISBN 978-3-540-45234-8. Available: <http://dx.doi.org/10.1007/978-3-540-45234-8_29>. Cited on page 52.

Schafer 2016 SCHAFER, B. C. Probabilistic Multiknob High-Level Synthesis Design Space Exploration Acceleration. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 35, n. 3, p. 394–406, March 2016. ISSN 0278-0070. Cited on pages 37, 87, 90, 95, 99, 112, and 113.

Schafer, Takenaka and Wakabayashi 2009 SCHAFER, B. C.; TAKENAKA, T.; WAKABAYASHI, K. Adaptive Simulated Annealer for high level synthesis design space exploration. In: **2009 International Symposium on VLSI Design, Automation and Test**. [S.l.: s.n.], 2009. p. 106–109. Cited on page 34.

Schafer and Wakabayashi 2010 SCHAFER, B. C.; WAKABAYASHI, K. Design Space Exploration Acceleration Through Operation Clustering. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 29, n. 1, p. 153–157, Jan 2010. ISSN 0278-0070. Cited on page 35.

Schafer and Wakabayashi 2012 ____. Divide and Conquer High-level Synthesis Design Space Exploration. **ACM Trans. Des. Autom. Electron. Syst.**, ACM, New York, NY, USA, v. 17, n. 3, p. 29:1–29:19, Jul. 2012. ISSN 1084-4309. Available: <http://doi.acm.org/10.1145/2209291.2209302>. Cited on page 35.

Schafer and Wakabayashi 2012 ____. Machine learning predictive modelling high-level synthesis design space exploration. **IET Computers Digital Techniques**, v. 6, n. 3, p. 153–159, May 2012. ISSN 1751-8601. Cited on pages 35 and 38.

Schryver *et al.* 2011 SCHRYVER, C. D.; SHCHERBAKOV, I.; KIENLE, F.; WEHN, N.; MARXEN, H.; KOSTIUK, A.; KORN, R. An energy efficient FPGA accelerator for monte carlo option pricing with the heston model. In: IEEE. **2011 International Conference on Reconfigurable Computing and FPGAs**. [S.l.], 2011. p. 468–474. Cited on page 29.

Schumacher and Jha 2008 SCHUMACHER, P.; JHA, P. Fast and accurate resource estimation of RTL-based designs targeting FPGAS. In: **2008 International Conference on Field Programmable Logic and Applications**. [S.l.: s.n.], 2008. p. 59–64. ISSN 1946-147X. Cited on page 41.

Shao *et al.* 2014 SHAO, Y. S.; REAGEN, B.; WEI, G. Y.; BROOKS, D. Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures. In: **2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)**. [S.l.: s.n.], 2014. p. 97–108. ISSN 1063-6897. Cited on page 36.

Silva *et al.* 2014 SILVA, B. da; BRAEKEN, A.; D'HOLLANDER, E.; TOUHAFI, A. Performance and resource modeling for fpgas using high-level synthesis tools. In: BADER, M.; BODE, A.; BUNGARTZ, H.-J.; GERNDT, M.; JOUBERT, G.; PETERS, F. (Ed.). **Advances in Parallel Computing**. IOS Press, 2014. v. 25, p. 523–531. ISBN 978-1-61499-381-0. Available: <http://dx.doi.org/10.3233/978-1-61499-381-0-523>. Cited on page 42.

Silva *et al.* 2013 SILVA, B. da; BRAEKEN, A.; D'HOLLANDER, E. H.; TOUHAFI, A. Performance Modeling for FPGAs: Extending the Roofline Model with High-level Synthesis Tools. **Int. J. Reconfig. Comput.**, Hindawi Publishing Corp., New York, NY, United States, v. 2013, p. 7:7–7:7, Jan. 2013. ISSN 1687-7195. Available: <http://dx.doi.org/10.1155/2013/428078>. Cited on page 42.

Silva and Bampi 2015 SILVA, J. S. da; BAMPI, S. Area-oriented iterative method for Design Space Exploration with High-Level Synthesis. In: IEEE. **Circuits & Systems (LASCAS), 2015 IEEE 6th Latin American Symposium on**. IEEE, 2015. p. 1–4. ISBN 978-1-4799-8332-2. Available: <https://ieeexplore.ieee.org/document/7250447/>. Cited on page 37.

Sittel *et al.* 2018 SITTEL, P.; KUMM, M.; OPPERMANN, J.; MÖLLER, K.; ZIPF, P.; KOCH, A. ILP-Based Modulo Scheduling and Binding for Register Minimization. In: . IEEE, 2018. p. 265–2656. ISBN 978-1-5386-8517-4. ISSN 1946-1488. Available: <https://ieeexplore.ieee.org/document/8533507/>. Cited on page 118.

So, Hall and Diniz 2002 SO, B.; HALL, M. W.; DINIZ, P. C. A Compiler Approach to Fast Hardware Design Space Exploration in FPGA-based Systems. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 37, n. 5, p. 165–176, May 2002. ISSN 0362-1340. Available: <http://doi.acm.org/10.1145/543552.512550>. Cited on page 52.

Sotomayor *et al.* 2015 SOTOMAYOR, R.; SANCHEZ, L.; BLAS, J. G.; CALDERON, A.; FERNANDEZ, J. AKI: Automatic Kernel Identification and Annotation Tool Based on C++ Attributes. In: **Trustcom/BigDataSE/ISPA, 2015 IEEE**. [S.l.: s.n.], 2015. v. 3, p. 148–153. Cited on page 34.

Vahid and Gajski 1995 VAHID, F.; GAJSKI, D. D. Incremental hardware estimation during hardware/software functional partitioning. **IEEE Transactions on Very Large Scale Integration (VLSI) Systems**, v. 3, n. 3, p. 459–464, Sept 1995. ISSN 1063-8210. Cited on page 42.

Vandierendonck, Rul and Bosschere 2010 VANDIERENDONCK, H.; RUL, S.; BOSSCHERE, K. D. The Paralax Infrastructure: Automatic Parallelization with a Helping Hand. In: **Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques**. New York, NY, USA: ACM, 2010. (PACT '10), p. 389–400. ISBN 978-1-4503-0178-7. Available: <http://doi.acm.org/10.1145/1854273.1854322>. Cited on page 34.

Venugopalan and Sinnen 2015 VENUGOPALAN, S.; SINNEN, O. ILP Formulations for Optimal Task Scheduling with Communication Delays on Parallel Systems. **IEEE Transactions on Parallel and Distributed Systems**, v. 26, n. 1, p. 142–151, Jan 2015. ISSN 1045-9219. Cited on pages 60, 61, and 62.

Wakabayashi and Schafer 2008 WAKABAYASHI, K.; SCHAFER, B. C. "all-in-c" behavioral synthesis and verification with cyberworkbench. In: **High-Level Synthesis**. [S.l.]: Springer, 2008. p. 113–127. Cited on page 35.

Wall 1948 WALL, H. S. **Analytic theory of continued fractions**. New York, NY: Van Nostrand, 1948. (University series in higher mathematics). Available: <http://cds.cern.ch/record/111131>. Cited on page 58.

Wang 2007 WANG, G. **Ant colony metaheuristics for fundamental architectural design problems**. [S.l.]: University of California, Santa Barbara, 2007. Cited on page 95.

Wang, Liang and Zhang 2017 WANG, S.; LIANG, Y.; ZHANG, W. FlexCL: An analytical performance model for OpenCL workloads on flexible FPGAs. In: **2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)**. [S.l.: s.n.], 2017. p. 1–6. Cited on page 42.

Weerasinghe *et al.* 2015 WEERASINGHE, J.; ABEL, F.; HAGLEITNER, C.; HERKERSDORF, A. Enabling FPGAs in hyperscale data centers. In: IEEE. **2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)**. [S.l.], 2015. p. 1078–1086. Cited on page 29.

Weinberg *et al.* 2005 WEINBERG, J.; MCCRACKEN, M. O.; STROHMAIER, E.; SNAVELY, A. Quantifying Locality In The Memory Access Patterns of HPC Applications. In: **Proceedings of the 2005 ACM/IEEE Conference on Supercomputing**. Washington, DC, USA: IEEE Computer Society, 2005. (SC '05), p. 50–. ISBN 1-59593-061-2. Available: <http://dx.doi.org/10.1109/SC.2005.59>. Cited on page 43.

Wolf and Lam 1991 WOLF, M. E.; LAM, M. S. A Data Locality Optimizing Algorithm. In: **Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation**. New York, NY, USA: ACM, 1991. (PLDI '91), p. 30–44. ISBN 0-89791-428-7. Available: <http://doi.acm.org/10.1145/113445.113449>. Cited on page 52.

Xilinx 2016 XILINX. **Xilinx SDAccel Development Environment User Guide**. 2016. Available: <http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_4/ug1021-sdaccel-tutorial.pdf>. Accessed: 2016-06-16. Cited on page 42.

Xilinx 2016 ____. **Xilinx SDSoC Development Environment User Guide**. 2016. Available: <http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_4/ug1023-sdaccel-user-guide.pdf>. Accessed: 2016-06-16. Cited on page 42.

Xydis *et al.* 2015 XYDIS, S.; PALERMO, G.; ZACCARIA, V.; SILVANO, C. SPIRIT: Spectral-Aware Pareto Iterative Refinement Optimization for Supervised High-Level Synthesis. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, v. 34, p. 155–159, 2015. ISSN 0278-0070. Available: <https://ieeexplore.ieee.org/document/6930749/>. Cited on pages 37 and 94.

Zhao *et al.* 2016 ZHAO, R.; LIU, G.; SRINATH, S.; BATTEN, C.; ZHANG, Z. Improving High-level Synthesis with Decoupled Data Structure Optimization. In: **Proceedings of the 53rd Annual Design Automation Conference**. New York, NY, USA: ACM, 2016. (DAC '16), p. 137:1–137:6. ISBN 978-1-4503-4236-0. Available: <http://doi.acm.org/10.1145/2897937.2898030>. Cited on page 43.

Zhong *et al.* 2016 ZHONG, G.; PRAKASH, A.; LIANG, Y.; MITRA, T.; NIAR, S. Lin-Analyzer: A high-level performance analysis tool for FPGA-based accelerators. In: **2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)**. [S.l.: s.n.], 2016. p. 1–6. Cited on page 38.

Zhong *et al.* 2017 ZHONG, G.; PRAKASH, A.; WANG, S.; LIANG, Y.; MITRA, T.; NIAR, S. Design Space exploration of FPGA-based accelerators with multi-level parallelism. In: **Design, Automation Test in Europe Conference Exhibition (DATE), 2017**. [S.l.: s.n.], 2017. p. 1141–1146. ISSN 1558-1101. Cited on page 38.

Zuo *et al.* 2015 ZUO, W.; KEMMERER, W.; LIM, J. B.; POUCHET, L.-N.; AYUPOV, A.; KIM, T.; HAN, K.; CHEN, D. A Polyhedral-based SystemC Modeling and Generation Framework for Effective Low-power Design Space Exploration. In: **Proceedings of the IEEE/ACM International Conference on Computer-Aided Design**. Piscataway, NJ, USA: IEEE Press, 2015. (ICCAD '15), p. 357–364. ISBN 978-1-4673-8389-9. Available: <http://dl.acm.org/citation.cfm?id=2840819.2840870>. Cited on pages 36, 44, and 111.

APPENDIX

A

HLS TOOLS DIRECTIVE LISTS

Chart 11 – BBO directives and possible locations on the source code.

Acro-nym	Directive	Size	Location	Note	Note
BBO	-O <level>	(0,1,2,3,s,4,5)	-O<level>	gcc specific	
	Scheduling	(dynamic, static, fixed)	-parametric-list-based[=<type>] compiler flag	Bambu specific	
	post re-scheduling	boll	–post-rescheduling	Bambu specific	instruction scheduling
	speculative SDC scheduling	boll	–speculative-sdc-scheduling	Bambu specific	
	Disable chaining	boll	–no-chaining	Bambu specific	
	22 memory options	N/A	N/A	compiler flags	memory

Source: Elaborated by the author.

Chart 12 – IOC directives and possible locations on the source code.

Acro-nym	Directive	Size	Location	Note	Note
IOC	Loop unroll	factor 1 to N	before **for** loops		
	Maximum work group size	int	before kernel	OpenCL specific	Data processing optimization
	Required work group size	(int, int, int)	before kernel	OpenCL specific	
	Number of compute units	int	before kernel	OpenCL specific	
	Number of SIMD work-items	(2,4,8,16)	before kernel	Associated to Required work group size	
	__locall__global	bool	before kernel argument		Memory access optimization
	Pointer Size in Local Memory	int	before variable declaration in argument list	OpenCL specific	
	Register	boll	before variable declaration	OpenCL specific	Local memory attributes
	Memory	boll	before variable declaration	OpenCL specific	
	Number of banks	2^n | n is int	before variable declaration	Bounded to Memory	
	bank width	int	before variable declaration	Bounded to Memory	
	pumped	(void, single, double)	before variable declaration	Bounded to Memory	
	Number of read ports	int	before variable declaration	Bounded to Memory	
	Number of write ports	int	before variable declaration	Bounded to Memory	
	Balancing floating point tree operations	boll	–fp-relaxed compiler parameter	compiler specific	
	Const cache memory size	int	–const-cache-bytes<N> compiler parameter		
	Remove loop-carried dependencies	boll	before loop for each variable		

Source: Elaborated by the author.

Chart 13 – LUP directives and possible locations on the source code.

Acronym	Directive	Size	Location	Note
LUP	function inlining	boll	tcl option	LLVM option
	optimizations	boll	tcl option	
	mem2reg	boll	-mem2reg opt option	
	loop simplify	boll	-loop -loop-simplify opt option	
	dead code elimination	boll	-globaldce opt option	
	link time optimizations	boll	-std-link-opts opt option	
	loop pipelining	boll	-loop-pipeline opt option	
	loop unroll	boll	CFLAG += -mllvm -unroll-threshold=0	
	combine if BBs	boll	SET_COMBNINE_BASICBLOCK	TCL script
	Local memory	boll	set_parameter LOCAL_RAMS	
	operation latency	int	set_operation_latency multiply	
	merge memories	int	GROUP_RAMS	
	max pattern sharing	int	set_parameter PS_MAX_SIZE	
	pattern share add	boll	set_parameter PATTERN_SHARE_ADD	
	pattern share sub	boll	set_parameter PATTERN_SHARE_SUB	
	pattern share bit ops	boll	set_parameter PATTERN_SHARE_BITOPS	
	pattern share shift	boll	set_parameter PATTERN_SHARE_SHIFT	

Source: Elaborated by the author.

Chart 14 – VVD directives and possible locations on the source code.

Acro-nym	Directive	Size	Location	Note	Note
VVD	inline	2 boll options	after loop declaration or function declaration		Loop options
	loop pipeline	int II, 3 boll options	after loop declaration	applies unroll in the inner loops	
	loop unroll	int factor, 2 boll options	after loop declaration		
	allocation	int instances, 3 types	after loop or function declaration		
	expression balance	boll	after loop declaration or function declaration		
	latency	int min, int max	after loop declaration or function declaration		
	loop flatten	boll	after loop declaration		
	loop merge	boll	after loop declaration or function declaration		
	loop trip count	int min, int max, int avg	after loop declaration		
	loop occurrence	int cycle	after loop declaration		
	protocol	int ot floating options	after loop declaration or function declaration		
	interface		for kernel arguments, after kernel declaration	#pragma HLS interface ap_<none|fifo>	scalar interface
	mode	12 modes	for kernel arguments	bounded to interface	
	depth	int	for kernel arguments	bounded to interface	
	resource	54 cores	for kernel arguments or arrays declaration or function declaration	bounded to interface	
	stream	boll	for kernel arguments or arrays declaration or function declaration	bounded to interface	
	latency	int	for kernel arguments	bounded resource	
	meta-data	n/a	for kernel arguments	bounded resource	
	depth	int	for kernel arguments	bounded to stream	
	dimension	int	for kernel arguments	bounded to stream	

Source: Elaborated by the author.

Chart 15 – VVD directives ans possible locations on the source code (Table 14 continuation).

Acro-nym	Directive	Size	Location	Note	Note
VVD	interface		for kernel arguments, after kernel declaration	#pragma HLS interface ap_<none|FIFO>	array interface
	array_map	2 modes, int offset	for kernel arguments or arrays declaration or function declaration	bounded to interface	
	array_-partition	3 modes, int factor, int dimension	for kernel arguments or arrays declaration or function declaration	bounded to interface	
	array_reshape	3 modes, int factor, int dimension	for kernel arguments or arrays declaration or function declaration	bounded to interface	
	data_pack	2 byte_pads	for kernel arguments	bounded to interface	
	clock	clock name	after function declaration		function
	function instantiate	variable name	after function declaration		
	occurrence	int	after function declaration		
	pipeline	int II, 2 boll options	after function declaration		
	unroll	int factor, 2 boll options	after function declaration		
	data flow	boll	after function declaration		
	balance float point ops	boll	compiler unsafe_math_operations flag		compiler options

Source: Elaborated by the author.