Leveraging Hardware Construction Languages for Flexible Design Space Exploration on FPGA

Bruno Ferres

Université Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

Spécialité : NANO ÉLECTRONIQUE ET NANO TECHNOLOGIES

Arrêté ministériel : 25 mai 2016

Présentée par

Bruno FERRES

Thèse dirigée par **Frédéric ROUSSEAU**, Professeur UGA / Polytech, Université Grenoble Alpes
et co-encadrée par **Olivier MULLER**, Maître de Conférences Grenoble INP / Ensimag, Université Grenoble Alpes

préparée au sein du **Laboratoire Techniques de l'Informatique et de la Microélectronique pour l'Architectures des systèmes intégrés**
dans **l'École Doctorale Électronique, Électrotechnique, Automatique et Traitement du Signal (EEATS)**

Leveraging Hardware Construction Languages for Flexible Design Space Exploration on FPGA

Utilisation de langages de construction matérielle pour une exploration flexible des espaces de conception sur FPGA

Thèse soutenue publiquement le **23 mars 2022**,
devant le jury composé de :

Sébastien PILLEMENT
Professeur, Nantes Université, Président et Rapporteur
Virginie FRESSE
Maître de Conférences, Université Jean Monnet, Rapporteur
Christophe JEGO
Professeur, Bordeaux INP, Examinateur
Régis LEVEUGLE
Professeur, Grenoble INP, Examinateur
Pierre-Henri HORREIN
Ingénieur Docteur, Responsable Technique FPGA, OVHcloud, Examinateur
Frédéric ROUSSEAU
Professeur, Université Grenoble Alpes, Directeur de thèse
Olivier MULLER
Maître de Conférences, Grenoble INP, Co-encadrant de thèse

À Carmen CARRION,
À Jacques CARRION,
À Pierrot FERRES,
Merci pour tout.

Remerciements

Tout d'abord, je souhaite remercier mes encadrants pour leur suivi, leurs conseils et leur compréhension pendant toute la durée de ma thèse. Je remercie en premier lieu Olivier, qui m'a donné l'envie d'enseigner lors de mon passage à l'Ensimag, et qui m'a accompagné pendant ces 3 années. Merci pour sa pleine confiance en mes capacités, même quand je ne m'en sentais pas capable, et pour toutes les discussions qui ont guidé mes réflexions et mon approche du travail d'enseignant chercheur. Merci également à Frédéric pour la confiance qu'il m'a accordée, pour ses retours toujours pertinent sur mes travaux, et pour le soutien qu'il a pu m'apporter pendant les moments difficiles.

Je souhaite ensuite remercier Sébastien Pillement et Virginie Fresse, qui ont non seulement accepté de faire parti de mon Comité de Suivi Individuel, mais aussi de rapporter sur mes travaux au terme de ces trois années — merci aussi à M. Pillement d'avoir accepté de présider le jury, au vu des circonstances particulières de la soutenance. Merci également à Christophe Jego et à Régis Leveugle pour leurs rôles d'examinateurs sur mes travaux, et à Pierre-Henri Horrein pour les interactions que l'on a pu avoir pendant ces trois années ainsi que pour sa participation lors de la soutenance.

Je remercie toute l'équipe SLS, qui m'a accueilli, conseillé et écouté, que ce soit sur la teneur de mes travaux ou sur mes ambitions par la suite. Merci donc à Breytner, Arthur V., Arthur P., Julie, Liliana, Frédéric P., Laurence P., Marie, Nathan, Adrien, Ana, Enzo, Thomas, Maxime C., Tiago, Luc, Clément, Damien, Arief, Georgios, Paul et Maxime M. de m'avoir aidé, de près ou de loin, durant ces trois années — merci également à Anne-Laure, Laurence B., Frédéric C. et Ahmed pour leur soutien précieux, qu'il soit technique ou administratif. Un merci particulier à Jean pour son aide précieuse, ses conseils toujours pertinents, et sa bonne humeur générale. Une pensée émue, également, à tous les meubles du 4$^{\text{ème}}$ étage, partis bien trop tôt, mais jamais sans efforts.

Je remercie également les membres des équipes pédagogiques de l'Ensimag et de Polytech de m'avoir accepté dans leurs rangs — merci à Florence, Lionel, Claire, Frédéric P., Olivier, Sébastien, Frédéric R. et Liliana.

Je remercie mes ami·es pour leur soutien sans faille pendant certaines périodes difficiles, et notamment Manon et Nicolas qui ont vécu cette aventure en même temps que moi, et qui ont su m'écouter et me conseiller quand j'en ai eu besoin.

Evidemment, je remercie grandement ma famille pour leur confiance et leur soutien toutes ses années. Merci Maman, merci Papa, merci Laurent, et le reste de cette grande famille.

Enfin, merci à Wendy, qui me soutient, m'écoute, me conseille et surtout me supporte depuis toutes ces années. Sans elle, ce manuscrit n'existerait pas.

Abstract

In a world where the required computational capacitities grow exponentially, FPGA-based hardware accelerators are imposing themselves as energy efficient alternatives to general purpose CPUs. However, while the software development methodologies can rely on new paradigms and techniques to improve the productivity, designing a digital circuit remains a daunting task where both expertise and time are primordial.

In order to increase the productivity of hardware developers, we explore the possibility of using a novel design paradigm called Hardware Construction Languages, which enables building parametrized design generators — increasing both code reusability and parametrization — and exploiting high level features such as object-oriented or functional programming.

The first contribution of this project aims at easing comparison of accelerators by exposing different estimation metrics and methodologies, in order to provide designers and tools with interesting feedbacks.

We then consider leveraging this new paradigm to generate and compare accelerators — introducing two complementary methodologies: *meta design* and *meta exploration*. Meta design is based on the prior analysis of a given algorithm to implement a parametrized design generator, where every generated design belongs to a design space to be explored. Meta exploration is then used to leverage the users expertise of both application domain and target execution board for an efficient exploration of so defined design space.

We choose Chisel as an HCL candidate, and introduce QECE — *Quick Exploration using Chisel Estimators* — as a demonstrator for both contributions. As Chisel is built on top of Scala, we hence bring high level features from software development to the hardware world. We finally leverage the introduced methodologies by developing various representative FPGA applicative kernels, and expose various *scenarii* of estimation and exploration.

Résumé

DANS un monde où le besoin de ressources de calcul croit exponentiellement, les accélérateurs matériels à base de FPGA s'imposent comme alternatives à haute efficacité énergétique aux processeurs généralistes. Cependant, alors que les méthodes de développement logiciel profitent de nouveaux paradigmes pour améliorer la productivité, la conception de circuits numériques demeure une tâche compliquée où le temps et l'expertise restent cruciaux.

Afin d'améliorer la productivité des développeurs matériels, nous explorons la possibilité d'utiliser un nouveau paradigme basé sur les langages de construction matérielle, qui permettent de construire des générateurs paramétriques de circuits, améliorant à la fois la réutilisabilité et la paramétrisation, et d'utiliser des fonctionnalités de haut niveau telles que la programmation orientée objet ou encore la programmation fonctionnelle.

La première contribution de ce projet vise à faciliter la comparaison d'accélérateurs en exposant différentes métriques et méthodologies d'estimation de circuits, afin de fournir aux développeurs et aux outils des retours constructifs sur le processus de développement.

Nous nous intéressons ensuite à l'exploitation de ce nouveau paradigme pour la génération et la comparaison d'architectures, et introduisons deux méthodologies complémentaires: la *méta conception* et la *méta exploration*. La méta conception est basée sur une analyse préalable de l'algorithme cible afin de concevoir un générateur paramétrique de circuits, où chaque implémentation générée s'intègre dans un espace de conception à explorer. La méta exploration est ensuite utilisée afin de mettre à profit l'expertise de l'utilisateur à propos du domaine applicatif et du matériel cible, permettant une exploration efficace de l'espace ainsi généré.

Parmi les langages de construction matérielle disponibles, nous choisissons Chisel afin de concevoir QECE — *Quick Exploration using Chisel Estimators* — comme démonstrateur pour les deux contributions. Comme Chisel est basé sur Scala, nous amenons ce faisant des fonctionnalités de haut niveau du développement logiciel au monde du matériel. Finalement, nous démontrons l'utilisabilité des méthodologies présentées en développant un ensemble de noyaux applicatifs représentatifs de l'utilisation des FPGA, et en mettant en avant différents scénarios d'estimation et d'exploration.

Contents

1 Introduction **1**

2 Motivations **3**
2.1 Hardware Design 4
2.2 Design Space Exploration 13
2.3 Motivations and Organization 16

3 State of the Art **19**
3.1 Overview of the Existing Tools 21
3.2 Design Space Exposition 22
3.3 Metric Definition and Estimations 26
3.4 Exploration Strategies 31
3.5 Synthesis on the Existing Approaches 36

4 Building and Integrating Estimators **39**
4.1 Importance of Qualitative Estimations 40
4.2 Resource and Timing Estimations 42
4.3 Quality of Service Estimation 52
4.4 Integrating Metrics in a HCF 54
4.5 Synthesis on the Estimation Methodologies 57

5 Design Space Exploration Methodology **59**
5.1 Defining DSE in a HCL context 60
5.2 Building Explorable Architectures 62
5.3 On Functional Programming for DSE 64
5.4 Discussions on the Proposed DSE Methodology 77

6 Experiments and Results **79**
6.1 Building a Software Demonstrator 80
6.2 Application Benchmark 85
6.3 Experimental Setups 86
6.4 Quality of the Estimators 86
6.5 Comparing the Exploration Strategies 93
6.6 Synthesis on the Experiments 109

7 Conclusion and Perspectives **111**

8 Résumé **115**
8.1 Motivations 116
8.2 État de l'art 116
8.3 Métriques et méthodologies d'estimation 119
8.4 Exploration d'espace de conception 120
8.5 Expérimentations et résultats 123
8.6 Conclusion 125

Appendixes

A Chisel Basics **129**

B Benchmark Composition **139**

C Quality of Resource Estimation **149**

D Comparing the Pruning Strategies **155**

Backmatter

Publications **162**

Glossary **163**

Bibliography **169**

List of Figures

2.1 Xilinx Virtex 7 FPGAs structure 5
2.2 Xilinx Configurable Logic Blocks 6
2.3 Example of RTL design flow 7
2.4 Example of DSL design flow 8
2.5 Example of HLS design flow 9
2.6 Example of HCL design flow 12
2.7 Pareto frontier representation 14

3.1 Taxonomy for HLS based DSE approaches 32

4.1 Basic estimation methodology 47
4.2 Multiply and accumulate macro block 48
4.3 Multiplexer macro block 49
4.4 Memory macro block 49
4.5 Macro block based estimation methodology 51
4.6 Taxonomy proposition for QoS estimators 53
4.7 Proposition for Chisel flow enhancement 55

5.1 Meta design methodology 60
5.2 Meta exploration methodology 61
5.3 Annotation based parameter generation 63
5.4 Building complex strategies 76

6.1 Object hierarchy for the implemented strategies 81
6.2 Proposing different space implementations 83
6.3 Quality of resource estimations on various kernels 88
6.4 Quality of frequency estimations on GEMM 91
6.5 Quality of Service estimations on dot product 92
6.6 Quality of resource estimations on GEMM 96
6.7 Quick pruning of design space 98
6.8 Comparing the accuracy of the pruning strategies 106

8.1 Méthodologie de méta conception 121
8.2 Méthodologie de méta exploration 122

A.1 Implementing an increment module using Chisel 131
A.2 Building a register: HDL *vs* HCL 134

A.3 Example of a dot product architecture to generate 137

B.1 Simple Role and Shell model used 140
B.2 Targeted chronogram for an efficient GEMM implementation . 141
B.3 Monte Carlo kernel meta architecture 144
B.4 Simplified schematic for a neuron implementation 147

C.1 Quality of resource estimation on Black Scholes 150
C.2 Quality of resource estimation on Pi 151
C.3 Quality of resource estimation on FFT 152
C.4 Quality of resource estimation on dot product 153

D.1 Quality of service evolution ($\mathcal{N}(0,1)$, 10 simulations) 158
D.2 Quality of service evolution ($\mathcal{N}(0,1)$, 50 simulations) 158
D.3 Quality of service evolution ($\mathcal{N}(0,1)$, 100 simulations) 159
D.4 Quality of service evolution ($\mathcal{N}(0,1)$, 1000 simulations) 159
D.5 Quality of service evolution ($\mathcal{N}(32,10)$, 10 simulations) 160
D.6 Quality of service evolution ($\mathcal{N}(32,10)$, 50 simulations) 160
D.7 Quality of service evolution ($\mathcal{N}(32,10)$, 100 simulations) . . . 161
D.8 Quality of service evolution ($\mathcal{N}(32,10)$, 1000 simulations) . . . 161

List of Tables and Listings

Tables

4.1 Estimation quality and abstraction level 42
4.2 Operator impacts on estimations 44

6.1 Experimental setups characteristics 86
6.2 Temporal behaviour of resource estimators 87
6.3 Timing estimations over GEMM implementations 90
6.4 Strategy comparison with no QoS concerns 94
6.5 Pruning results with respect to used optimizations 98
6.6 Comparing pruning strategies on Black Scholes 106
6.7 Exploration results on Black Scholes 107

B.1 Benchmark composition 148

D.1 Comparison of the pruning strategies on dot products 157

Listings

5.1 Exposing the dot product design space 62
5.2 Enhancing the dot product design space 64
6.1 Gradient based strategy definition 85
6.2 Expertise-based design space for Black Scholes meta design . . 101
6.3 Defining an helper for empirical quality of service estimations 103
6.4 Transform helper for the exploration 103
6.5 Black Scholes exploration strategy 105
A.1 Type parametric adder generator in Chisel 133
A.2 Behavioural description of a register in Verilog 134
A.3 Building a register using Chisel constructs 134
A.4 Building a dot product kernel using Chisel 135
A.5 Dot product generic implementation in Verilog 136

1

Introduction

THE usage of digital circuits has grown exponentially for the last decades, and embedded systems can be found everywhere nowadays. In fact, the semiconductor industry leverages billions of euros each year, and circuits tend to be smaller, denser and more energy efficient. From smartphones to pay machines to super computers, electronic systems are being designed every day, and the number of hardware designers — which build them — is increasing accordingly.

However, while the software developers — which are their counterparts from the computer science world — have benefited from conceptual advances those past years to improve their productivity, designing a digital circuit remains a daunting task that require both time and expertise.

Initiatives are thus being proposed in order to ease the life of hardware developers, by providing faster processes and simpler ways to describe the behaviour of an electronic system.

Among them, a trend has been growing since the 80's that aim at developing circuits from more abstract descriptions, such as software programs, instead of the verbose languages that are typically used. This approach is based on automatic tools that iteratively compare and select circuits, as a given code can be translated in many different ways to actually produce a circuit with the same functionality. In order to find a best solution in a set of various circuits, the developers thus rely on the automatic exploration of a space composed of the different designs, which would otherwise be a tedious and long task for them.

Nevertheless, while this approach has grown mature and is now used in industrial processes, it is difficult to provide tools that produce clever decisions during their exploration, and building an efficient software for this task is still widely being investigated in both academic and industrial worlds.

On the other hand, different classes of circuits are to be considered depending on their usage and functioning environment. Indeed, the development processes can heavily differ, resulting in the need for specific work flows, depending on both the target technology and the applicative domain.

While application specific integrated circuits are used in most integrated systems — such as smartphones and components from the Internet of Things — some use cases require to be able to modify and adapt the functionality provided by a hardware design.

Among them, *Field-Programmable Gate Arrays* (**FPGA**) are reconfigurable circuits that can be used to implement algorithms on digital electronics. On such technology, a simple process can be used to change the circuit hosted in the FPGA, in order to modify its functionality. Doing so, the computations can be faster than they would be on a standard processor, but the resulting circuit can be modified at any time, rather than being fixed as it is the case with dedicated circuits.

The adaptability of those reconfigurable circuits makes them excellent candidates for the design of hardware accelerators — *i.e.* digital designs that can be used to speed-up specific computations in many electronic system, while reducing the required power consumption.

Designing a digital circuit is a tedious task relying either on old and time consuming technologies, or on novel approaches which leverages automatic tools but are still limited in their usage — and it is even more true for FPGA-based implementations, as the heterogeneous structures of the targets make it even more difficult to build generic and reusable designs. To cope with those limitations, another approach has emerged recently, improving the older programming languages while avoiding the limitations introduced by using more abstract descriptions. This approach leverages recent software techniques, which are adapted to digital design to provide the developers with new methods to describe a circuit.

We present an initiative that leverage recent languages based on this approach to increase the productivity of hardware developers. More specifically, we propose an exploration tool that can be configured by the designers to adapt to their use cases, and uses high level features from the software world to bring more expressivity to its users.

2

Motivations

This chapter introduces the problematic of this thesis, as well as the issues and challenges at stake in this work. Important considerations are discussed to highlight some interrogations that will be answered throughout this manuscript.

To begin with, we introduce the notion of hardware acceleration, with a particular focus on how *Field-Programmable Gate Arrays* (**FPGA**) can be used to build hardware accelerators. We then expose the standard design methodologies to discuss their limitations, and introduce the *Design Space Exploration* (**DSE**) processes as a way to increase the productivity of hardware developers. Finally, we discuss the usage of a new development paradigm — known as *Hardware Construction Languages* (**HCL**) — to cope with the limitations of the standard design methodologies, and how it can be used for efficient **DSE** implementations.

Table of contents

2.1 Hardware Design 4
2.1.1 Hardware Acceleration 4
2.1.2 FPGAs as Hardware Accelerators 5
2.1.3 Standard Paradigms 6
2.1.4 Hardware Construction Languages 10
2.2 Design Space Exploration 13
2.2.1 Definitions and Interests 13
2.2.2 Standard Approaches 14
2.2.3 Limitations 15
2.3 Motivations and Organization 16
2.3.1 Problem Statement 16
2.3.2 Thesis Organization 17

2.1 Hardware Design

2.1.1 Hardware Acceleration

While people are growing familiar with the concept of generic purpose *Central Processing Units* (**CPU**) integrated in their embedded devices and computers, they might not be familiar with the notion of hardware acceleration. Hardware accelerators are digital circuits built to cope with the limitations of **CPUs**, *i.e.* insufficient energy efficiency or performance, and can be found in many application domains ranging from image processing to network filtering. They relies on a particular trade-off between a circuit efficiency and its programmability: **CPUs** can be programmed for every possible usage manipulating digital data, but an *Application-Specific Integrated Circuit* (**ASIC**) implementing a given algorithm will perform way faster than a **CPU** providing the same functionality, and consume way less energy — however the **ASIC** function is fixed.

Other accelerators do exist on this programmability *vs* performance range, varying from domain specific processors such as *Graphical Processing Units* (**GPU**) or *Digital Signal Processors* (**DSP**) to (re)configurable circuits, based on fabrics of basic operators that can be (re)programmed to perform any operation, providing an interesting trade-off between programmability and performance/energy efficiency.

Domain specific processors are promising candidates for hardware acceleration, as programming processes are similar to software development. Usages are evolving to cope with hardware acceleration needs — *e.g.* **GPUs** are now used in many computation intensive algorithm acceleration beside graphical processing, using particular programming patterns (such as matrix operations) to take advantage of the inherent structure of the hardware. However, such development processes are limited to particular usages (either domain specific algorithm or particular patterns), and are thus not appropriate for every usages.

On the other hand, digital circuits (such as **ASICs** or configurable circuits), require specific development processes, known as **hardware development processes**. In this context, configurable circuits are also to be considered when building hardware accelerators, as their programmability allows evolution abilities while offering performances and energy efficiency orders of magnitude better than **CPU** implementations. Among them, *Field-Programmable Gate Arrays* (**FPGA**) — digital chips built as arrays of reconfigurable basic blocs — are commonly used for hardware acceleration. However, developing a hardware accelerator for a given algorithm requires expertise about the target, as an **ASIC** implementation is way different from

a **FPGA** one, and specific knowledge is to be brought by the developer in order to develop efficient circuits. Hardware design is thus a time consuming task which requires a lot of effort and expertise, and initiatives are taken to ease and accelerate the work of hardware developers.

2.1.2 *Field-Programmable Gate Arrays* as Hardware Accelerators

As **FPGAs** offer interesting performances for hardware implementation while remaining more programmable than **ASICs**, they have been used as hardware accelerators for a long time, and keep providing promising circuits in various domains, such as network filtering [BHM+21], neural network implementations [NVS+17] or DNA sequencing [DTOBS17].

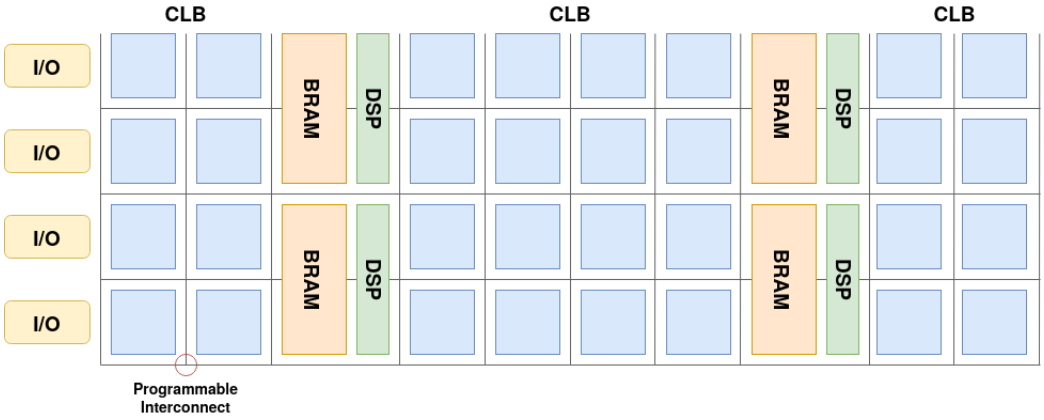


Figure 2.1: Simplified schematic of Xilinx Virtex 7 FPGAs structure

In order to build efficient design methodologies for **FPGA** based accelerators, one must understand board structures to comprehend technological specificities. Figure 2.1 introduce a simplified structure of a Xilinx **FPGA** as an example. As can be remarked, the structure is inherently heterogeneous, area being distributed between *Input/Output blocks* (**IO**), *Configurable Logic Blocks* (**CLB**), *Digital Signal Processors* (**DSP**) and *Block RAMs* (**BRAM**). Digital functions are based on **CLBs** (Fig. 2.2, as defined in [Xil16]) that include both computation resources with *Look-Up Tables* (**LUT**) and memory resources with *Flip Flops* (**FF**), but **DSP** blocks can be used to perform specific computations like multiplications, and **BRAM** can be used as embedded memory to offer large memories with low access latency. At least four different resources are thus to be considered when developing a hardware accelerator for such target — **LUT**, **FF**, **DSP** and **BRAM** — depending on both objectives and constraints of the problem.

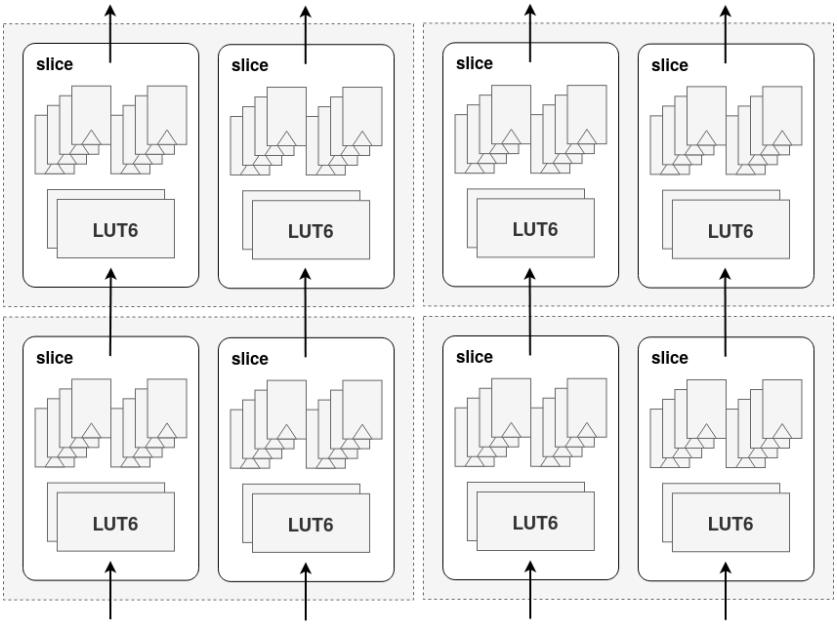


Figure 2.2: *Configurable Logic Blocks* (**CLB**): Xilinx FPGAs basic blocks

Nowadays, most of **FPGAs** are integrated in various *Systems on Chips* (**SoC**), tightly coupled with **CPUs** and other peripherals. However, in the context of this work, we will only consider **FPGA** implementation of different algorithms, and try to ease the life of **FPGA** developers, with no further focus on the integration of the accelerators.

Such considerations raise the first interrogations of this manuscript:

- How can we ease hardware development ?
- How can we integrate **FPGAs** specific constraints in hardware design methodologies ?

2.1.3 Standard Paradigms

As circuit complexity (*i.e.* number of transistors used in a chip) is growing exponentially, as stated by **Gordon Moore's law** in 1965 [Moo65], hardware designs methodology are bound to evolve and exhibit high level abstraction to developers to cope with the scaling of *Very Large Scale Integration* (**VLSI**).

First languages developed to describe hardware circuits by abstracting low level considerations were *Hardware Description Languages* (**HDL**) such as VHDL and Verilog. Originally developed for hardware simulation, working at *Register-Transfer Level* (**RTL**) — *i.e.* considering data signal as entity

instead of considering physical implementation details such as transistor layout and power supply — their aim was to raise the abstraction level and allow design of large scale circuits. To do so, they allow to describe complex circuits by combining behavioural descriptions (how digital signals are behaving in the circuit, often used to describe basic modules with simple functionalities), and structural descriptions (how are basic blocks interacting). **HDL** classic development processes are quite straightforward, as can be seen in Figure 2.3 (as translated from Prost-Boucle's thesis [PB14]), consisting in manual translations of sequential algorithm to hardware description of the functionality. Circuit descriptions are then fed to a set of time consuming software — including **synthesis** and **place and route** steps — in order to translate such high level descriptions to physical representations that can be used to program **FPGAs**. Moreover, in order to comply with design constraints and objectives, such process requires manual iterations to find an acceptable fit, and each iteration can be expensive due to the complexity of both synthesis and place and route processes, and may require manual modifications of the original design, which can be both time consuming and error prone tasks.

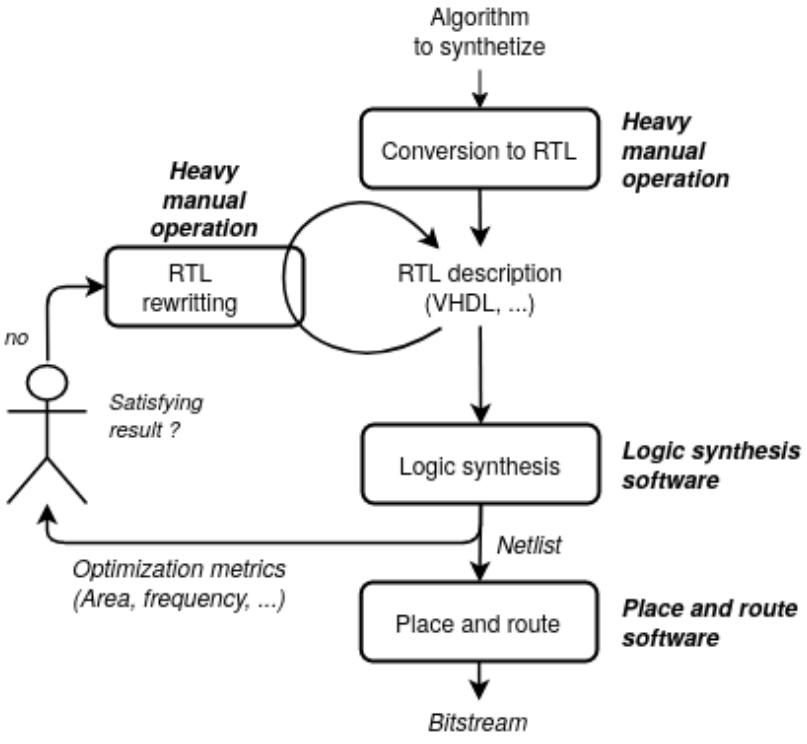


Figure 2.3: Example of RTL design flow

Another approach for easing hardware development is based on *Domain Specific Languages* (**DSL**). **DSLs** are languages leveraging prior knowledge about the application domains (what are the important operations to perform, what data patterns are frequently used, ... *i.e.* what domain specific optimizations can be performed) to build efficient mapping for acceleration. In this way, this approach differs from **DSP** usage, as the goal is not to program specific processors but rather build generic circuits — *e.g.* on **ASIC** or **FPGA** — using specificity of a particular domain. A typical **DSL** development flow (quite similar to **HDL** ones) is presented in Figure 2.4. Using domain specific knowledge allow easier writing and refinement of the **DSL** code base, compared to **HDL** programming, resulting in faster iteration steps, and feedback generations can also be enhanced by using pre-synthesis estimations based on **DSL** operations, producing faster development methodologies. In fact, providing early estimations of iteration metrics — *e.g.* area and frequency — is a common technique to accelerate development flows by reducing the time dedicated to synthesis and place and route steps, and it was used as used as a basis of one of the main initiatives for easier development: *High Level Synthesis* (**HLS**).

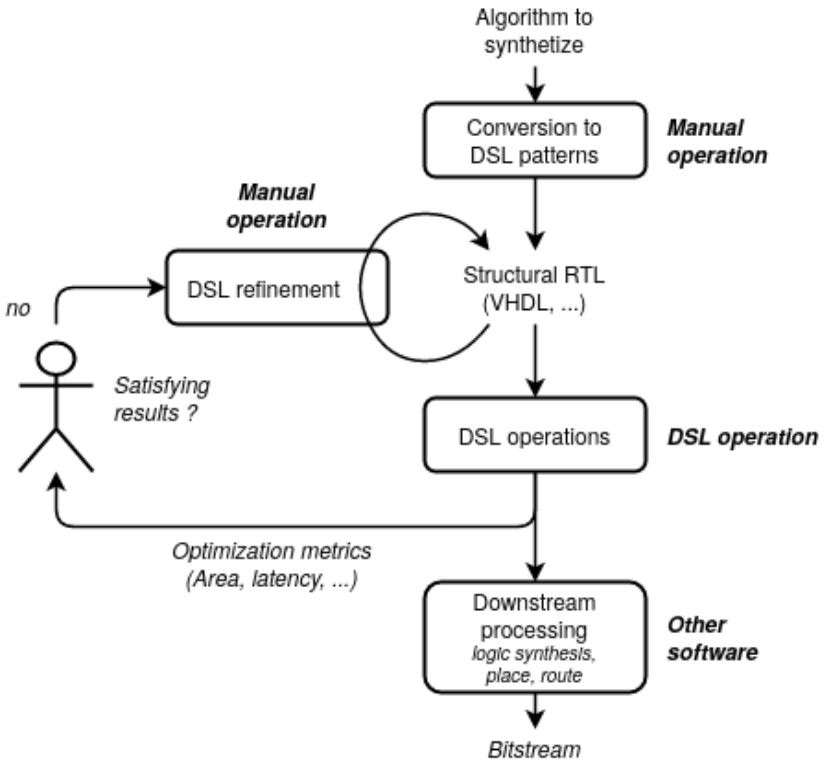


Figure 2.4: Example of DSL design flow

HLS methodologies have been developed to close the gap between software developers and hardware designers. It is based on a modification of the entry point of the whole process, as it does no longer require to use hardware specific languages such as **HDLs** or **DSLs**, but rather uses standard software languages such as **C language** to describe the target algorithm, and then relies on a translation tool that compiles this algorithmic approach to a hardware description at **RTL** level. Figure 2.5 (also translated from Prost-Boucle's thesis [PB14]) introduces a classical **HLS** based design flow, operating on an intermediate representation issued from the compilation of the input algorithm to perform optimization-estimation steps until given constraints and objectives are respected. Manual interventions of the developer are reduced as the software is able to modify the generated circuits in order to optimize the design, but can be needed in order to guide the tool by adding specific knowledge — using optimization directives (such as **C pragmas**) directly in the algorithm description to define memory patterns or exhibit parallelism for example. This feature allows to close the performance gap with **HDL** design flows, but as it requires specific hardware knowledge, it can no longer directly be used by software neophytes and require prior skill improvement.

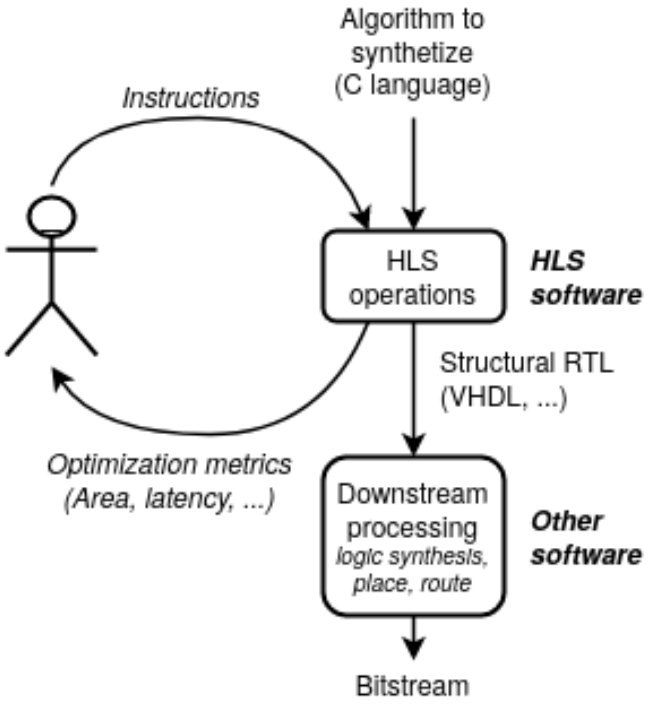


Figure 2.5: Example of HLS design flow

A quick review of standard paradigms for hardware development hence exhibits some interrogations about classical design methodologies:

- What are the strength of standard paradigms for hardware development ?
- What are their limitations ?
- How can we ease hardware development processes ?

2.1.4 Addressing Standard Paradigms Limitations through *Hardware Construction Languages*

Standard paradigms for hardware development have evolved the last decade in order to cope with the needs of faster development methodologies and close the gap with software processes. However, such methodologies present limitations that one should try to overcome in order to ease the life of hardware developers. **HDL** development requires time consuming and error prone processes, and does not allow easy component re utilization, as evolving a design for a new use case require heavy manual modification of the base code. On the other hand, **HLS** methodologies can accelerate development processes, and code re utilization is easier as the base code is lighter than with **HDLs** — nonetheless, modifying an existing accelerator requires an advanced knowledge of inserted directives which may not be intuitive, and the whole process relies on automatic tool inferences. Algorithmic compilation toward hardware description is a difficult problem, as it implies a change of paradigm, from a sequential representation of the algorithm toward a behavioural definition of a circuit. In order to do so, the tool is taking decision that are usually taken by expert developers in standard **HDL** methodologies that may generate non optimal designs, and wrong directive usage may accentuate this flaw. Moreover, with the growing need of performance and efficiency, some of those decisions may be very specific to produce optimal utilization of available resources, and programming languages should allow developers to have full control over generated hardware — and it cannot be done if automatic decisions are taken without the user knowing it. In fact, **HLS** enables higher abstraction of circuits for design processes, but results in a lack of control over how those circuits are generated. As for **DSLs**, their usage is by-design limited to specific domains, and thus cannot be generalized as hardware development methodologies.

To cope with those problems and improve the expressivity of developers while maintaining control over generated hardware, new paradigms are emerging, with *Hardware Construction Languages* (**HCL**) among them.

HCLs are based on high level languages, allowing to build hardware generators instead of hardware circuits: instead of building a specific accelerators for a given task, such as a digital *Finite Impulse Response Filter* (**FIR Filter**) with 128 taps operating with 32 bits fixed point numbers, it can be used to build a parametrized **FIR Filter** generator that can be used to build various circuits, varying on the number of taps or the data type used for example. To do so, they leverage high level features such as *Object-Oriented Programming* (**OOP**), functional programming or reflexivity in order to provide more generic and reusable representations for hardware circuits. Moreover, as **HCLs** are working at **RTL** like standard **HDLs**, no performance/area overhead is introduced from using **HCL** over any other **HDL** [IKL+17], resulting in controllable generation of hardware implementations. Figure 2.6 represents a typical **HCL** development flow, where the main difference with respect to standard paradigms is that both generator description and parameters definition are exposed by users, allowing to manage generated circuits while easing reuse and iteration over generated designs through exposed programming features.

HCLs relies on *Hardware Construction Frameworks* (**HCF**) to translate target-independent **RTL** code to technology dependent **RTL**, leveraging a compiler-like separation of concerns: using an *Intermediate Representation* (**IR**) of circuits to perform optimization and code generation, target specific concerns are decorrelated from **HCL** description as they can be managed directly by operating on the **IR**. Common **HCLs** use high level languages such as **scala** for *Constructing Hardware in a Scala Embedded Language* (**Chisel**) [BVR+12], **python** for PyMTL [LZB14] or **Haskell** for Cλash [BKK+10].

As **Chisel** is a promising **HCL**, with state of the art implementations from both academic and industrial worlds, such as in-order and out-of-order RISC-V implementations [CPA15] [AAB+16] or Google latest *Tensor Processing Unit* (**TPU**) [LT18], it will be used as an example of **HCL** in this work. **Chisel** development flow is quite similar to standard **HDL** flows as it relies on the same tool set to generate both constraint and objective metrics and resulting bitstream — in fact **Chisel** is integrated in standard **HDL** flows by generating structural **RTL** (*Verilog*) code after what is called the emission phase (*i.e.* building a non parametrized hardware circuit from the corresponding hardware generator). However, as both hardware generator notion and high level programming features for hardware development are recent progresses, improvements of **HCL** based methodologies are still to be proposed.

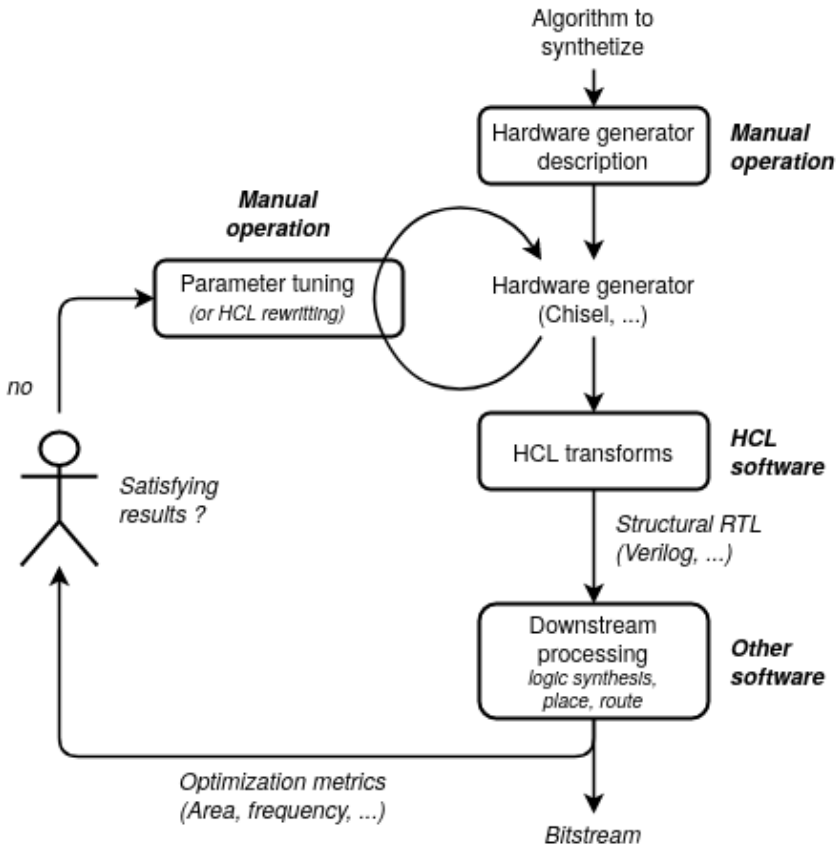


Figure 2.6: Example of HCL design flow

Emergence of **HCL** paradigm hence raises few interrogations on the improvement that new technologies and methodologies can bring to the industry of semi-conductor:

- How can **HCLs** cope with standard paradigms limitations ?
- How can high level programming features be used in hardware development ?
- What can **HCLs** bring to hardware developers ?
- How can **HCLs** flow be improved for easier iterations over generated designs ?

Moreover, as **Chisel** will be used as a basis for this work, an analysis of its interesting features is provided in Appendix A. It should help readers to understand the opportunities brought by the **HCL** paradigm, using simple examples of **Chisel** usage to expose differences from standard **HDLs**.

2.2 Design Space Exploration

2.2.1 Definitions and Interests

It may seem counter intuitive, but hardware designers expertise is more about decision making than describing circuits. As a matter of fact, implementing a particular algorithm for a given target can be done in a lot of different ways, and it is up to the developer to choose among different implementation options to build optimal circuits with respect to its use case. However, even with senior expertise, optimal solutions may not be trivial, and developer may not even consider them in the process, resulting in suboptimal designs.

To cope with this problem, *Design Space Exploration* (**DSE**) methodologies have emerged, allowing exploration and comparison of implementation options at different granularity levels — options can vary from basic operation implementations such as multiplication algorithm, to global parameters, such as parallelism level of the design. A design space is defined as the set of every possible implementation candidate for a given algorithm, and as implementation options may grow exponentially with circuit complexities, exhaustive exploration processes may be unfeasible in an acceptable amount of time. Due to that, more complex exploration strategies were built to reduce space traversal, thus reducing exploration time while maintaining satisfying *Quality of Results* (**QoR**), *i.e.* finding implementations with performances comparable to a global optimal implementation as could be found from exhaustive space traversal.

However, one cannot define a generic, optimal solution for exploration, as a lot of parameters are both target and algorithm specific [SW20]. Moreover, design space definition is not trivial either, as defining the implementation options require either automatic tool inferences, or programming expressivity for developers to bring their expertise in the process. Ergo, **DSE** methodologies remain a prolific search domain, and standard approaches are continuously being improved — raising even more interrogations:

- What does **DSE** bring to hardware developers ?
- Which features are necessary for efficient **DSE** ?
- Which features could be useful ?

2.2.2 Standard Approaches

As exhaustive exploration remains impracticable for complex circuits, more clever exploration strategies were developed to fasten exploration processes. In order to do so, one of the main approach is based on **Pareto optimal solutions** [SW20]: considering both a performance and a cost metric, Pareto optimal solutions are implementations that are optimal in their neighbourhood, meaning that modifying a parameter will result in either a more costly or a less performant solution. The goal is thus to build algorithm to approximate the Pareto frontier without exhaustive traversal of the design space, resulting in optimal design finding in a heavily reduced amount of time. Figure 2.7 introduce a classical, Pareto based **DSE** methodology, using resource usage (which can be amount of transistor, of **LUT**, of **DSP**, ...) as a cost metric, in order to find best implementations under both resource and performance constraints. It enables design space partitioning to exhibit a reduced number of implementations to developers, helping them in the design process, as they can now choose among a set of implementations with a warranty that they did not miss potentially optimal solutions.

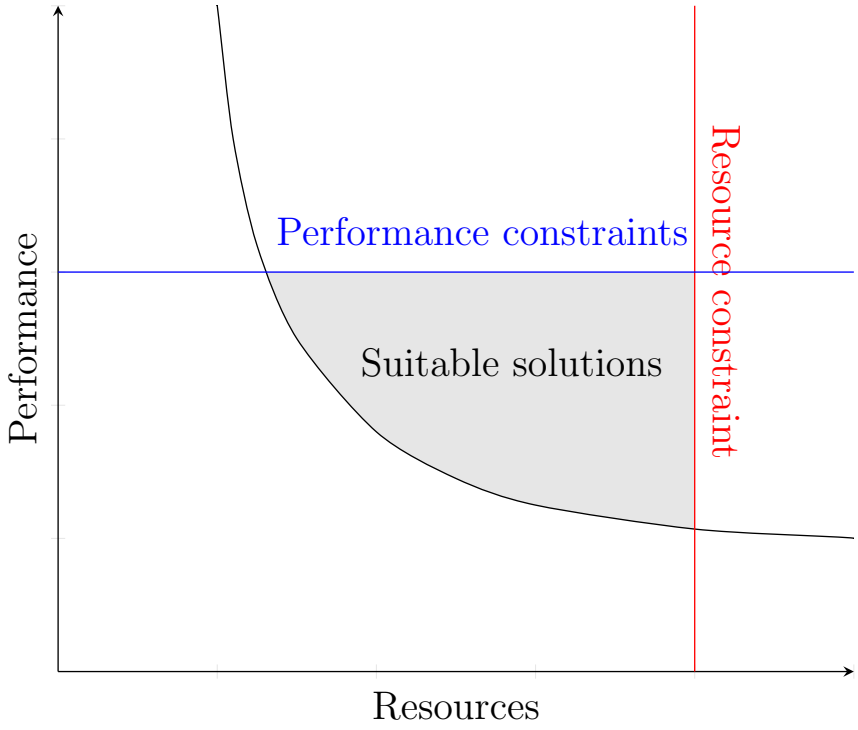


Figure 2.7: Example of Pareto optimal solutions finding under resource and performance constraints

As shown in Figure 2.5, standard **HLS** development flows rely on **HLS** operations over generated circuit in order to generate acceptable solutions with respect to objective and constraints. To do so, **DSE** is used in most **HLS** tool as a medium to find acceptable solutions, varying implementation parameters from the original algorithm, such as loop unrolling, operation pipelining of memory partitioning: by giving freedom to the tool to make its own decision, a heavy design space is generated to compare a lot of different implementations. In order to reduce design space width, most flows thus use algorithms to approximate the Pareto frontier, using specific heuristics such as machine learning [NKO19] [FKA+20] or genetic algorithms [MKC+20], and building efficient exploration strategies remains a trending search topic. Moreover, optimization directives can also be given to guide the exploration of the generated design space, resulting in efficient **DSE** processes.

However, defining optimization directives requires user expertise, and automatic inferences about the design space and how to explore it may result in a lack of control of generated accelerators, which may lead to the impossibility to find the best fit — especially on specific targets such as **FPGAs** where resource heterogeneity requires experience to produce clever decisions.

Such considerations about standard **DSE** approaches bring interrogations about such processes:

- What are the limitations of standard **DSE** approaches ?
- Can **HCLs** help addressing those limitations ?

2.2.3 Limitations

While standard **DSE** approaches have grown mature the last two decades — especially with the rise of **HLS** tools — limitations are still to be addressed to provide generic methodologies appropriate for every possible use cases.

We discussed the challenge of building an interesting design space to explore, as automatic inferences of the tools used to generate implementation options allow multiple variations to be explored, but might result in an uncontrollable design generation in the end.

Moreover, doing so, the generated design spaces are composed in a vast majority of known sub optimal solutions, as no particular expertise is used at design space generation step. It results in heavy design spaces that cannot be explored exhaustively, with the emergence of exploration strategies such as Pareto approximations to cope with the large amount of possible implementations, even if a lot of those implementations are not sound and would never be considered by a hardware developer.

Last but not least, most of **DSE** tools define their own strategies, as well as metrics to optimize in the process, resulting in methodologies that are hard to adapt and reuse to any new use case. In fact, some particular use cases require to define specific metrics to optimize and consider, as well as particular exploration strategies, and tools should allow developers to bring their expertise and knowledge to the exploration process definition. For example, specific domains such as *Approximate Computing* (**AxC**) requires to consider the *Quality of Service* (**QoS**) of circuits in order to provide guarantees about the functionality of a design. However, most **DSE** tools does not allow users to add such metric in the exploration process, resulting in the need to use different flows to consider multiple metrics.

In this context, one may wonder what can **HCLs** bring to hardware developers, especially in the context of **DSE**:

- What can flexibility and genericity bring to **DSE** ?
- How can users define use case specific metrics ?
- How can users build custom exploration strategies ?
- How can **HCLs** be used for efficient **DSE** ?

2.3 Motivations and Organization

2.3.1 Problem Statement

This chapter introduces the motivations of the work presented in this thesis. We discuss the role of **FPGAs** in the context of hardware acceleration, and expose various methodologies used to develop hardware accelerators relying on **FPGAs** specificities. Among them, we introduce **HCLs** as an emerging programming paradigm, and consider their usage to cope with standard paradigm limitations. We put a particular focus on **DSE** processes as a way to increase hardware developer productivity, and discuss **HCLs** usage to improve those processes.

With respect to those considerations, the goal of this thesis is hence to answer those questions:

- **How can FPGA designers productivity be improved using HCLs ?**
- **Which limitations of standard paradigms for hardware development can be addressed using HCLs ?**
- **How can HCLs be used for efficient DSE on FPGAs ?**

2.3.2 Thesis Organization

In order to provide an intelligible analysis of the usage of *Hardware Construction Languages* (**HCL**) to build a flexible *Design Space Exploration* (**DSE**) framework that target *Field-Programmable Gate Arrays* (**FPGA**), this thesis is organized as follows.

Chapter 3 outlines the related works of the literature on efficient **DSE** for **FPGAs**, as well as their limitations. Chapter 4 discusses the need of qualitative estimators to increase the developers productivity, and proposes interesting metrics to be considered in design and exploration processes, as well as relevant estimation methodologies to be used. Chapter 5 introduces the usage of **DSE** in an **HCL** context, before exhibiting two complementary methodologies to exploit this paradigm in user defined custom strategies. In particular, a novel formalism for **DSE** is introduced in order to exhibit how the functional programming paradigm can be used to build intelligible and concise exploration strategies. Chapter 6 exposes the experimental setup, including a software demonstrator integrating the defined methodologies and a benchmark of representative applications, and presents the results of the experimentations that were led. Chapter 7 concludes this manuscript, discussing both the proposed contributions and the perspectives of evolution.

In addition to those chapters, four appendixes are to be found at the end of this manuscript. Among them, Appendix A provides some insights about the usage of **Chisel**, the chosen **HCL** for this work, which will help the reader to have a better apprehension of the features that such language can bring to the world of hardware design.

3

State of the Art

This chapter outlines the related works about *Design Space Exploration* (**DSE**) on *Field-Programmable Gate Array* (**FPGA**). As the notion of **DSE** can refer to any process exploring variations of implementations, it can be applied at various levels of granularity, from basic operator implementations [REHS+16][GMX+21] to whole *Systems-on-chip* (**SoC**) [ACP04][CV10], and can even be applied for software/hardware co design [BTBB21].

Moreover, when it comes to digital design, **DSE** processes can be applied for both *Application-Specific Integrated Circuit* (**ASIC**) and **FPGA** circuits, and some previous works showed that an optimal **ASIC** solution may prove to be suboptimal when targeting a **FPGA** [LLS19].

In the context of this work, we will hence focus on **DSE** application at **kernel level**, meaning that we will consider **FPGA**-based implementations of more or less complex algorithms that fit on a single **FPGA** device.

We will start by exposing some popular tools leveraging **DSE** for **FPGA** based designs, and will consider how are design spaces exposed in the state of the art methodologies. We will then examine which metrics are used in **DSE** processes, and how relevant estimators are built and integrated in the exploration frameworks. Finally, we will discuss various exploration strategies, along with both their advantages and their limitations, before providing a synthesis of the approaches for **FPGA**-based **DSE** in the literature.

Table of contents

3.1 Overview of the Existing Tools 21
3.2 Design Space Exposition 22
3.2.1 Explicit Parametrization 22
3.2.2 Implicit Inferences 23
3.2.3 Mixing-up Approaches 24
3.2.4 Exposing an Explorable Design Space 25
3.3 Metric Definition and Estimations 26
3.3.1 Estimating Temporal and Spatial Concerns 26
3.3.2 Quality of Service Estimation 29
3.3.3 Estimating Other Metrics 30
3.3.4 Defining and Integrating Metrics 31
3.4 Exploration Strategies 31
3.4.1 Meta Heuristics 32
3.4.2 Dedicated Heuristics 33
3.4.3 Supervised Learning 34
3.4.4 Graph-Based Algorithms 35
3.4.5 Discussion on the Exploration Needs 35
3.5 Synthesis on the Existing Approaches 36

3.1 Overview of the Existing Tools

As an exhaustive listing of every *Design Space Exploration* (**DSE**) framework is both unfeasible and of limited utility, we will start by providing a brief review of the most popular tools aimed at easing the developers life, focusing on rising the abstraction of design processes with approaches such as *High Level Synthesis* (**HLS**) and *Domain Specific Languages* (**DSL**).

To begin with, Windh et *al.* [WMH+15] provide a rapid study of high level tools for reconfigurable computing. A particular focus is put on **HLS** initiatives, with two industrial tools, namely Xilinx Vivado HLS [Zha08][Xil19] and Altera OpenCL [Sin11], and two academic frameworks, LegUp [CCA+11] and ROCCC [VPNH10]. BlueSpec System Verilog [Nik08], another approach based on a simplified behavioural model — abstracting some of the difficulties of standard *Hardware Description Languages* (**HDL**) — is also introduced.

Whereas the **HLS** approach is a main trend toward higher productivity, some **DSL** based initiatives are also considered to close the gap between domain specialists — such as data scientists or signal processing experts — on one hand, and **FPGA** developers on the other. Kapre et *al.* [KB16] expose a brief survey of different **DSLs** and their application domains: while DFiant [PE17] focuses on dataflow based applications, Sano [San15] proposes a framework targeting parallel streaming architectures. Kristien et *al.* [KBSD19] expose how the `Lift` framework can be used for efficient design, leveraging functional patterns for compilation, while Spatial [KFP+18] exhibits **FPGA** specific patterns and constructs for accelerator generation.

All those initiatives demonstrates the variability of the approaches for efficient design, where the abstraction level, the parametrization, the application domain or the target technology are as many knobs that can be tuned to build an efficient design framework. Most of those approaches rely — more or less heavily — on **DSE** processes to help designers make expertise-based decisions, such as parallelism exhibition, interface definition or dataflow pipelining.

In order to provide an interesting analysis of the existing literature on **FPGA** based **DSE**, we thus chose to focus on three main concerns, which we identified as being key levers for efficient strategy definition:

- **design space exposition** — *i.e.* defining which implementation variations can be explored, and how to generate corresponding architectures
- **metric definition** — *i.e.* exposing metrics of interests for a given use-case, as well as how to build and integrate estimators
- **exploration algorithm** — *i.e.* describing how to scan the design space in a clever way, providing rapid yet accurate results

3.2 Design Space Exposition

When building a **DSE** framework, one of the main concerns is to define which variations are to be considered in the exploration, and how to generate the corresponding implementations from the initial description.

Various abstraction levels are also to be considered, ranging from high level programming languages such as python or C/C++, which can be used to allow software developers to design their own circuits, to *Register-Transfer Level* (**RTL**) languages, which enable hardware designers to efficiently control the built accelerators based on their expertise.

However, the input language and the abstraction level can also be considered from another point of view: the level of control over the implementation variations. The **DSE** methodologies can either rely on the explicitation of generation parameters, or on the automatic inference of the implementation variations — but most of the popular high level tools are based on a mix-up of those two options, allowing users to define some variations while inferring others, based on the standard design methodology (with potential optimizations linked to the application domain and/or the target board).

In this section, we will hence consider how the design variations are chosen in the different approaches, rather than their abstraction levels. Doing so allows to classify the **DSE** initiatives without introducing the classical distinction between **HLS**, **DSL** and other techniques, as the main difference when it comes to exploration is the way to expose the explored design space.

3.2.1 Explicit Parametrization

The basic idea of the explicit parametrization approach is to allow users to define explicit parameters in their code, in order to provide an explorable design space built over understandable variations. Such parameters can be application specific, for example matrix dimensions in a matrix multiplication kernel, or independent, with standard parameters such as the *Input/Output* (**IO**) bandwidth or the element type [FMR21a].

Application specific generators have been introduced in the literature to easily expose and explore design spaces, at various levels of granularity. While Rehman et *al.* [REHS+16] use a library of basic blocks and the composition of components to explore approximate multiplier implementations, Yiannacouras et *al.* [YSR07] introduce SPREE (Soft Processor Rapid Exploration Environment), a framework allowing to explore the variations of a processor by leveraging standard **CPU** notions such as *Functional Unit* (**FU**) implementation and dataflow pipelining. Other initiatives have been proposed for

domain specific **DSE**, such as the implementation of *Convolutional Neural Networks* (**CNN**), for example using different dataflow techniques as variations [PPP20]. We can thus state that developers may need to expose very specific parameters in their exploration flow, and the exploration frameworks should provide a way to do so.

To easily integrate the definition of parameters into the standard design flows, Paletti et *al.* [PCS21] provide **Dovado**, a **RTL**-based exploration framework leveraging **HDL** *generic* features to build the design space. However, their perspectives include supporting other languages with more convenient ways for parametrization, such as **Chisel** or other **HCLs**.

HCLs are recent initiatives focused on building parametrizable generators instead of use-case specific accelerators, using high level languages as entry points to leverage promising software paradigms in the hardware world. Among them, we can list **python** based initiatives such as **MyHDL** [JS15] or **PyMTL** [LZB14], **scala** based frameworks such as **Chisel** [BVR+12] and **SpinalHDL** [Pap17], and **Cλash**, an **Haskell** based project [BKK+10].

Several **Chisel** based initiatives are exploring the possibilities of both high level programming features and highly parametrizable constructors to provide interesting exploration features. Cook et *al.* [CTL17] proposed **Diplomacy**, a parameter negotiation framework to automatically select and propagate generation parameters of the **Rocket chip** processor[1] [AAB+16]. To go further, Bai et *al.* introduced the **BOOM-Explorer** [BSZ+21] to explore the *Berkeley Out-of-Order Machine* (**BOOM**) core[2], varying the generation parameters of the different pipeline stages in the dataflow. A more generic approach was also provided with **JackHammer** [SI15], an exploration framework for **Chisel** based designs. However, the initiative is not maintained anymore, and is claimed to be too specific for SHA-3, the target algorithm.

As one can observe here, exposing explicit parameters through **HCL** usage seems to be a promising way for an efficient **DSE** framework definition. Moreover, as **Chisel** is used as a basis for this work, we provide some useful insights and basics in Appendix A, that should help readers to be more familiar with the possibilities that this language offer.

3.2.2 Implicit Inferences

While a generator-based exploration allows users to define which variations are to be explored, it cannot be used in frameworks based on a change of programming paradigm, such as **HLS**. As **HLS** is based on translating an

[1] Rocket-chip is an open-source, in-order version of a **Chisel**-based RISC-V processor.
[2] **BOOM** is an out-of-order implementation of a **Chisel**-based RISC-V core.

algorithmic description to a hardware one, choices are to be taken to model the decisions that a hardware developer would take when doing the same task — *e.g.* the level of loop unrolling, the memory model or the input bit width. While those choices would usually require a lot of parameters and description to be efficient, standard **HLS** tools often automatically infer which transformations are to be triggered to provide an efficient solution.

To do so, the **HLS** flows iteratively select which transforms to perform over the *Intermediate Representation* (**IR**), optimizing a certain set of objective functions under constraints [PBMR14] — Ye et *al.* [YHC+21] even consider a multi-level **IR** to expose different optimizations depending on the abstraction level. Some **DSL** also use a similar approach for compilation: the `Lift` [KBSD19] frameworks expose implementation variations using rewrite rules over the original description, while Sano et *al.* [San15] exploit the inherent parallelism to generate different implementations to be explored.

A similar approach is also used in *Approximate Computing* (**AxC**) based exploration frameworks, as the actual implementation of used operators does not really matter. As a result, different approximations are considered to generate the implementations (named *variants* by Witschen et *al.* [WAGM+19]) and build the design space [MKC+20][BTBB21]. Other initiatives consider some optimizations that are possible due to the **AxC** inherent freedom, such as optimizing the data word lengths to reduce required hardware [HMS05] or reordering the floating-point operations to optimize latency [GWC16].

Those approaches thus bypass both explicit decisions and parameter definition that should be provided by user, and automatically infer how to transform the circuit representation to provide a best fit — which may result in non optimal designs (with respect to a manually tuned design), and does not allow to fully control generated circuits.

3.2.3 Mixing-up Approaches

However, when looking to the most popular **DSE** flows, we can remark that they are actually based on a mix-up of those two approaches.

To begin with, Schafer et *al.* [SW20] provide a comprehensive study of the **HLS** approaches for **DSE**, and formalize a notion that is key in the design space exposition process: **exploration knobs**. Knobs are design parameters exposed at some point in the process, that allow the users to consider the synthesis process as a *black box* — providing different knobs as inputs of the process resulting in a different **RTL** implementation at the end of it. They are classified in three different categories, that are mainly relevant for *Application-Specific Integrated Circuit* (**ASIC**) design, but can also be considered when targeting a **FPGA** implementation:

– *local synthesis directives*, which are used to control the local inferences of the **HLS** tool.

  They are often implemented using *pragmas* that are directly integrated in the entry code, in order to fix some parameters that would have been inferred by the tool otherwise — leveraging user expertise.

– *global synthesis directives*, which are used to specify some global parameters for the synthesis.

  Among them, we can find the scheduling algorithm to be used for compilation, the clock constraints or the *Finite State Machine* (**FSM**) encoding scheme.

– **FU** *constraints*, which defines the level of sharing of **FUs**, introducing a performance/area trade-off.

  Such knob is often not considered by **FPGA** targeting **HLS** tools, as sharing a **FU** on **FPGA** often requires more resources (mainly muxes) than implementing another computation unit.

The popular **HLS** tools mainly leverage *local synthesis directives*, as it allows to finely control generated hardware. However, defining such *pragmas* is often a tedious task, as their impact on the generated circuit is often difficult to understand and may vary depending on the compiler version, which requires an expertise about both hardware development and used **HLS** flow. Moreover, it is difficult to modify the base code, as reusing the behavioural description in another context will probably require to tune every *pragmas* again, to guide the exploration flow.

Another approach based on controllable yet automatic inferences as been introduced in **Spatial** [KFP+18], where automatic inferences for scheduling and pipelining coexist with user defined parameters for generation.

3.2.4 Exposing an Explorable Design Space

We remark that exposing an interesting design space to be explored — *i.e.* a design space that consider potentially optimal designs for the current use case while providing a structure enabling an efficient exploration — often requires the developers to bring their expertise using custom parameters, while some freedom can be left to the tool for simple optimizations in order to allow compact but usable descriptions.

In this way, we consider building an *Hardware Construction Framework* (**HCF**) based approach for **DSE**, to allow users to expose specific parameters for the exploration, while allowing automatic transforms — which should be easily configurable — to be performed before generation.

3.3 Metric Definition and Estimations

Another important concern when it comes to **DSE** is how to define the **goal** of the exploration. As stated by Schafer et *al.* [SW20], no generic exploration strategy can be defined as it depends on a lot of parameters such as the application specificities, the target, the functioning conditions or the use-case constraints and objectives. An exploration process is thus to be defined specifically for a given combination of those parameters, including the objective function(s) to be optimized through the flow — as a matter of fact, multiple objectives and/or constraints are often to be considered in such processes to provide usable solutions. Barone et *al.* [BTBB21] introduce the need for efficient *Multi-objective Optimization Problem* (**MOP**) solving — *i.e.* "finding, for some *decision variables*, a set of values satisfying *imposed constraints*, while optimizing a set of *objective functions*" [Osy85].

In this context, we hence aim at providing some insights about interesting metrics and corresponding estimation methodologies for **DSE**: in other terms, we want to discuss how two implementations can be compared for a given set of constraint(s) and objective(s). Two complementary notions will be considered in this section: **metric definition**, which corresponds to circuit properties that hardware developers consider when iterating over their designs, and **estimation methodologies**, which are the ways those metrics are estimated in the flow.

Actually, to be able to efficiently explore a design space, an **accuracy** *vs* **speed** trade-off is introduced in the process, as providing an accurate estimation of a metric may require long procedures from specific tools — *e.g.* vendor frameworks such as *vivado* or *quartus* — while quick estimation methodologies often result in approximate estimations. We will thus consider mainstream metrics for exploration process, as well as interesting methodologies for their estimations, in order to provide users with a way to comprehend and take the best of this trade-off in their **DSE** design.

3.3.1 Estimating Temporal and Spatial Concerns

In standard **FPGA** processes, developers mainly consider two concerns for validating a circuit — that it fits the target board, and that the temporal behaviour respects the performance constraints.

The temporal behaviour is mainly considered using **two metrics**: **functioning frequency**, which defines the clock cycle duration for the design, and **latency**, which is an abstract notion considering the number of cycles needed for a computation (*e.g.* in stream based applications such as *Finite*

Impulse Response Filter (**FIR Filter**), it can be defined as the number of cycles needed for an input sample to have an impact on the kernel output). It can also be defined by combining those two metrics, providing a global latency metric in second.

On the other hand, the spatial part is mainly divided in **four metrics** for **FPGA** design, corresponding to the basic elements of a **FPGA** structure:

- *Look-Up Tables* (**LUT**)
- *Flip Flops* (**FF**)
- *Digital Signal Processors* (**DSP**)
- embedded memories, such as *Block RAMs* (**BRAM**)

Estimations for both metrics are classically based on manual analysis of vendor synthesis or implementation reports, which can hence be used as the **reference value** to achieve through estimations, but is obtained after long processes. We will hence discuss different levels of estimations, from high abstraction level to **RTL** based methodology, to provide insights about speed, accuracy and usability of different methodologies.

Resource Estimation

To begin with, resource estimation is a key feature for efficient exploration, with various level approaches proposed the last decades.

As **RTL** approaches are easy to integrate in any design flow (the representation level being the entry point of most synthesis flows), **HDL** based methodologies have been proposed for resource estimation. They mainly rely on two complementary approaches: modelling steps that the synthesis is expected to take in order to provide realistic estimations [SJ08], and use characterisation of basic blocks and *Intellectual Property* (**IP**) cores [DSC08]. This characterization approach allows to adapt estimators to a given **FPGA** and synthesis flow by providing a target specific way to generate the basis of the estimation methodology — to do so, basic blocks of interests are identified and synthesized only once, the report being parsed to provide a characterized model of the component to be used each time a similar component is identified in the estimated circuit.

As for higher abstraction initiatives, **MATLAB** based approaches have been introduced for early estimation of resource from algorithmic description through analytical models of circuits [NHCB02][SHM+04] — however, as **HLS** initiatives mostly consider C/C++ implementations, approaches based on such model have been left aside for more integrated solutions.

Regarding **HLS** recent initiatives, different techniques have been used in order to provide quick feedback to the exploration flow, either based on analytical models — *e.g.* to estimate **DSP** usage for multiplication implementation [ALS15], **BRAM** usage [ZFS+20] or approximate arithmetic implementation [CGMVSH20] — or on statistical approaches to infer resource usage from a circuit model [MOG+13].

Some application specific initiatives have also been proposed, for example using a mathematical model to estimate the resource usage of a **FPGA**-based *Network-on-Chips* (**NoC**) through a learning model [FCB14].

Timing Estimation

When it comes to early timing estimation, two main approaches are to be considered: either try to estimate the functioning frequency, or fix it using synthesis directives, and use scheduling algorithms to provide a latency estimation.

While this first approach is closer to standard design flows, it is impracticable to estimate critical path delay at early stages of **FPGA** based flows, as most of the delay is actually induced by the **routing phase** — more than 60% of the delay [XK96] — that happens late in the process, is really difficult to model and is heavily target dependent.

As a result, most of the **HLS** methodologies rely on the second approach, which is more feasible, is a better fit due to the change of paradigm, and can use the literature on scheduling methods to provide efficient ways to use allocated hardware [PBMR14][WSL+20].

In contrast with such static approaches, recent initiatives based on dynamic execution have been proposed to extract sub traces from C/C++ runs, using a **Resource constrained List-Based Scheduler** for latency estimation [ZPL+16][SDR+21]. The profiling approach provides an interesting alternative to static analysis of the circuit, as the entry point is an executable description of the target algorithm [OLT+18].

Joint Estimation

While we considered separated estimations of both concerns until now, most solutions used joint estimations of both spatial and temporal dimensions, as they are heavily correlated.

When the design flow cannot be easily modelled by description analysis — as it is done in **HLS** tools — it remains difficult to estimate the timing accurately, and it is often necessary to run the full implementation flow to validate the timing of a design after **DSE** phase. Todman et *al.* [TL12] use an inner loop for quick iteration, and an outer, more time consuming loop for

estimation validation, while Paletti et *al.* [PCS21] leverage multiple features to the reduce synthesis time: syntheses are run on some implementations, and a statistical method, named **Nadaraya-Watson model**, is used to estimate both resource usage and timing considerations of the implementations near to the synthesized ones. They also use the incremental compilation feature from *vivado* to perform quicker synthesis of new points.

As for **HLS** based initiatives, they actually use the result of the scheduling algorithm to provide both resource usage (by considering the minimum amount needed for each operation primitive with an optimal scheduling) and latency estimation. However, such estimations are quite inaccurate with respect to post place and route results, and initiatives are taken to consider multi-fidelity metrics in exploration process, using fast **HLS** based estimation to select implementations to explore, and accurate synthesis results to actually find the best fit among selected variations [LC18]. Some **MATLAB** based approaches also use models to estimate both resource usage and timing [SHM+04], in a similar way to initiatives discussed in the previous sections, or profile based methods to extract metrics from **MATLAB** executions [BMJ02].

Other initiatives for joint estimation include *Machine Learning* (**ML**) methods — *e.g.* using transfer learning method to estimate resource and timing using prior knowledge from previous **DSE** runs [KC20] — and profile based approaches, where sub trace extraction is mixed with learning methods [ZPW+17] to improve the **Lin-analyzer** as introduced by Zhong et *al.* [ZPL+16]. Bannwart Perina et *al.* [BPSBB21] also aim at improving **Lin-analyzer**, providing a roofline model to perform efficient exploration.

As can be remarked, a lot of initiatives are taken to provide quick yet accurate way to estimate both resource usage and temporal behaviour of generated circuits early in the design flow.

3.3.2 Quality of Service Estimation

Most of design flows only consider introduced metrics, as they can be defined in a generic way and don't rely on considered application. However, in their study on history and perspectives of **HLS** based **DSE**, Schafer et *al.* [SW20] consider applying such methodologies to the **AxC** domain, by integrating *Quality of Service* (**QoS**) concerns as additional metrics in the flow.

Different approaches have been proposed in the **AxC** domain in order to build **QoS** based **DSE** frameworks — the first step being to define how to estimate the accuracy of a given circuit for a particular use case.

Two orthogonal approaches can be considered for **QoS** estimation: either use an analytic model [HMS05], or use empirical results, which can be based

on **RTL** based simulations [MKC+20] or higher level executions (*e.g.* in the context of **HLS** tools) [REHS+16][GWC16]. Recent initiatives have been taken to provide quicker yet accurate estimation models, using **ML** methods instead of long running simulations to predict **QoS** [AGMP21].

A notable feature provided by Manuel et *al.* [MKC+20] is the ability to define custom **QoS** metrics, as accuracy can be evaluated in different manners depending on the use case (*Peak Signal to Noise Ratio*, *Root Mean Squared Error*, average error, ...). By providing different models for error estimation, such approach thus enables users to tune the exploration tool for their particular use case, thus providing a more generic approach.

Other approaches can be considered to provide relevant metrics for **AxC**-based explorations — for example, Savino et *al.* [SPLDC19] consider the usage of the **Register Data Lifetime** metric to identify the critical regions in approximate circuits, and thus adapt the global effort to produce efficient yet accurate designs.

3.3.3 Estimating Other Metrics

As **DSE** processes are built to guide and mimic developer intuitions in their iterative workflow to achieve acceptable designs, only considering a fixed set of metrics is a real limitation, and various initiatives have been proposed to integrate more exotic metrics in exploration flows.

While Li et *al.* [LZPC15] expose an exploration flow to optimize application throughput, Siracusa et *al.* [SDR+21] provide a comprehensive model to consider memory usage in **DSE**.

Moreover, as limiting power consumption is a raising challenge for ecological considerations, multiple approaches have been provided to consider such concerns at exploration time. Deng et *al.* [DSC08] provide target specific estimation of resources and power usage for given **IP** cores, while Manuel et *al.* [MKC+20] use early power estimation tools from vendors [Int21][Xil21b] to guide exploration. Other approaches use profiling for power estimation [OLT+18], or **ML** based techniques [LZSZ20], in order to integrate energy considerations early in the exploration process.

Finally, initiatives have been proposed to model and study security impact of variations in **DSE CPU** based systems [AAPLP21][GSN21], to provide useful insights in an era where security concerns are critical.

However, those contributions mainly aim at integrating specific metrics in custom flows, instead of allowing users to define their needs in an integrated fashion, by providing a comprehensive *Application Programming Interface* (**API**) to expose and integrate custom defined metrics and corresponding estimation methodologies.

3.3.4 Defining and Integrating Metrics

Defining metrics of interest for a **DSE** process is a tedious task, as it depends on various parameters from the application domain, the target environment and even the positioning of the circuit on the market.

While generic metrics can be defined — based on spatial and temporal considerations — other specific metrics can be defined, such as **QoS**, power consumption or security, and initiatives are yet to be proposed to define a generic **DSE** framework which would allow developers to define both metrics of interests and estimation methodologies (including needed abstraction level for the estimation).

In this section, we have exposed a few use cases where exotic metrics are considered, without providing a comprehensive study of the possible needs for metric definition in the context of hardware design. However, based on the exposed needs, we can remark that depending on the objectives of the exploration, different flows are to be considered, which potentially requires acquiring new skills and specific knowledge each time.

3.4 Exploration Strategies

The last stage to provide efficient **DSE** methodologies is the definition of the exploration strategies, which are the algorithms used to browse the design space in an efficient way.

As the problem of finding a best fit or a set of best fits in a design space is a tedious task with exponential complexity — where performing a comprehensive study of the whole space is most of the time not tractable — Schafer et *al.* [SW20] propose an interesting taxonomy for exploration strategies (Fig. 3.1). This taxonomy outlines the multiplicity of approaches that have been taken to provide a solution to this problem, as well as the fact that it is not possible to propose a best strategy for **DSE** in any context, as application specificities, target particularities and execution environment are as many variables to take into account when providing a solution for a particular use case. Four different categories are considered: *meta-heuristics*, *dedicated heuristics*, *supervised learning* and *graph analysis based* approaches, relying either on synthesis results, model based analysis, or a combination of both. Moreover, Shathanaa et *al.* [SR18] propose a similar taxonomy, providing a set of different approaches for efficient exploration. As both surveys exhibit a covering analysis of the various approaches, we will only consider initiatives that seem promising here, and will not provide a comprehensive study of this wide domain of research.

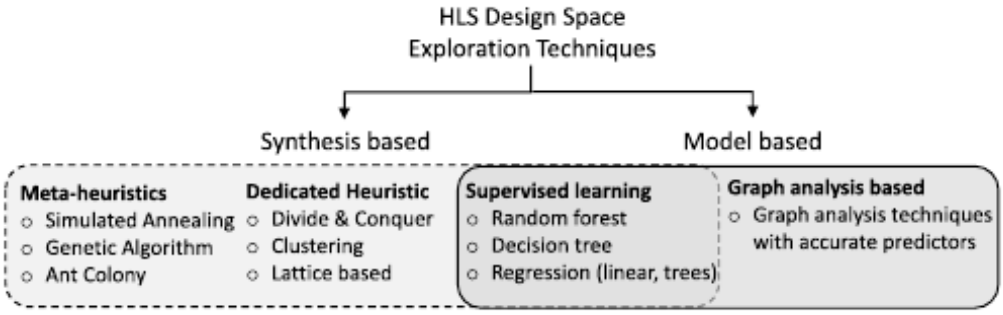


Figure 3.1: Taxonomy for HLS based DSE approaches, as proposed by Schafer et *al.* [SW20]

On the other hand, Barone et *al.* [BTBB21] expose a substantial literature for **AxC**-based exploration, showing that specific domains have their own optimization problems, and claim that we need generic **DSE** framework to address those specificities.

This section will thus expose a set of exploration initiatives based on introduced taxonomy from Schafer et *al.* [SW20] (Fig. 3.1), and provide some insights about usage and limitations of considered approaches.

3.4.1 Meta Heuristics

A first class of exploration algorithms is based on meta heuristics, which are problem-independent heuristics that can be used to efficiently solve *Multi-objective Optimization Problems* (**MOP**).

To begin with, *Genetic Algorithms* (**GA**) have been used to perform **MOP** solving in an efficient way [MKC+20]. **GA** are evolutionary algorithms that relies on performing biologically inspired operations to evolve population toward better solutions, by model variations of implementation as potential mutations in a population of architecture. Dovado [PCS21] exploit a **Non-Dominated Genetic Algorithm** from [SHM13] to solve the **DSE** optimization problem, while Barone et *al.* [BTBB21] provide an evolutionary search engine Bellerophon in their E-𝕀DE𝔸 automatic exploration framework. This framework itself is an amelioration of 𝕀DE𝔸 framework [BIM16], which was using a branch-and-bound approach to solve this optimization problem.

We can also consider stochastic processes such as **Bayesian optimization** for **MOP** solving, which can be used to optimize costly (*i.e.* long to estimate) functions in an efficient fashion. To do so, multi-variate spaces are modelled using **Gaussian processes** [LC16][LC18], and space exploration is

then performed without having to exhaustively explore the whole space. This is the approach used by **BOOM-Explorer** to efficiently propose a performant implementation of the **BOOM** core [BSZ+21]: **Gaussian processes** are used for initial set characterization (along with machine learning techniques), and **Bayesian optimization** is then used for Pareto optimization of considered design space.

Other meta heuristics for optimization have been exposed by Schafer et *al.* [SW20], where biological and natural behaviours are used as model for optimization process, as it is the case with simulated annealing, for example. Simulated annealing is a Monte Carlo based method aiming to model the process of metal annealing, consisting in heating and cooling the metal in order to achieve a better stability. It can be used to approximate a global optimum for an optimization problem, by allowing to avoid local optima which could be found by simpler strategies, and was used to solve **DSE** problem by various initiatives, including Witschen et *al.* [WAGM+19].

As meta heuristic usage for **MOP** solving is a very wide research domain, we will not list every initiative to provide efficient manners of solving such problem. However, one should know that many algorithms can be used, and should consider them when building a **DSE** framework.

3.4.2 Dedicated Heuristics

While meta-heuristics are problem-independent heuristics for optimization, dedicated heuristics are designed for specific problem solving. As **DSE** problem has been considered for decades now, many initiatives have proposed dedicated algorithms to perform efficient exploration, instead of trying to apply generic heuristics from other optimization domains.

While Prost-Boucle et *al.* [PBMR14] propose a greedy approach to synthesize C algorithms under constraints (selecting and applying transforms in a sequential way to find Pareto optima), other approaches focus on quick estimation methodologies to perform exhaustive search of the space [ZPW+17][BPSBB21]. Other approaches uses the structure of design space itself to perform more efficient optimizations: for example, **random sampling** of the space can be used, either to easily find Pareto optimum designs through neighbourhood exploration [YHC+21], or to identify **regions of interests** before performing sub region exploration through other processes such as search tree algorithms [AGMP21]. In order to provide efficient exploration of the space, one can also consider iterative processes in a greedy approach — *e.g.* iteratively selecting next directive to optimize in **HLS** based processes [SDR+21], or using gradient based algorithms to find local optimum [WAGM+19].

Moreover, dedicated heuristics can also be defined for application specific explorations, with a rising needs for *Convolutional Neural Network* (**CNN**) implementation exploration [MGAG16][PPP20]. This exhibits the specificities a domain can bring to the exploration problem, as well as the need for users to define specific explorations strategies for their use cases.

More complex initiatives leverage multi-fidelity estimation: Dong Liu et *al.* [DS16] perform **HLS** based estimation of the whole space, then perform pruning before syntheses, using **Rival Penalized Competitive Learning** to efficiently synthesize remaining implementations. In this way, they expose a need for a way to explicitly specify sequential exploration strategies to build complex strategy, while leveraging supervised learning methods to achieve efficient exploration.

3.4.3 Supervised Learning

As defined in exploration taxonomy from Figure 3.1, both meta-heuristics and dedicated heuristics are synthesis based processes, meaning that they only consider synthesis (or other estimators) results to guide exploration.

On the other hand, model based initiatives have been proposed in order to reduce the number of estimation processes to be run in an exploration process, thus accelerating the global flow. Supervised learning methods have thus emerged, leveraging synthesis generated knowledge to feed learning methods to be exploited later in the flow.

Different uses of supervised learning have been proposed, from leveraging knowledge from prior explorations to speed-up the current one [FKA+20], to fast simulated annealing using a decision tree [MS14].

Geng et *al.* [GMX+21] use **Graph Neural Processing** to explore adder implementation, while Nardi et *al.* [NKO19] also leverage prior knowledge in **Spatial** before using active learning under unknown feasibility constraints to provide Pareto approximation in their **DSE** engine. Liu et *al.* [LC13] study the usability of **Random-Forest** algorithm for **HLS** based **DSE**, providing a comprehensive study on eight different learning models. Moreover, Meng et *al.* [MAGK16] provide an interesting approach to re-think machine learning usage for **DSE**, by using such methods to prune the design space before running end-to-end exploration using synthesis processes.

Finally, **BOOM-Explorer** [BSZ+21] uses Gaussian process along with active learning and deep kernel learning functions for both initial set generation and design space characterization before employing Bayesian optimisation for exploration, showing that multiple approaches can be mixed to generate efficient exploration strategies.

3.4.4 Graph-Based Algorithms

The last class of **DSE** strategies in the considered taxonomy is purely based on modelling the whole synthesis flow instead of performing costly syntheses.

Such initiatives include **Lin-analyzer** [ZPL+16], which uses profiling to build a **Dynamic Data Dependence Graph** (DDDG) model, **COMBA** [ZFS+17][ZFS+20], a model based exploration framework for **HLS**, and **FlexCL** [WLZ17].

This approach does expose the needs for expressivity for users to be able to build purely analytic models for exploration in their **DSE** framework.

3.4.5 Discussion on the Exploration Needs

A plethora of different approaches have been proposed in more or less recent initiatives in order to improve **DSE** processes, and various heuristics and algorithms should be considered by users aiming at implementing efficient exploration flows.

Among those initiatives, some leverage multiple approaches by combining them for efficient exploration [DS16][BSZ+21], displaying a need for flexibility in the process of building an exploration strategy — the designers should be able to define and guide their exploration based on both their applicative and technological expertises.

In order to provide developers with such flexibility, a library of modular exploration passes should be proposed for efficient strategy building, allowing users to sequentially compose basic strategies to define more complex **DSE** operations in a given design space — *e.g.* leveraging quick estimation through learning methods for efficient pruning before running syntheses for more accurate results [MAGK16].

3.5 Synthesis on the Existing Approaches

Analysis of the literature

In this chapter, we have provided an analysis of the existing *Design Space Exploration* (**DSE**) strategies in the literature, focussing on three main aspects — the design space exposition, the metrics to be considered for both evaluation and comparison of different architectures, and the exploration strategies to be used to efficiently browse through a design space.

We have identified some initiatives that could be considered to provide performant exploration frameworks, including wide research domains such as *High Level Synthesis* (**HLS**) and *Domain Specific Languages* (**DSL**), and more recent approaches such as the *Hardware Construction Language* (**HCL**) paradigm. Among them, we put a particular focus on building flexible frameworks for strategy definition, such as the *Approximate Computing* (**AxC**) specific E-𝕀DE𝔸 [BTBB21], which aims at proposing a generic exploration framework that allows its users to define application and target-independent strategies in an extensible fashion.

In addition to this, we consider various taxonomies of exploration strategies, and among them the one proposed by Shathanaa et *al.* [SR18], where three strategy approaches are considered — namely hierarchical approach, iterative approach, and sequential approach. We hereby identify an opportunity to leverage the composition of basic exploration strategies in a functional fashion, in order to provide the users with sequential approaches in their exploration framework.

Considered approach and planned contributions

Based on this analysis of the **DSE** research field, we plan to build an efficient exploration tool to ease the life of the hardware developers. More specifically, we examine providing a generic *Field-Programmable Gate Array* (**FPGA**) based exploration framework which would allow users to expose and control:

- their own target and applicative domains
- their own design spaces
- their own metrics and estimation methodologies
- their own exploration strategies

To do so, the first step is hence to define how the design spaces are to be exposed in the framework. While we considered the usage of both **HLS** and **DSL**-based techniques for exposing explorable design spaces in this analysis, one can remark that such approaches have been widely explored in the past decades, resulting in the creation of multiple, industrially used frameworks. However, such tools are mainly based on the usage of architectural directives in the code to guide the exploration, and do not allow to finely control the generated accelerators as every choice that are not manually tuned by the developers are produced by automatic inference tools. On the other hand, some recent initiatives considered using *Register-Transfer Level* (**RTL**) languages to build generic exploration frameworks such as the one we are discussing here [PCS21]. Nevertheless, they are based on standard *Hardware Description Languages* (**HDL**), which are difficult to use and apprehend, and do not benefit from emerging paradigms to increase the productivity of the designers. To cope with those limitations, we hence consider using an **HCL** and its associated *Hardware Construction Framework* (**HCF**) to provide users with high level features for hardware development, while enabling to efficiently expose expertise-based design spaces.

As for the definition of the metrics of interest, as well as the corresponding estimation methodologies, we consider implementing the most commonly used methodologies as a *proof of concept*. However, some other approaches among the ones that have been presented in this chapter could be applied to build meaningful ways to evaluate and compare architectures, in order to provide users with both modularity and flexibility in their design processes. Indeed, some other initiatives cannot be applied in this context — *e.g.* considering an **HCL** as the entry point of the exploration flows is not compatible with the estimation methodologies that rely on an algorithmic description of the circuits behaviours, as it is the case for most of the **HLS**-based latency and resource estimators.

Finally, when it comes to the exploration strategies, most of the considered algorithms can be applied for **DSE** regardless of the exposition of the design space — except for graph-based algorithms, which also require an algorithmic description to be used. In this context, and even if we only consider implementing basic strategies in a first time, one could consider each of those approaches to build a configurable library of exploration algorithms, hence providing the users with a generic and modular tool.

Identified limitations

After defining the scope of this work, as well as the planned contributions, we identify some limitations about the proposed approach.

First of all, it will require its users to explicitly expose the design space, thus requiring expertise about both the algorithm and the target board. It is quite different from the **HLS** approach, for example, where the design space is essentially defined by the transformations performed by the **HLS** tool to generate the accelerators, that are quite transparent for the developers.

Moreover, our approach will require the users not only to define the design space to be explored (by exposing the generation parameters as architectural knobs), but also to describe the different generators of accelerators in a **RTL**-based language, which also requires a lot of time and expertise.

Last but not least, we will only consider sequential exploration strategies in this work, which could be a limitation for the algorithms relying on parallel and interacting steps for instance.

However, we propose to explore the opportunity of building a fully configurable exploration framework, that would leverage the features of recent **HCFs** to facilitate the life of hardware developers.

4

Building and Integrating Estimators

This chapter discusses the definition of metrics used in hardware development processes, as well as their integration in a *Hardware Construction Framework* (**HCF**). We then consider different types of metrics to be used for hardware development, as well as various ways to estimate them, and finally expose a generic *Application Programming Interface* (**API**) for users to build their own estimation methodologies.

Table of contents

4.1 Importance of Qualitative Estimations 40
4.1.1 Quality of the Estimators 40
4.1.2 Feedback Usage for Hardware Development 41
4.2 Resource and Timing Estimations 42
4.2.1 Intermediate Representation Usage 43
4.2.2 Basic Operator Characterization 44
4.2.3 Data Path Building 45
4.2.4 Limitations of the Approach 46
4.2.5 Macro Block Replacement 48
4.3 Quality of Service Estimation 52
4.3.1 Taxonomy of the Estimation Approaches 52
4.3.2 Analytical Estimations 53
4.3.3 Empirical Estimations 53
4.4 Integrating Metrics in a HCF 54
4.4.1 Exposing Multi-fidelity Estimators 54
4.4.2 Proposed Application Programming Interface . . . 56
4.5 Synthesis on the Estimation Methodologies . . . 57

4.1 Importance of Qualitative Estimations

In order to build efficient hardware designs under various constraints — such as resource usage, power consumption or exploitable throughput — hardware developers require quantitative estimations to make the best decision possible. However, design processes remain time-consuming tasks, and standard design methodologies rely on heavy computations — *e.g.* logic synthesis — to obtain exploitable feedback, resulting in long turnaround times. In this section, we will consider different metrics and the quality of estimators to retrieve them, then discuss the importance of qualitative estimators for hardware design, and more specifically for *Design Space Exploration* (**DSE**).

4.1.1 Quality of the Estimators

Various metrics can be used to define the quality of a hardware implementation, depending on the target circuit environment and the objectives of the developed accelerators. Among the most used metrics are resource usage, operating frequency, circuit latency or power consumption. However, they need to be adapted to the running environment of the design — *e.g.* resource usage can be defined as the area of silicon used when building *Application-Specific Integrated Circuits* (**ASIC**), but is most difficult to define for *Field-Programmable Gate Array* (**FPGA**) design, as multiple resources have to be considered due to the inherent heterogeneity of the boards (Fig. 2.2).

The quality of estimations needs to be considered to build efficient designs and exploration flows, as using poor estimators will more than probably result in a poor decision by the designer or by the exploration framework. However, we first need to define what is the quality of an estimator for a given metric, in order to quantify and compare multiple estimation techniques. Quality of an estimator can only be defined with respect to a reference value, *i.e.* the expected value for the metric being estimated. In the rest of this work, we will consider **synthesis results** as the reference value for both resource and frequency metrics, and consider estimators *Quality of Results* (**QoR**)[1] with respect to vendor specific tools such as *vivado* or *Quartus*.

In order to quantify quality estimators, we consider three aspects of an estimation methodology: **accuracy**, **faithfulness** and **speed**. **Accuracy** is defined as the proximity of the estimations to the expected values, while **faithfulness** is defined using the standard deviation of the estimations —

[1] In the context of this thesis, both *Quality of Results* (**QoR**) and *Quality of Service* (**QoS**) notions will be used. Please refer to the glossary for disambiguation purposes.

i.e. variation of estimators accuracy on different implementations.[2] As a result, estimators can have high faithfulness but low accuracy — *e.g.* always estimating twice as many resources allows to easily compare different architectures, but can result in wrong design decisions as estimations are far from real values. Finally, **speed** is defined with respect to the time needed to perform estimation.

In this context, estimation methodologies offer different trade-offs between speed, accuracy and faithfulness, and estimator usage depends on the design goal. For example, to validate that a circuit is suitable for a given constraint set, one may consider slow but accurate estimations. Conversely, performing **DSE** requires fast and faithful estimators, but accuracy is not a main concern as the focus is put on comparing hardware implementations.

We built Table 4.1 using our expertise and the literature introduced in Chapter 3. It presents different estimation methodologies, as well as the corresponding quality metrics, showing different trade-offs depending on the usage. Standard *Register-Transfer Level* (**RTL**) methodologies are used as a baseline (Fig. 2.3), since they are used to generate the reference values. Various works have explored rapid **RTL**-based estimations of both resource usage and operating frequency. However, critical paths are almost impossible to accurately estimate without running whole standard processes, therefore, **RTL**-based methodologies are not appropriate for such estimations. To cope with this challenge, *High Level Synthesis* (**HLS**) methodologies rely on fast resource estimations to allow quick exploration of the design space, but most tools use latency estimation and automatic scheduling rather than frequency estimation, in order to build accurate and controllable timing estimations. As for Spatial, it provides quick, reasonably faithful and accurate estimators, but the framework relies on a *Domain Specific Language* (**DSL**), and is hence not adaptable to every possible use case.

As a result, different metrics and estimation methodologies are to be considered for hardware development, depending on use case specificities — this aspect is known as **multi-fidelity metrics** usage and was already considered in the context of **HLS** exploration [LC18].

4.1.2 Feedback Usage for Hardware Development

After defining qualitative concerns about metric estimators, we consider the different uses of metric estimation for hardware development processes.

In standard **RTL** methodologies, iterating over produced designs is a time-consuming task as obtaining feedback about the developed circuit qual-

[2] Estimators **QoR** should thus consider both notions.

Entry level	Metric	Tools	Accuracy	Faithfulness	Speed
RTL	Resource, frequency	Synthesis (*vivado*, *Quartus*)	High	High	Slow
RTL	Resource	RTL estimation [SJ08][DSC08]	Low	Medium	Fast
HLS	Resource, latency	HLS tools [CCA+11][Xil19]	Low	Medium	Fast
DSL	Resource, frequency	Spatial [NKO19]	Medium	Medium	Fast

Table 4.1: Comparison of different estimators depending on abstraction level

ity can take up to days of complex tool usage such as synthesis or place and route. To increase productivity, quick estimators can be used to swiftly provide feedback to developers, guiding them toward acceptable solutions while reducing the time spent in heavy computations — however, significant time is still needed at the end of the process to validate that a design actually fits on the target board.

Moreover, qualitative estimators are primordial for efficient **DSE**, as exploration speed is a main challenge for such methodologies — actually, two levers can be considered to speed up exploration processes, namely exploration strategies and estimation methodologies. As exploration strategies will be considered in Chapter 5, this chapter will be dedicated to discussing the impact of estimation methodologies on **DSE** processes.

As such processes are based on architecture comparisons in order to determine the impact of implementation options on resulting designs, one of the main challenges is related to how architectures should be compared. Thereupon, quick but faithful estimators need to be considered in such processes to perform meaningful comparisons in a reduced amount of time.

4.2 Resource and Timing Estimations

Resource usage and timing concerns (*e.g.* operating frequency, circuit latency) are among the principal considerations when it comes to **FPGA** development, as target specific constraints are to be respected for a design to pass the validation process. However, as stated in Section 4.1, classical flows used to produce such metrics are time consuming, resulting in long iterations over generated designs to find a suitable solution.

This section hence considers building fast resource and timing estimators based on the *Hardware Construction Language* (**HCL**) paradigm to speed up its usage — more specifically, we consider a **Chisel** based implementation.

The aim is not to build state of the art estimators with particular focus on a good accuracy, but rather to integrate *proof of concept* estimators in an *Hardware Construction Framework* (**HCF**) and consider their usage as a way to increase hardware developer productivity.

In order to do so, we chose to implement a resource estimation methodology similar to the one introduced by Schumacher et *al.* [SJ08], based on operator characterization and an *Application Programming Interface* (**API**) allowing users to model compilation steps that the synthesis tool is expected to take in order to provide realistic estimations.

4.2.1 Intermediate Representation Usage

Using **Chisel** as an entry point for a hardware development flow, one still requires a **HCF** to perform circuit elaboration, optimizations and backend emission, allowing the generated design to be fed to any **RTL** based toolchain. As an **HCF** requires an *Intermediate Representation* (**IR**) to operate on — as does any compiler — *Flexible Intermediate Representation for RTL* (**FIRRTL**) [LIB16][IKL+17] was introduced along **Chisel**. It was primarily built as an initiative to use **HCLs** for parametrized hardware library building, abstracting technology specific knowledge from the **HCF** frontend and relying on further transforms and backend to transform target-independent **RTL** to technology specific **RTL** — allowing, for example, to use the same **Chisel** for both **ASIC** and **FPGA** targets. As a matter of fact, adapting software compilers structure — *i.e.* separating entry language (frontend), **IR** transforms and code generation (backend) — enables to reuse **HCL** code, instead of using *ad-hoc* scripts to replace specific structures in the entry code to target a particular technology, as is usually done in standard development processes. Such target specific transforms can now be enabled by operating directly on the **IR**, and modifying the target technology becomes as simple as changing the transforms and configuration used for generation. Moreover, as both **Chisel** and **FIRRTL** are based on **scala**, high level programming features can be used for both hardware generator description and transform definition, enhancing developers expressivity for hardware description.

Thereupon, to integrate estimators in an **HCF**, one must consider defining **IR** transforms as a way to operate on a given circuit and provide metric estimations in the process.

4.2.2 Basic Operator Characterization

Our first approach to build **FIRRTL** based estimators is based on a naive approach of digital circuits, considering each operator individually on data paths, adding individual metrics to build global estimators.

To begin with, we oppose costly *vs* non costly operators — based on our designer experience — in Table 4.2, considering impact on both resource usage and data paths traversal time.

Operator	Description	Impacting parameters	Considered for estimation
`ADD`	Adders	Operand widths	**yes**
`MULT`	Multipliers	Operand/result widths	**yes**
`BINOP`	Binary operations (`OR`, `AND`, `NOT`, ...)	Operand widths	**yes**
`MUX`	Multiplexers	Operand/Condition widths	**yes**
`DSHIFT`	Dynamic shifts	Shift/result widths	**yes**
`REGISTER`	Registers	Element width	**yes**
`MEMORY`	Memory primitives	Memory width and depth	**yes**
`SSHIFT`	Static shifts	Input width	no
`SELECT`	Bit selection among words	Result width	no
`PAD`	Word padding	Result width	no
`CAT`	Word concatenation	Result width	no
`CONVERT`	Type conversion	Operand type and width	no
`IO`	Input/output	Source/dest width	no
`CONST`	Constant definition	Constant value	no
`CONNECT`	Signal connection	Source/dest widths	no

Table 4.2: Operator impacts on both timing and resource usage estimations

To define non costly operators, we consider operations that are mostly reduced to rewiring of signals, as such action does not require particular resource usage but routing resources, and as wire traversal is considered non significant in this approach.

After identifying operators to consider for estimation, we use a pre-characterization based approach. To do so, we use vendor specific tools — such as the *vivado* synthesis tool for Xilinx **FPGAs** — to generate **reference values** for both resource usage and operator traversal time. Operators are characterized for a given set of different operand bit widths — for example, adders are characterized with operands on 1, 2, 4, ..., 256 bits — storing all the results in a **library file** (using the **JSON** format).[3] Once this is

[3]Different operand widths are not considered in this process to enable single parameter characterization, thus only maximum operand bit width is considered for estimation in this naive approach.

done, we can use this characterized library to estimate both resource usage and timing of a given operator, using maximum operand bit width in the **FIRRTL** representation as a parameter for estimation. The estimation process is then based on a simple statistical model: we retrieve the two nearest results — *i.e.* nearest maximum operand bit widths — for an operator from the library, and estimate both resource usage and data path cost using linear regression between those two results, for each considered metrics.

A first distinction is then to be made between resource and timing estimation: while we can estimate global resource usage by adding each individual operator estimation — for each metric considered (*e.g.* **LUTs**, **FFs**, **DSPs** and **BRAMs**) — we need further circuit analysis to provide interesting timing estimations.

Remark: characterization is not mandatory for `REGISTER` resource estimation as technological mapping is quite straightforward — a n bit register will only use n **FFs**. As for `MEMORY` primitive estimations, both width and depth should be considered for estimation, and characterization is not needed either for the same reason. We thus use an *ad-hoc* computation of memory resource, based on a simple estimation of the total amount of bit in the `MEMORY` primitive ($depth \times width$).

4.2.3 Data Path Building

Timing considerations for hardware design are mainly based on the notion of **critical path**. Critical path is defined as the longest path in the circuit — each path corresponding to a sequence of wires and transistors from a starting point to an ending point, the longest being the one where the total traversal time is the greatest. In the context of synchronous designs, both starting and ending points are mostly defined as memory primitives, being either registers (**FFs**) or more complex memory components (*e.g.* **BRAMs**). Circuit **IOs** are also to be considered for critical path definition, as they represent the interface with the external world, and may impact on the operating frequency. Maximal operating frequency is then defined as the inverse of the longest path traversal time, as operating at a higher frequency will result in some computation signals not being saved before new computations start.

In order to provide a maximum operating frequency estimation based on **FIRRTL**, we thus have to estimate traversal time of each possible path in the circuit, compare them, and expose the longest one(s) to users. We consider three different cases in the process: register to register paths, **IO** to register paths, and register to **IO** paths. It is important to note that the **FIRRTL** representation uses a hierarchical approach for describing a circuit — like do standard **HDLs** such as *verilog* or *VHDL* — which is used to describe

a complex circuit as a composition of simpler parts, the **modules**. The main circuit of a design process is denoted as being the **top level module**, and corresponds to the main **Chisel** class being compiled. One of the main challenges is hence to build every possible path, including **cross-module** ones — *i.e.* paths originating in a module and terminating in another one — which requires multiple passes over the circuit representation.

Once every possible path is built, we can finally use individual operator traversal time estimations built in Section 4.2.2. We consider a basic, addition based approach, where every estimation in a given path are added to estimate its total traversal time. We then compare each path traversal time, and provide users with feedback on timing concerns.

Figure 4.1 introduces an example of this estimation methodology usage on a simple circuit (Fig. 4.1b), when targeting a Xilinx *VC709* board. Sub results in Table 4.1c are computed using characterized values from Table 4.1a and simple linear regression — *e.g.* for `MULT` operator estimation on 24 bits, each value is computed using the average of values on both 16 and 32 bits. Then, for resource estimation, each individual estimation is added, while for critical path estimation, all paths are compared.

In this case, critical path goes from a 24 bits register to the 49 bits register through both `MULT` and `ADD` operations, resulting in a total traversal time of $0.695 + 1.512 + 2.702 = 4.909$ ns[4] — corresponding to a maximum operating frequency of 204 MHz.

4.2.4 Limitations of the Approach

While this first approach for both resource and timing estimation is simple to apprehend, it presents some heavy limitations.

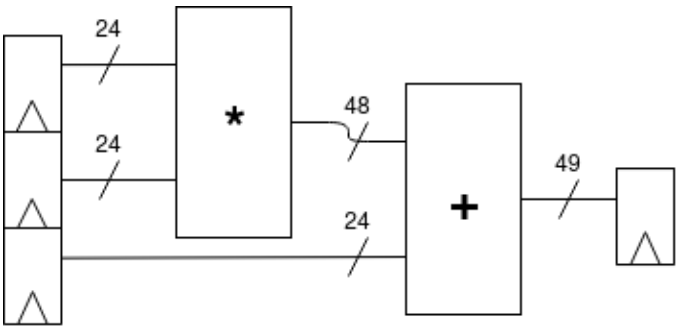
First of all, the operators are estimated using only the maximum operand bit width as parameter. Nonetheless, some operators might require some additional parameters, as shown in Table 4.2 — such as `MEMORY` primitives, `MUXes` or `DSHIFTs`.

Moreover, the heterogeneous structure inherent to **FPGAs** led developers to consider specific design patterns, in order to take advantage of available resources on a given target. For example, *Multiply and Accumulate* (**MAC**) operations are commonly used in domains such as signal or image processing, and can use **DSPs** — if available — to favourably replace **LUTs** and improve performance while reserving resources for other computations. Such usage cannot be expressed when considering operators as individual entities, and higher granularity should be enabled in the flow.

[4]The `REGISTER` traversal time is considered only once in the critical path.

Operator	Bit width	LUT	FF	DSP	BRAM	Path (ns)
ADD	16	16	0	0	0	1.296
	32	32	0	0	0	1.512
	64	64	0	0	0	1.944
MULT	16	0	0	1	0	1.228
	32	0	0	4	0	4.176
	64	0	0	16	0	5.439
REGISTER	16	0	16	0	0	0.695
	32	0	32	0	0	0.695
	64	0	64	0	0	0.695

(a) Characterized operator library for estimation (Xilinx *VC709* board)



(b) Example of a simple circuit to estimate

Operator	#	Bit width	LUT	FF	DSP	BRAM	Path (ns)
ADD	1	32	32	0	0	0	1.512
MULT	1	24	0	0	3	0	2.702
REGISTER	3	24	0	24	0	0	0.695
REGISTER	1	49	0	49	0	0	0.695
Circuit			32	121	3	0	4.909

(c) Simple estimation of resource usage and critical path for a simple circuit (Fig. 4.1b)

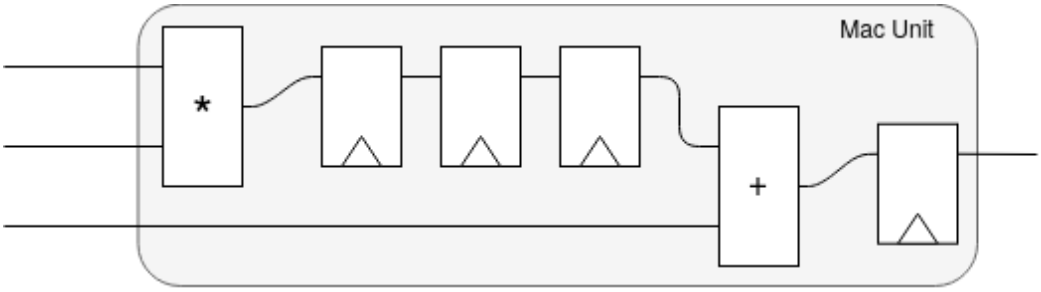
Figure 4.1: Basic estimation methodology based on individual characterization

Finally, data paths can heavily differ depending on many factors such as the operation, the operands or the target. For instance, adders are usually implemented using **carry adders**, each result bit being generated in a sequential way — the first bit only requiring a 2 bit logical operation, while the n bit require every results in $[\![0, n-1]\!]$ to compute the input carry. Let t_n be the traversal time of a n bit adder. We consider chaining two n bit adders

a_0 and a_1, with the output of a_0 being one of the operand of a_1. Our naive approach consider the total traversal time of the path as being $2 \times t_n = t_{2\times n}$, however computations in a_1 can begin after only t_1, as computing a_1 first bit only requires knowledge about the first bit of a_0 result. This means that most of the total traversal time of this path is actually absorbed in a pipeline fashion, with a total traversal time of t_{n+1} instead of $t_{2\times n}$.

All of these properties must be considered in order to build exploitable estimators — *i.e.* estimators that may help designers into taking advantageous decisions in the development process — as not considering them might result in erroneous feedback.

4.2.5 Macro Block Replacement



(a) `MAC` unit **macro block** example for pattern $(R^3)(+^1)(R^1)$

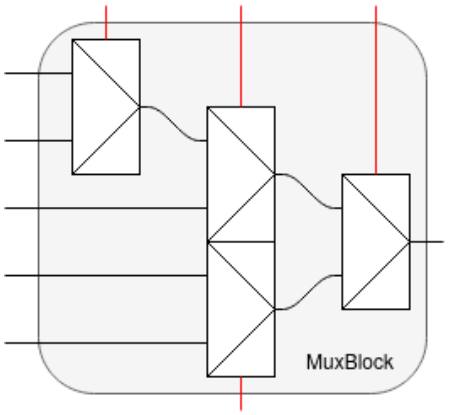
Name	Description	Type	Range
bit width	maximum input bit width	generation	$[\![1, 256]\!]$
outFactor	useful output bit width	configuration	$[\![1, 2]\!]$
mult register	number of `REGISTER` after `MULT`	configuration	$[\![0, 3]\!]$
add number	number of `ADD` in the `MAC` pattern	configuration	$[\![0, 1]\!]$
add register	number of `REGISTER` after `ADD`	configuration	$[\![0, 1]\!]$

(b) `MAC` unit **macro block** parameters w.r.t. Xilinx *VC709* specifications

Figure 4.2: `MAC` unit **macro block**

In order to improve this first approach, we now propose to consider more complex patterns for estimation, modelling steps that the synthesis tool is expected to take to pack operators [SJ08]. To keep the method generic, we define two main steps for pattern recognition, replacement and estimation, and consider a user trying to find and replace x different pattern $[\![p_0, ..., p_{x-1}]\!]$ in a **FIRRTL** representation of a circuit.

First of all, they need to define which pattern they are looking for, with respect to the **FIRRTL** representation. To do so, we build a *Directed Graph*

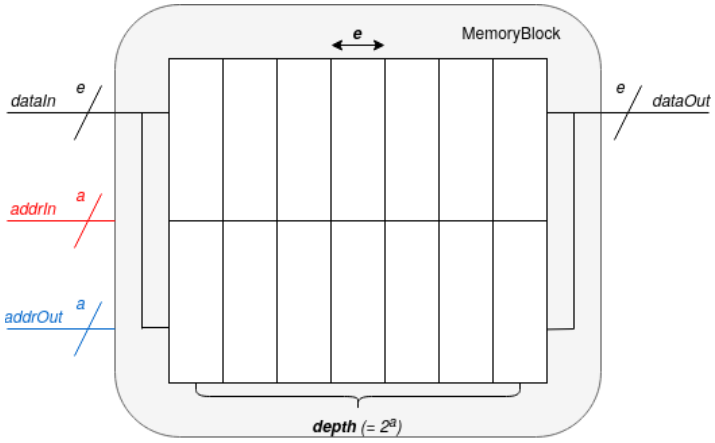


(a) `MUXBLOCK` example with 5 inputs and 4 conditions

Name	Description	Type	Range
bit width	maximum input bit width	generation	$[\![1, 256]\!]$
input number	number of inputs	configuration	$2^{i, i \in [\![1,8]\!]}$
condition number	number of conditions	configuration	$2^{i, i \in [\![0,8]\!]}$

(b) `MUXBLOCK` parameters w.r.t. Xilinx *VC709* specifications

Figure 4.3: `MUXBLOCK` **macro block**



(a) `MEMORYBLOCK` schematic

Name	Description	Type	Range
element width	memory element bit width	generation	$[\![1, 1024$
address number	number of addresses	generation	$2^{i, i \in [\![1,16]\!]}$
register number	number of register on address lines	configuration	$[\![0, 1]\!]$

(b) `MEMORYBLOCK` parameters w.r.t. Xilinx *VC709* specifications

Figure 4.4: `MEMORYBLOCK` **macro block**

(DG) from a circuit **FIRRTL** representation to operate on. We then developed a set of custom utility functions to scan the DG, recognize patterns and replace them by **macro blocks** in the graph. We thus search and replace each pattern $p_{i,i\in[\![0,x-1]\!]}$ in the DG representation in a sequential fashion, therefore capturing complex computation patterns. At the end of the process, an updated **FIRRTL** representation of the input circuit is produced in which all operators recognized as belonging to a particular **macro block** are replaced by the corresponding **macro block**. The HCF flow can then continue, and this representation can be fed into further **FIRRTL** transforms — *e.g.* for estimation purposes.

The second step is used to characterize the **macro blocks** in a way similar to the one described in Section 4.2.2. The user thus needs to provide a **Chisel** implementation generator for each pattern $p_{i,i\in[\![0,x-1]\!]}$, which is used to generate the **reference values** for both resource usage and traversal time, using vendor tools such as *vivado* syntheses. The generated values are then used to enhance the operator library with new parametrized **macro blocks**.

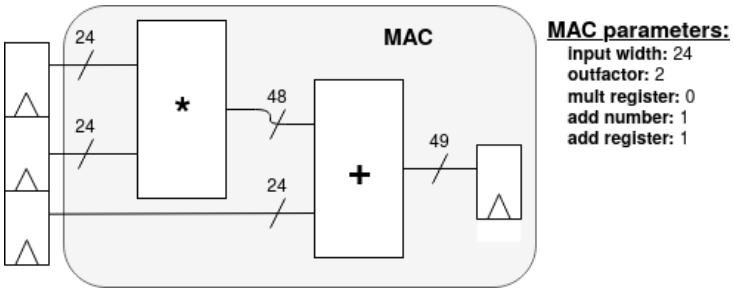
However, as stated in Section 4.2.2, the first approach only considered single parameter estimation based on operator input bit width. To cope with such limitations, we consider two types of parameters to enhance the estimation process — namely **configuration parameters** and **generation parameters**. The main difference between those two types is the value set width: **configuration parameters** can be explored exhaustively for library building while **generation parameters** would require too many runs and are thus only explored on a value subset for library building, before using linear regression for estimation as was done for bit widths in previous sections. Doing so enhances the library quality by adding a new entry for each possible configuration — *i.e.* each configuration in the cardinal product of **configuration parameters** — and each entry defines a regression based estimator for each considered metric (LUT, FF, DSP, BRAM and delay path), using **generation parameters** as arguments of estimation functions.

Three **macro blocks** were considered to the enhanced basic operator library described in the first approach: *Multiply and Accumulate* (MAC) units, complex `MUXBLOCK`s, and `MEMORYBLOCK`s. Figure 4.2 introduces the `MAC` unit macro blocks patterns and parameters, based on Xilinx *VC709* specifications. This pattern is defined to fill DSP blocks with a maximum amount of **FIRRTL** operations by packing them, in order to produce more accurate resource and timing estimations. This enables the `MAC` patterns to absorb every `MULT` operators and potential `ADD`/`REGISTER` operators in the same pattern, as the specifications state, for example, that Xilinx *Virtex 7* boards can absorb up to 4 `REGISTER`s and one `ADD` in a DSP block [Xil18]. Figure 4.3 presents a complex `MUXBLOCK` macro block, as the **FIRRTL** repre-

sentation system only considers *2-to-1* multiplexers in the emission process, resulting in erroneous estimations of *n-to-1* multiplexers, as they are represented as a chain of *2-to-1* `MUX` operations. In order to cope with those limitations, we thus absorb any `MUX` pattern with n inputs and m conditions, and use **macro block** characterization to improve resource estimation. Finally, `MEMORYBLOCK` **macro blocks** were introduced with respect to Table 4.2, as both element widths and depth must be considered for accurate estimation. Figure 4.4 introduces both `MEMORY` patterns and parameters, which are quite straightforward but enables a multi-variate estimation of the primitives using so defined characterized library methodology.

Operator/macro	Bit width	LUT	FF	DSP	BRAM	Path (ns)
`MAC` [2, 0, 1, 1]	16	0	0	1	0	1.000
	32	79	0	4	0	6.642

(a) Characterized macro library for estimation (Xilinx *VC709* board)



(b) Updated circuit from Figure 4.1b after macro recognition

Operator/macro	#	bit width	LUT	FF	DSP	BRAM	Path (ns)
`MAC` [2, 0, 1, 1]	1	24	40	0	3	0	3.821
`REGISTER`	3	24	0	24	0	0	0.695
Circuit			40	72	3	0	4.516

(c) Macro based estimation of resource usage and critical path for Fig. 4.5b circuit

Figure 4.5: Macro block based estimation methodology

Using the macro block replacement technique to improve our basic estimation methodology enables better estimations of complex patterns, and thus better estimations of **FIRRTL** circuits on **FPGA**. Figure 4.5 enhances the estimation methodology exposed in Figure 4.1 with macro block recognition and replacement. As macro blocks are parametrized by both **generation** and **configuration parameters**, we denote with `MACRO` $[p_0, ..., p_{n-1}]$ a **macro block** with n **configuration parameters** $p_{x,x\in[\![0,n-1]\!]}$ in Figures 4.5a and 4.5c. For example, the `MAC` [2, 0, 1, 1] pattern represents a `MAC` unit with an

outFactor of 2, no `REGISTER` after the `MULT` operation, one `ADD` operation and one `REGISTER` after the computation. As we can see, for a maximum *input bit width* of 16 bits, the whole pattern can be absorbed in only one **DSP**, while it would have taken one **DSP**, 32 **LUTs** and 33 **FFs** for the same pattern using only basic operator characterization. For the pattern introduced in Figure 4.5b, the estimations thus differ depending on the used methodology.

4.3 Quality of Service Estimation

Both timing and resources are key concerns when building hardware designs that target **FPGAs**, and should be considered in every design process. However, additional metrics should also be considered in specific use cases, as stated in Chapter 3, and estimation methodologies should allow to easily integrate new metrics in order to be as flexible as possible.

To demonstrate such flexibility, we chose to consider the *Quality of Service* (**QoS**) of circuits in our methodology, to demonstrate how additional estimators can be built and used in hardware design processes alongside resource and timing estimations. **QoS** is a main concern when it comes to particular domains such as *Approximate Computing* (**AxC**) or signal processing systems, as applications often require guarantees about the maximum error that may be introduced by approximations through computations. For example, usage of fixed point representations instead of IEEE-754 floating point numbers enable efficient hardware acceleration with no dedicated *Floating-Point Unit* (**FPU**), but results in divergences with respect to the software computational model — which often uses floating point representations as **CPUs** embed dedicated **FPU**. It is thus necessary to analyse the error introduced by such changes in the data representation to insure that acceleration does not provide erroneous results, in particular for critical systems.

4.3.1 Taxonomy of the Estimation Approaches

In order to build flexible estimators, we define a taxonomy for estimation methodologies and apply it to **QoS** estimation in Figure 4.6. An estimation can either be based on an analytic formula, an empirical approach, or both — *e.g.* when adding two fixed point numbers, one can consider the maximum error that may be generated, provide a statistical distribution of the two operands to derive an error model, or take an empirical approach and estimate the error by running multiple simulations. Each solution should be considered in this methodology to enable users to leverage their knowledge about the application domain and the target specificities — *e.g.* by making the decision to use **FPUs** if any are available on the target board.

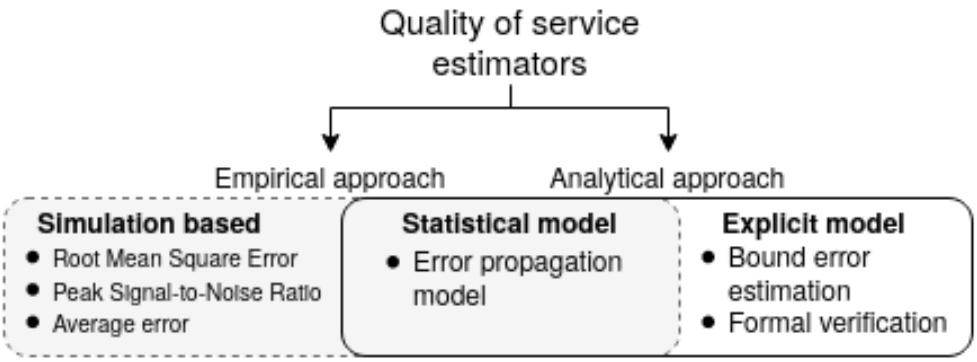


Figure 4.6: Taxonomy proposition for QoS estimators

4.3.2 Analytical Estimations

In order to provide analytical estimations for **QoS**, one should consider the specificities of both the data representation being used and the computations being performed. In most cases, a purely analytical approach is not desirable, as always considering the worst case for error propagation often results in inaccurate conclusions. For instance, knowing that an output on 8 bits with 4 bits of precision might have accumulated an error of $\pm 1/4$ does not bring any information on whether the implementation choices were good or not — although can be used for critical system validation. Rather, statistical error propagation models can be used as basis to chose the data representation in a design, as assumptions on the statistical distributions of operands can help reduce the error propagation deduced from the analytical study of the system, and thus provide more relevant and exploitable results.

In this context, we chose to enable users to define estimators at an analytical level, to derive **QoS** estimation from an expertise based analytical formula or error propagation models — or any model based on analytical study of the design. Such estimators can use hardware generator parameters in analytical formulas, and should be achieved early in the flow, as they should not require further information about the circuit (see Section 4.4.1).

4.3.3 Empirical Estimations

On the other hand, for non critical systems, an empirical approach may be used to provide **QoS** estimation, as occasional deviations for the expected behaviour should not have a noticeable impact on the design global **QoS** — *e.g.* occasional glitches in a video decoder are not as serious as erroneous results in the flight computer of plane.

In this context, the users should be able to build estimators using simulation processes to bring information about empirical results. We thus integrate simulators in the estimation methodology, and propose helpers to build statistical analysis of empirical values to build significant estimators such as average error, standard deviation or *Root-Mean-Square Error* (**RMSE**) (which can be normalized). As a trade-off exists between the accuracy of an empirical estimator and the number of simulations to run, we expose the number of simulation runs as a parameter of the estimation process.

After defining estimation methodologies for resource usage, critical paths and **QoS**, we then propose a generic way to integrate them in an **HCF**.

4.4 Integrating Metrics in a Hardware Construction Framework

The main goal of this contribution is not to integrate a particular set of estimators to the chosen **HCF**, but rather to expose a generic **API** for users to define their own metrics and estimators with respect to their particular use cases. To do so, we propose a generic model for both metric definition and their integration in the framework, and apply them to the integration of estimators defined in Sections 4.2 and 4.3.

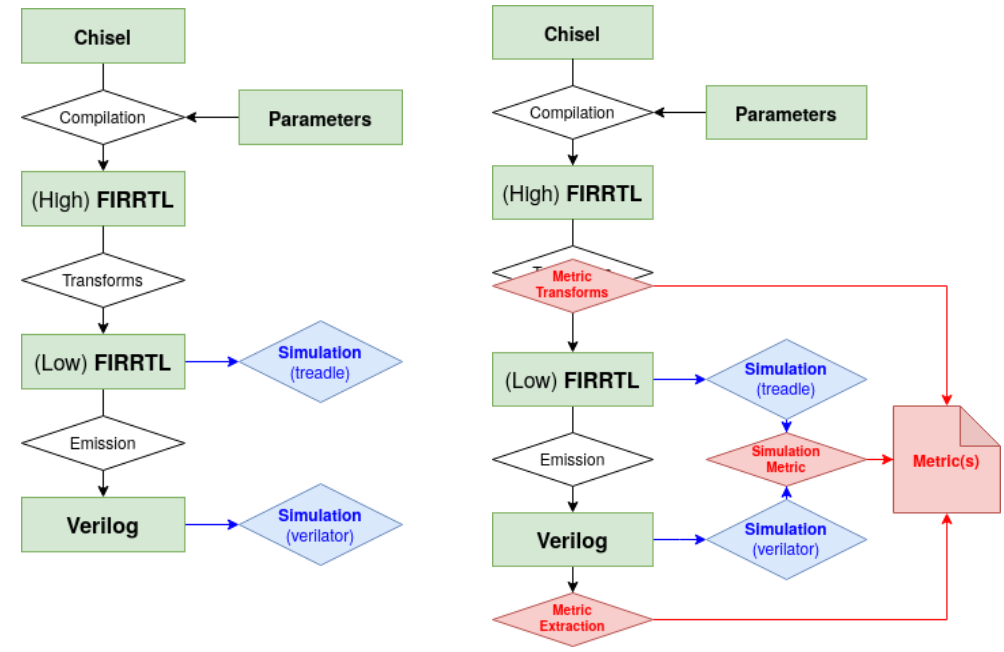
4.4.1 Exposing Multi-fidelity Estimators

In order to allow easy metric integration in the **Chisel** **HCF**, we start by defining abstraction levels where metric integration should be possible, as prior works exposed how **multi-fidelity metrics** can be useful for exploration [LC18] — for it can enable either quick or accurate feedback depending on the needs and the exploration step.

Figure 4.7 introduces the baseline **Chisel** development flow (Fig. 4.7a) and the proposed estimator integration steps (Fig. 4.7b). We choose to focus on three abstraction levels where estimators can be defined:

- Graph level — *i.e.* operating on **FIRRTL** circuit representation
- Simulation level — *i.e.* using simulation results for estimation
- Register-Transfer level — *i.e.* calling **RTL** based external tools

As we can see on Figure 4.7b, the entry point of the flow is not altered with respect to the standard **Chisel** process (Fig. 4.7a).



(a) Chisel baseline development flow

(b) Integration of multi-fidelity metric in chisel development flow

Figure 4.7: Proposition for Chisel flow enhancement

The **graph level** estimators are then built by adding some custom transforms to the ones that are used by **Chisel** to produce a low **FIRRTL** representation, while the **simulation level** estimators are build by exposing an interface with the different **simulation backends** that are integrated in **Chisel** and operate either at **FIRRTL** level or on the generated *verilog*. For readability purposes, we only expose the two main backends integrated in **Chisel**: *treadle*, which is specific for **Chisel** and directly operate on the low **FIRRTL** representation, and *verilator*, which is an open-source tool that uses C++ for efficient cycle-accurate simulations. However, other simulators can be integrated as well, as long as they can be interfaced with one of the different representations at stake in the flow. Finally, the **RTL** estimators are integrated through the usage of the file system, and can rely on any tool that can take the generated *verilog* descriptions as its entry points.

Using those three abstraction levels, one can adjust both estimation **QoR** and speed for their custom flows in a generic way — for example, both resource and timing estimators from Section 4.2 are integrated at graph level, while **QoS** estimators from Section 4.3 are integrated either at simulation level or at graph level. The reference values that are computed through the syntheses are integrated at **RTL** level, showing that a classical design flow can also be achieved through this **API**.

4.4.2 Proposed *Application Programming Interface*

In order to integrate our estimation methodology in **Chisel HCF**, we propose to use the **FIRRTL transform system**, which is used by the framework to enable optimization and circuit generation through **IR** scans, as shown in Figure 4.7. In fact, we define three levels for the integration:

- **pre-elaboration** level — which does not require **RTL** generation (*e.g.* analytical approach)
- **elaboration** level — which operates on **FIRRTL** representations (*e.g.* resource and timing estimation)
- **post-elaboration** level — which operates after **RTL** generation (*e.g.* simulation and syntheses)

Pre-elaboration estimations are performed directly on the generator, by extracting metric estimations from constructor parameters, while **post-elaboration** estimations are run at the end of the **HCF** process, and may leverage external tools such as synthesis suits or simulators. As for **elaboration** estimators, they are integrated directly in the **FIRRTL** flow, and uses the inner transform and annotation system — which consists in a sequence of transforms operating on a circuit representation, which can modify it and/or use an annotation system to forward information in the flow — to generate estimations from **IR** scans and forward it to the rest of the flow through annotation usage.

At the end of an estimation run, all the estimated metrics can be collected from the annotation system, and used by the developers to iterate on their designs until a satisfying solution is found.

4.5 Synthesis on Estimation Methodologies

In this chapter, we discussed the importance of qualitative estimations for hardware development processes, and put a particular focus on their application for *Design Space Exploration* (**DSE**).

We considered three different types of metric to be estimated — namely resource usage, maximum operating frequency and **QoS** — as well as how they can be estimated at different levels of fidelity. We proposed a simple estimation methodology based on an *Intermediate Representation* (**IR**) analysis for both spatial and temporal concerns, as well as an estimation taxonomy for *Quality of Service* (**QoS**) metrics, focusing on the *Approximate Computing* (**AxC**) domain.

We then introduced a generic *Application Programming Interface* (**API**) integrated in the **Chisel** *Hardware Construction Framework* (**HCF**) to allow users to define their own metrics and estimators. This interface is supposed to be comprehensive enough to allow any user to leverage the **FIRRTL** representation system to define their custom metrics.

As the *Quality of Results* (**QoR**) of estimators is a main focus when it comes to **DSE**, various metrics and estimators will be considered in the next section, which will focus on building an exploration framework integrated in a **HCF**.

5

Design Space Exploration Methodology

After defining the interest of relevant metrics and accurate estimators for hardware development processes, we focus on their usage for *Design Space Exploration* (**DSE**). This chapter outlines the specificities of *Hardware Construction Languages* (**HCL**) that can be used for efficient **DSE**, before proposing a methodology for **HCL**-based **DSE**. This methodology is based on high level programming features such as *Object-Oriented Programming* (**OOP**) or functional programming, which enable more expressivity for the developers. We also exhibit the limitations of this approach and discuss various solutions to improve the proposed methodology.

Table of contents

5.1 Defining DSE in a HCL context 60
5.1.1 Meta Design and Meta Exploration 60
5.1.2 Implementation Knobs and Parameters 61
5.2 Building Explorable Architectures 62
5.2.1 Designing Explorable Hardware Generators 62
5.2.2 Impact of the Implementation Parameters 63
5.2.3 Exposing an Expertised Based Design Space . . . 64
5.3 On Functional Programming for DSE 64
5.3.1 Mathematical Formalization 65
5.3.2 Basic Exploration Strategies 70
5.3.3 Complex Strategy Building 75
5.4 Discussions on the Proposed DSE Methodology 77
5.4.1 Limitations of the Approach 77
5.4.2 Synthesis on the Contributions 77

5.1 Defining Design Space Exploration using Hardware Construction Languages

This section aims at defining how *Hardware Construction Languages* (**HCL**) specificities can be leveraged for efficient *Design Space Exploration* (**DSE**), in contrast to other **DSE** methodologies based on various paradigms.

The **HCL** paradigm enables to describe hardware circuit generators instead of hardware circuits, allowing to fully control generated hardware at *Register-Transfer Level* (**RTL**). Doing so, we can define **DSE** methodologies which leverage the developers expertise to reduce the amount of implementations to explore, as some of them can easily be pruned considering prior knowledge on both algorithm and target. Moreover, high level features such as *Object-Oriented Programming* (**OOP**) or functional programming can be used to expose variations, exposing more complex implementation options — such as the definition of various computation units using functions as module parameters — when compared to standard **DSE** flows, such as *High Level Synthesis* (**HLS**) methodologies.

5.1.1 Meta Design and Meta Exploration

In order to propose an efficient **HCL**-based **DSE** methodology, we start by defining two main concepts: **meta design** and **meta exploration**.

Meta design is defined as the process of building an explorable design generator based on a prior analysis of both algorithm and target (Fig. 5.1). Leveraging this analysis, a developer can define relevant implementation variations — meaning that each possible implementation results from a choice from the user — and expose them at top level, directly in the module constructor. Doing so, the exploration process no more relies on tool inferences to generate such variations — *e.g.* as it is the case with **HLS** methodologies — but on controlled variations of the generators, resulting in a more meaningful design space to be explored. Moreover, using such methodologies enable a more intelligible approach of the exploration, as the implementation options are directly defined by users, enabling a better apprehension of their impacts on the generated design.

Figure 5.1: Meta design methodology

We then define **meta exploration** (Fig. 5.2) as a process exploiting a design generator — that was built using **meta design** — to explore its implementation variations, thus defining a **HCL**-based **DSE** methodology. In order to provide efficient exploration processes, this methodology allows the users to define custom strategies — after defining the design space — leveraging their expertise to guide the flow. In the next sections, we will consider a strategy to be composed of various metrics to be estimated (with respect to the considerations introduced in Chapter 4), as well as the different steps to browse the design space (see Section 5.3).

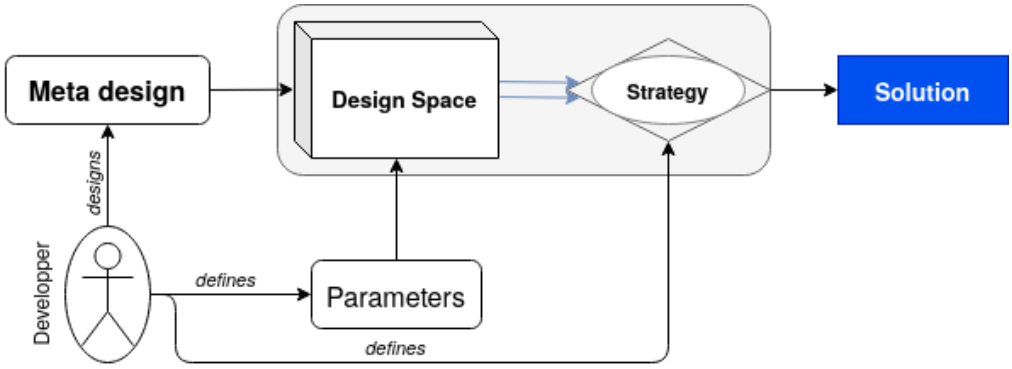


Figure 5.2: Meta exploration methodology

5.1.2 Implementation Knobs and Parameters

As seen in Section 3.2.3, **HLS** and *Domain Specific Languages* (**DSL**) based **DSE** methodologies are based on **knobs** — *i.e.* implementation options — which can be of three kinds: local attributes (as pragmas), global synthesis options and *Functional Units* (**FU**) options [SW20].

The exploration knobs are used by **DSE** tools to generate implementation variations, resulting in an explorable space of equivalent implementations of a same algorithm. It is thus comparable to the parameters of **meta design** based circuit generators as defined in Section 5.1.1.

However, the high level parameters exposed by the **meta design** methodology cannot express global synthesis options as it is done with knobs, and such variations must thus be considered in the **meta exploration** strategy, in order to offer comparable features. Nevertheless, both local attributes and **FU** options can be leveraged using design generator parameters, and will then be used in this work to define explorable design spaces.

5.2 Building Explorable Architectures

As defined in the previous section, the first step to define our **HCL**-based **DSE** methodology is to build an efficient **meta design** method.

5.2.1 Designing Explorable Hardware Generators

To design explorable hardware generators, one should begin by analysing both the algorithm and the target to exhibit meaningful implementation variations. It should result in a design space of relevant architectures to be explored, while leveraging the users expertise to expose both specific and non specific parameters.

For example, the *Input/Output* (**IO**) bandwidth is a non specific parameter, as it will have a significant impact on most of circuits, while the potential of parallelism or the number of a particular **FU** in a given architecture is application specific, as it will not have the same impact nor the same definition, depending on the targeted algorithm. The user expertise is thus required to produce a good analysis, and is primordial for this whole methodology to produce meaningful results.

After defining the exploration parameters, one needs to exhibit the possible values for each of them, in order to build the design space — *i.e.* a set of parameters which will be used in elaboration to give a particular implementation. For each parameter, users thus need to define a set of possible values, and each set is then considered in a **cartesian product** to build the resulting design space.

In order to integrate the **meta design** methodology in an *Hardware Construction Framework* (**HCF**), we propose to expose both parameters and their values at circuit top levels, in order to build intelligible flows and allow easy evolutions of the exploration processes.

```
0 class DotProduct(
1   @linear(6, 12)     dynamic: Int,
2   @linear(6, 12)     precision: Int,
3   @enum(16)          nElem: Int,
4   @linear(0, 4)      parallelism: Int
5 ) extends Module with Explorable
```

Listing 5.1: Exposing the dot product design space

In Listing 5.1, we use **Chisel** constructors to expose the design space of a **dot product** design, computing the dot product of two vectors a_n and b_n.[1]

We define four different parameters, namely the element dynamic and precision in bits (as the circuit only consider fixed point elements), the number of elements in each vector and the level of parallelism.

For each parameter, we only consider integer values for simplicity purpose, and define different set generators in Figure 5.3.

- `@byX(x, y)` $\Leftrightarrow$ $p \in [\![x, y]\!] \wedge p \equiv 0 \pmod{X}$
- `@linear(x, y)`[2] $\Leftrightarrow$ $p \in [\![x, y]\!]$
- `@enum(x0, ..., xn)` $\Leftrightarrow$ $p \in \{x_0, ..., x_n\} \wedge \forall i \in [\![0, n]\!], x_i \in \mathbb{N}$
- `@powX(x, y)` $\Leftrightarrow$ $p \in \mathbb{N} / p = X^z \wedge z \in [\![x, y]\!]$
- `@pow2(x, y)` $\Leftrightarrow$ $p \in \mathbb{N} / p = 2^z \wedge z \in [\![x, y]\!]$

Figure 5.3: Annotation based parameter generation

We use the **scala** annotation system to directly annotate the constructor parameters with value generators, thus embedding both parameters and their values in the top level module — *i.e.* the entry point of the **Chisel** flow.

For the **dot product** algorithm, we thus build a $7 \times 7 \times 1 \times 5 = 245$ wide design space to be explored, using the cartesian product of each set of possible values: both *dynamic* and *precision* can each take $|[\![6, 12]\!]| = 7$ different values, while *parallelism* can take 5, and *nElem* is fixed.

5.2.2 Impact of the Implementation Parameters

This approach allows to define the design space in an easy way by integrating it directly in the top level module, through its constructor.

However, leveraging the user expertise is not only about defining meaningful parameters and their values for the exploration, but also about bringing information about how the parameter variations may impact the exploration processes. For example, in the **dot product** implementations, we can state that the three first parameters — namely *dynamic*, *precision* and *nElem* — impact the algorithm *Quality of Service* (**QoS**), while the fourth one — *parallelism* — does not, as it only impacts the latency of the generated circuit.

In order to improve this first approach with respect to this observation, we thus need a way to define if a particular parameter variations has an

[1] Given $a_n = (a_0, ..., a_{n-1})$ and $b_n = (b_0, ..., b_{n-1})$, we compute $c = \sum_{i=0}^{n-1} a_i * b_i$.

[2] Equivalent to `@by1(x, y)`.

impact on a given metric. With such feature, a given exploration step that considers only the **QoS** of the designs will not consider the fourth dimension of the **dot product** generator, as every variation of the fourth parameter in the design space is to be considered equivalent with respect to the **QoS**. This enables to reduce the design space, dividing the number of implementations to explore by a factor 5 — the cardinality of the *parallelism* set of values — for an exhaustive exploration process.

5.2.3 Exposing an Expertise Based Design Space for Exploration

In order to enhance design space with informations on how chosen parameters impact given metrics, we add yet another annotation system to module constructors, in a way similar to the one introduced in Section 5.2.1. We thus add a `@qualityOfService` annotation, which bears the information that the annotated parameter does impact the **QoS** metric.

```
0 class DotProduct(
1   @qualityOfService @linear(6, 12)  dynamic: Int,
2   @qualityOfService @linear(6, 12)  precision: Int,
3   @qualityOfService @enum(16)       nElem: Int,
4                     @linear(0, 4)   parallelism: Int
5 ) extends Module with Explorable
```

Listing 5.2: Enhancing the dot product design space with quality of service concerns

For the dot product example, Listing 5.2 shows how new information can be brought about built design space: by specifying that only 3 of 4 parameters have an impact on circuit **QoS**, we then reduce the design space from **245** to **49 different implementations** to be explored for explorations focusing on such concerns.

5.3 On Functional Programming for Design Space Exploration

Using high level languages such as **scala** enables leveraging features such as **OOP** or **functional programming** for hardware generation. However, as such features are directly integrated in the **HCF**, they can also be used for side functionalities beside hardware elaboration.

In order to demonstrate how high level features can be used to improve hardware developers life, we build a **DSE** methodology based on functional programming.

5.3.1 Mathematical Formalization

In this section, we formalize how functional programming can be used to define **DSE** strategies, and expose a methodology based on this formalism.

Theoretical basis

Let $\mathcal{A}$ be an input vocabulary, which will be used to define metric names. We define $\mathcal{M}_\mathcal{A} = \mathcal{A} \times \mathbb{R}$ the set of **named metrics** with values in $\mathbb{R}$, representing any metric in an exploration process.

Metrics can be of two kinds: they either refer to the implementation parameters that were exposed through the **meta design** methodology, or they represent objective and constraint metrics that were generated during prior exploration steps. **Named metrics** are thus pairs of the form $(name, value)$, such as $(dynamic, 12.0)$ or $(frequency, 275.96)$.

Let $n \in \mathbb{N}^*$, we define a **configuration of order n** — *i.e.* a configuration relying on **n named parameters**[3] — as $x_n = \{x_0, ..., x_{n-1}\}$, with $x_i \in \mathcal{M}_\mathcal{A}$ and $i \in [\![0, n-1]\!]$. Each configuration stands for a different implementation variation, and we hence define a design space as being all the possible implementations for a given **meta design**.

We then define a **point of order (n, k)** as being an improved configuration, bearing both the configuration parameters x_i with $i \in [\![0, n-1]\!]$ and some generated metrics m_i with $i \in [\![0, k-1]\!]$. A point $p_{(n,k)}$ can then be defined as a vector of elements in $\mathcal{M}_\mathcal{A}$ which characterizes a given implementation:

$$p_{(n,k)} = \{\underbrace{x_0, ..., x_{n-1}}_{n \text{ parameters}}, \underbrace{m_0, ..., m_{k-1}}_{k \text{ metrics}}\} \tag{5.1}$$

Using such definition, we characterize a **design space of order n** as being a **set of points of order n**, denoted as being $s_n = \{p_{(n,\_)}\}$. We only consider the number of parameters for each configuration to define the dimensions of a design space, as the metrics do not represent dimensions but only information on the designs. For generalization purposes, we define $\mathbb{S}_n$ as being the set of all the possible exploration spaces s_n.

[3] **Named parameters** are a special case of **named metrics**, as they will represent not only metrics (*i.e.* design properties) but also coordinates in the **design spaces** that will be defined in this section.

We now aim at defining **design space exploration strategies** operating on so defined design spaces. We start by defining **cost functions** c as a way to generate new **named metrics** in $\mathcal{M}_\mathcal{A}$:

$$\begin{aligned} c : {\mathcal{M}_\mathcal{A}}^{n+k} &\to \mathcal{M}_\mathcal{A} \\ p_{(n,k)} &\mapsto c(p_{(n,k)}) \end{aligned} \tag{5.2}$$

Using so-built cost functions, we define **estimation transforms of order** θ, which are used to enhance a given design space with θ **new metrics**. Given θ **cost functions** c_i with $i \in [\![0, \theta - 1]\!]$, an estimation transform f_θ operating over points $p_{(n,k)}$ is defined as follows:

$$\begin{aligned} f_\theta : {\mathcal{M}_\mathcal{A}}^{n+k} &\to {\mathcal{M}_\mathcal{A}}^{n+k+\theta} \\ p_{(n,k)} &\mapsto p_{(n,k+\theta)} \end{aligned} \tag{5.3}$$

The resulting points are thus enhanced with θ new **named metrics**:

$$\begin{aligned} p_{(n,k+\theta)} &= \{x_0, ..., x_{n-1}, m_0, ..., m_{k-1}, c_0(p_{(n,k)}), ..., c_{\theta-1}(p_{(n,k)})\} \\ &= \{\underbrace{x_0, ..., x_{n-1}}_{n \text{ parameters}}, \underbrace{m_0, ..., m_{k-1}}_{k \text{ old metrics}}, \underbrace{m_k, ..., m_{k+\theta-1}}_{\theta \text{ new metrics}}\} \end{aligned} \tag{5.4}$$

We define a **morphism** as a modification of a **design space of order** n, which can be a way to sort, prune or even enhance it — and we call $\mathbb{M}_n$ the set of all the possible **morphisms** of order n. A **morphism** can also modify the dimensions of the **design space**, hence changing the number of **points** to be explored:

$$\begin{aligned} m_n : \mathbb{S}_n &\to \mathbb{S}_{n'} \\ s_n &\mapsto s'_{n'} \end{aligned} \tag{5.5}$$

We finally define how the **estimation transforms** are to be applied over a design space, before potentially modifying it through a **given morphism**.

Considering an estimation transform f_θ of order θ, a morphism μ_n of order n, and an input **design space** s_n of order n, we define a **transform application function** of order (n, θ) as being:

$$\begin{aligned} a_{(n,\theta)} : \mathbb{F}_\theta \times \mathbb{M}_n \times \mathbb{S}_n &\to \mathbb{S}_{n'} \\ (f_\theta, \mu_n, s_n) &\mapsto \mu_n(\{f_\theta(p_{(n,k)})\}) \text{ with } p_{(n,k)} \in s_n \end{aligned} \tag{5.6}$$

It is important to remark that one cannot presume about how the morphism and the estimation transform are applied over the input design space, and how they interact together. For example, the estimation transform can be

applied to all the points in the design space before applying a modification of its structure, or it can be applied through a more selective approach, for example using a gradient descent algorithm. In the following of this work, we will denote the set of all the possible **transform application functions** of order (n, θ) as $\mathbb{A}_{(n,\theta)}$.

For a more concise writing, we will use the **currying** notion, which is used in the **functional programming paradigm**. It refers to the action of converting a function with multiple arguments to a parametrized functions, which only takes one argument. For example, a function $f(a, b)$ can be converted to a set of functions $f(a)$, which can then be applied to the second argument, b.

We will hence convert our **transform application functions** to be simple functions operating over an input **design space**:

$$a_{(n,\theta)}(f_\theta, \mu_n, s_n) \Rightarrow a_{(n,\theta)}(f_\theta, \mu_n)(s_n) = \alpha_{(f_\theta,\mu_n)}(s_n) \tag{5.7}$$

Using all those constructs, we define an **exploration step** as a **function** operating over a **design space** by applying, given some **transform application function**, a set of **estimation transforms** to the **points** composing the space, before modifying its structure using a given **morphism**, and returning a new space enhanced with **new metrics**. More formally, we define an exploration step as being:

$$\begin{aligned} e : \mathbb{M}_{n'} \times \mathbb{A}_{(n,\theta)} \times \mathbb{S}_n &\to \mathbb{S}_{n''} \\ (m_{n'}, \alpha_{(f_\theta,\mu_n)}, s_n) &\mapsto m_{n'}(\alpha_{(f_\theta,\mu_n)}(s_n)) \end{aligned} \tag{5.8}$$

Doing so, we can use **currying** once again, to define **exploration steps** as being simple functions operating over **design spaces**:

$$e(m_{n'}, \alpha_{(f_\theta,\mu_n)}, s_n) \Rightarrow \epsilon_{(m_{n'},\alpha)}(s_n) \tag{5.9}$$

We finally use **functional programming** to compose basic strategies and build more complex ones, by applying n exploration strategies $[\![\epsilon_0, ..., \epsilon_{n-1}]\!]$ in a sequential way over an initial **design space**.

Basic functions for a concise definition of the exploration steps

In order to take the best of the functional programming paradigm for **DSE**, we provide some basic functions for a concise description of some popular programming patterns. For each introduced function, we will propose multiple equivalent descriptions — which are more or less compact and understandable — to help the user to understand how this emerging paradigm can be used for **DSE**.

First of all, we consider the **map-reduce** pattern, where a function is applied to every element in a given sequence, before performing a reduction to return only one value. For example, considering a vector of elements e_i, $i \in [\![0, k]\!]$, one can use this pattern to compute a sum of squares:[4]

$$\begin{aligned} sum &= e.map(x \Rightarrow x^2).reduce((a, b) \Rightarrow a + b) \\ &= e.map(\_^2).reduce(\_ + \_) \\ &= e.map(square).reduce(add) \\ \text{with } square(x) &= x^2 \text{ and } add(a, b) = a + b \end{aligned} \tag{5.10}$$

We will also use some simple operations over the collections, for example the `sortWith` function, which operates over a collection `col` and sort its elements by applying a comparison function. For example, if we want to sort a collection `col` of objects using a particular attribute `.value`, we can use:

$$\begin{aligned} newCol &= col.sortWith((a, b) \Rightarrow a.value \leq b.value) \\ &= col.sortWith(\_.value \leq \_.value) \\ &= col.sortWith(compare) \\ \text{with } compare(a, b) &= a.value \leq b.value \end{aligned} \tag{5.11}$$

Another useful operation is about filtering a collection, using a boolean function — *e.g.* to select only the elements for which the `.value` attribute is above a threshold min_{value}:

$$\begin{aligned} newCol &= col.filter(x \Rightarrow x.value > min_{value}) \\ &= col.filter(\_.value > min_{value}) \\ &= col.filter(func) \\ \text{with } func(x) &= x.value > min_{value} \end{aligned} \tag{5.12}$$

With respect to the formalism introduced in the previous section, we can remark that `sortWith`, `map` and `filter` can be defined as **morphisms**, if the collection `col` is a design space. As those constructs do not modify the number of **parameters** in the points they are operating on, they can even be considered as **endomorphisms** — *i.e.* morphisms from $\mathbb{S}_n$ to $\mathbb{S}_n$.

In the following section, we will then use a compact description of the different functions to be applied on the design spaces, to exhibit how the **functional programming paradigm** can help users to define concise yet intelligible exploration strategies.

[4] In this context, we will use some simplification coming for the functional programming paradigm, to replace implicit parameters (*e.g.* x) by a simple placeholder _, when there is no ambiguity for the compiler.

Application examples

As an example, we define **exhaustive strategies**, where the **transform application function** (from Eq. 5.7) consists in an exhaustive application of the **estimation transform** to all the **points** in the **design space**:

$$exhaustive_{(m_n,f_\alpha)}(s_n) = m_n(s_n.map(f_\alpha))) \tag{5.13}$$

It can be applied to define **exhaustive sort**, based on a comparison function *cmp* used to define an **order** over a **design space** s_n:

$$\begin{aligned} sort_{(f_\alpha,cmp)}(s_n) &= exhaustive_{(sortWith(cmp),f_\alpha)}(s_n) \\ &= s_n.map(f_k).sortWith(cmp) \end{aligned} \tag{5.14}$$

We also define **exhaustive pruning** of the space, using a pruning function f_{prune} to specify which **points** are to be left in the resulting design space:

$$\begin{aligned} prune_{(f_\alpha,f_{prune})}(s_n) &= exhaustive_{(filter(f_{prune}),f_\alpha)}(s_n) \\ &= s_n.map(f_\alpha).filter(f_{prune}) \end{aligned} \tag{5.15}$$

Based on those two basic strategies, we can define a more complex one, which uses both quick metric generation through **RTL** estimations of the resources, and accurate estimations through synthesis processes.

To do so, we define a first strategy ϵ_0 which we will refer too as the *preliminary pruning*, and a second one, ϵ_1, that will be referred as the *refinement*. We define *estim* as a **cost function** of order 1, based on a **RTL** estimation of the **LUT** resource usage, producing a metric of named LUT_{estim} for a given circuit. We also define *synth* as another **cost function** of order 1, based on an external synthesis tool call, producing a metric named LUT_{synth}, which is the reference value that the **LUT** estimation should approximate.

ϵ_0 is thus defined as follow, considering a threshold t_{max} which represents the maximum amount of **LUTs** acceptable in an implementation:

$$\begin{aligned} \epsilon_0(s_n) &= prune_{(estim,\_.LUT_{estim}<t_{max})}(s_n) \\ &= s_n.map(estim).filter(\_.LUT_{estim} < t_{max}) \end{aligned} \tag{5.16}$$

As for ϵ_1, it is defined as:

$$\begin{aligned} \epsilon_1(s_n) &= sort_{(synth,\_.LUT_{synth}>\_.LUT_{synt})}(s_n) \\ &= s_n.map(synth).sortWith(\_.LUT_{synth} > \_.LUT_{synth}) \end{aligned} \tag{5.17}$$

The global **exploration strategy** $\epsilon_\tau = \epsilon_0 \circ \epsilon_1$ can hence be defined as:

$$\begin{aligned} \epsilon_\tau(s_n) = s_n.&map(estim).filter(\_.LUT_{estim} < t_{max}) \\ &.map(synth).sortWith(\_.LUT_{synth} > \_.LUT_{synth}) \end{aligned} \tag{5.18}$$

This methodology hence enables to describe and compose **exploration strategies** in a functional way, as each can be considered as a simple function over a **design space**. Moreover, each strategy can itself be defined in a functional way, as it is mainly defined as a combination of various **estimation transforms** and some **morphisms** to be applied to a given **design space**.

5.3.2 Basic Exploration Strategies

To demonstrate how the defined methodology can be used by hardware developers to leverage their expertise and build intelligent **exploration strategies**, we propose to implement five **basic strategies** as a *proof of concept*:

1. exhaustive sort of the space (Eq. 5.14)
2. exhaustive pruning of the space (Eq. 5.15)
3. explicit dimension removal, based on the user knowledge
4. gradient descent search over the space (Algo. 5.1)
5. quick pruning through frontier approximation (Algo. 5.2)

The first two strategies have already been defined in the previous section, and will be used as basic elements for more complex strategy building — as they rely on the exhaustive application of a given function to the explored design space, their implementation can be trivially based on any high level language featuring functional programming.

The third strategy is a simple way for users to specify that for the remaining of an exploration process, a given dimension is no more useful as it will not impact the remaining steps, thus reducing the number of different implementations in the design space.

Finally, for strategy 4 and 5, we consider a design space implementation which provides a given set of functions to scan it. More specifically, we consider `map` and `filter` functions, which respectively enhance every point with a given number of metric(s), and prune some points by applying a boolean function. We also consider some order functions, such as *min* and *max* functions to find extremities of the space — *i.e.* combination of minimal/maximal parameters — as well as a neighbourhood definition.

To do so, we consider building two n dimensional discrete distances, namely $d_{\|.\|_1}$ and $d_{\|.\|_\infty}$, to formally define the neighbourhood of a given point.

For a given space s_n based on the **cartesian product** of n dimensions, we consider all the possible value sets δ_i for each dimension $i \in [\![0, n-1]\!]$. We begin by building an **indexation function** ϕ, which enables, for each parameter x_i ($i \in [\![0, n-1]\!]$) of a point $p \in s_n$, to retrieve its position with respect to all the possible values for the dimension i — *i.e.* the values of δ_i.

As an example is always better to understand, we consider a simple design space of order 1, with 5 different points $\{a, b, c, d, e\}$ in it. We only consider the **configuration parameters** of those points, resulting in a vector *space* of elements in $\mathbb{R}$:

$$space = \{\underbrace{1.0}_{a_0}, \underbrace{2.0}_{b_0}, \underbrace{4.0}_{c_0}, \underbrace{8.0}_{d_0}, \underbrace{16.0}_{e_0}\} \tag{5.19}$$

In this vector, we can remark that the distance between two consecutive parameters is growing exponentially, which makes it difficult to state that both the pairs (a, b) and (d, e) are direct neighbours. To cope with this problem, we define a vector $space'$, which corresponds to the position of the parameter x_0 in the sets of all the possible values *space* (which is actually the set δ_0 as defined earlier, as it is the set of all the possible values that the **first parameter** of each point can take), for any p in $\{a, b, c, d, e\}$:

$$space' = \{\underbrace{0}_{a'_0}, \underbrace{1}_{b'_0}, \underbrace{2}_{c'_0}, \underbrace{3}_{d'_0}, \underbrace{4}_{e'_0}\} \tag{5.20}$$

We hence define a function which maps a point p to a set of coordinates $\{x'_i\}$ ($i \in [\![0, n-1]\!]$) with values in $\mathbb{N}^n$, considering the sets of all the possible values for each dimension, δ_i, for $i \in [\![0, n-1]\!]$. More formally, we define this function ϕ as being:

$$\begin{array}{rccc} \phi: & \mathbb{S}_n & \rightarrow & \mathbb{N}^n \\ & \{x_0, ..., x_{n-1}, m_0, ..., m_{k-1}\} & \mapsto & \{x'_0, \ ..., x'_{n-1}\} \\ & \text{with } x'_i = k \ \text{ such that the } k^{st} \text{ value of } \delta_i == x_i & & \end{array}$$

In simpler words, we consider the position of each parameter x_i ($i \in [\![0, n-1]\!]$) of a point p in the sets of the possible values that it could have in s_n — we thus need to consider the whole space to define such function. For any point $p \in s_n$ we hence define its coordinates as being $\phi(p) = \{\phi(p)_0, ..., \phi(p)_{n-1}\}$.

Based on those new coordinates, we define both distances as follows, by considering two points $(p, q) \in s_n$:

$$d_{\|.\|_1}(p, q) = \sum_{k=0}^{n-1} |\phi(p)_k - \phi(q)_k| \tag{5.21}$$

$$d_{\|.\|_\infty}(p, q) = max(|\phi(p)_k - \phi(q)_k|)_{k \in [\![0, n-1]\!]} \tag{5.22}$$

Using those distances, we define the neighbourhood of point $p \in s_n$ — called $\mathcal{N}_{\|.\|_x,y}(p, s_n)$ — as the set of points $q \in s_n$ for which $d_{\|.\|_x}(p, q) \leq y$, for a given norm $\|.\|_x$. In both Algo. 5.1 and 5.2, we define $s_n.getNeighbours(p, \|.\|_x, y)$ as being the function to compute $\mathcal{N}_{\|.\|_x,y}(p, s_n)$.

Strategy 4. is based on a simple gradient descent algorithm as introduced in Algo. 5.1, where the search is performed by sequentially exploring all the neighbour points of the current optimum until a local optimum is found — it is considered as being the global optimum. This strategy is similar to the **Hill Climbing** algorithm that was used by Witschen et *al.* [WAGM+19], and can be leveraged when one is confident in the growth of the cost metric(s) with the dimensions of the problem (Hypothesis 5.1).

Hypothesis 5.1. *The cost metric(s) to be optimized are growing with respect to every dimension of the design space, until the constraints are violated.*

Assuming such postulate — which can easily be stated for some use cases, based on the user expertise — any local optimum can be considered global as well, thus finding a global optimum can be achieved without applying the cost function to the whole design space. However, the explored design space may already have been pruned before applying this strategy, or the hypothesis may be locally erroneous — *e.g.* for technological reasons, a local optimum can result from a transfer from **LUTs** to **DSPs** — and this strategy may not converge toward a global optimum. Nevertheless, it can still be leveraged to find an acceptable solution in a reduced amount of time.

As for Strategy 5., it is based on Hypothesis 5.2 and is introduced in Algo. 5.2. It is similar to a Pareto approximation approach proposed by Ye et *al.* [YHC+21], which iteratively uses space sampling to find some Pareto optimal points before exploring their neighbourhoods to approximate the frontier. We define the `isOnFrontier` method as follows, for a point $p \in s_n$ and a pruning function f — a point is considered to be **on frontier** *if and only if* it is not pruned and at least one point in its direct neighbourhood (as of the meaning of $\mathcal{N}_{\|.\|_1,1}$) is pruned. In other word, a point is on the frontier only if it is not filtered by the pruning function, but is in contact with a point to be removed.

$$isOnFrontier(p) \Leftrightarrow !f(p) \wedge \exists q \in \mathcal{N}_{\|.\|_1,1}(p, s_n)/f(q) \tag{5.23}$$

Hypothesis 5.2. *The pruning function partitions the space in two closed sets of implementations.*

Algorithm 5.1 Gradient descent algorithm

```
1: Input
2:     S    design space to explore
3:     f    cost function to sort S
4:     {x}  optional starting point for the descent
5: Output
6:     S'   sorted (and pruned) design space
7: procedure GRADIENT(S : Space, f : Point ⇒ Double, {x : Point})
8:     S' ← ∅                          // the result set is empty at first
9:     // either use x as starting point, or the head of S
10:    (current, cost) ← x ? (x, f(x)) : (S[0], f(S[0]))
11:    // iterate until a local optimum is found
12:    while True do    // the space is finite; an optimum exists
13:        neighbours ← S.getNeighbours(current, ||.||_1, 1)
14:        costs ← neighbours.map(f)    // apply f to all neighbours
15:        index ← indexWhere(costs.max)  // select best neighbour
16:        S' ← S' + neighbours
17:        // a neighbour is better than the current implem.
18:        if costs[index] > cost then
19:            // update current and cost with best neighbour
20:            (current, cost) ← (neighbours[index], costs[index])
21:        else
22:            // return sorted resulting space with respect to f
23:            return S'.sort
24:        end if
25:    end while
26: end procedure
```

Assuming that a single frontier separates pruned and non pruned implementations in the given space, one may prune it by applying the pruning function to only a fraction of the points, resulting in a faster convergence. To do so, the first step is to identify a first point on the frontier, which is done by the START procedure, using a simple assumption: if a frontier exist, and if Hypothesis 5.2 is true, then the frontier crosses the diagonal subspace composed of points ranging from the minimal to the maximal configuration (with respect to the implementation **parameters**). After identifying this first point, we iteratively build the frontier using the FRONTIER procedure, which uses neighbourhood exploration to build it step by step, as we know that the frontier is continuous (from the same hypothesis). The last step is then to update the design space (using the UPDATE procedure), only to keep the points that are "above" the frontier.

Algorithm 5.2 Quick pruning algorithm

```
1: Input
2:     S    design space to explore
3:     f    pruning function to discriminate space
4: Output
5:     S'   pruned design space
6: procedure QuickPruning(S : Space, f : Point ⇒ Boolean)
7:     // try to find a starting point on the frontier
8:     procedure Start
9:         // explore a sub space to find the starting point
10:        // (a diagonal between the extrema)
11:        diag ← S.getDiagonal(S.min, S.max)
12:        // select the first non pruned point on the diag.
13:        p ← diag.filter(!f)[0]
14:        // if a frontier exists, it crosses this diag.
15:        // either directly, or by neighbourhood
16:        if isOnFrontier(p) then
17:            return p
18:        else
19:            return S.getNeighbours(p, ‖.‖∞, 1).filter(!f)[0]
20:        end if
21:    end procedure
22:    // iteratively build the frontier
23:    procedure Frontier(p : Point)
24:        (currents, frontier) ← ([p], [p])
25:        while !currents.isEmpty do
26:            // explore neighbourhoods to find frontier points
27:            n ← currents.flatMap(S.getNeighbours(_, ‖.‖∞, 1)).filter(!f)
28:            onFrontier ← n.filter(isOnFrontier) − frontier
29:            frontier ← onFrontier + frontier
30:            // update with the limits of the actual frontier
31:            currents ← onFrontier
32:        end while
33:        return frontier                // a frontier has been found
34:    end procedure
35:    procedure Update(frontier : [Point])
36:        // select only the points above the frontier
37:        return S.filter(isAbove(frontier))
38:    end procedure
39:    return Update(Frontier(Start))
40: end procedure
```

These five strategies will be used as a basis for building more complex strategies in the following of this thesis. Moreover, there are built considering parallelism, in order to define efficient and adaptable strategies which can easily be integrated in an exploration framework.

Other strategies may be defined by the users — *e.g.* to build application specific space traversal order, using neighbourhood constructions — meaning that a framework implementing this methodology should allow an easy integration of new basic exploration steps, to provide a library of functions operating over design spaces. Such library could be used to define custom exploration strategies, by composing different steps in a iterative way.

5.3.3 Complex Strategy Building

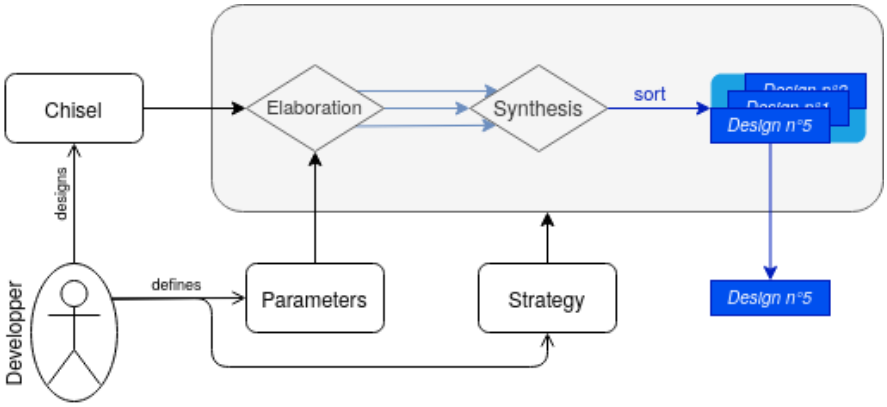
After defining some basic exploration functions, we can now expose more complex strategies using the features introduced in the previous section, and Figure 5.4 introduces three strategy examples using the defined constructs.

Figure 5.4a is a simple application of the `sort` strategy (Eq. 5.14). As it can be seen, the strategy is quite simple as it relies on a single step, where every possible implementation is synthesized before the results are analysed, compared and sorted to indicate the best solution for the given use case.

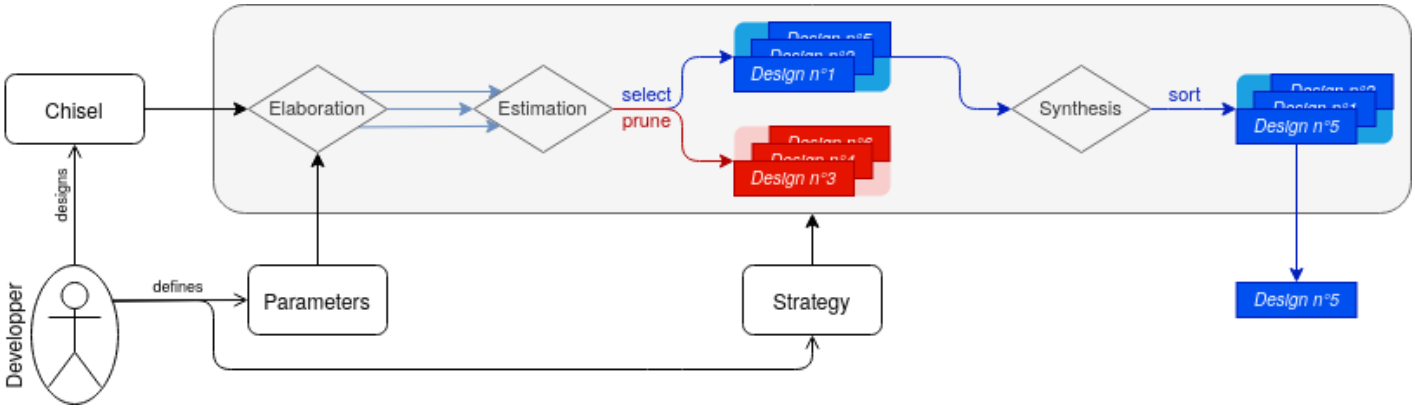
As for Figure 5.4b, it illustrates the strategy ϵ_τ (Eq. 5.18). We here use a two-step approach, with a preliminary pruning of the design space — where the resource usage of each implementation is estimated, and a user-defined boolean function is used to partition the space between fitting and non fitting designs — to reduce the number of syntheses to run, before applying these accurate but long processes on the remaining solutions to select the best one.

Finally, Figure 5.4c demonstrates how leveraging basic exploration steps can be used to build a more clever strategy. After performing a preliminary pruning and sorting of the design space using **RTL** estimations — *i.e.* applying both `sort` (Eq. 5.14) and `pruning` (Eq. 5.14) strategies — we use a gradient descent algorithm (Algo. 5.1) to run a minimal amount of synthesis process and find a local optimum. The starting point of this last step is the widest implementation that still fits in the remaining space after the pruning step, in order to speed-up the convergence of this greedy approach.

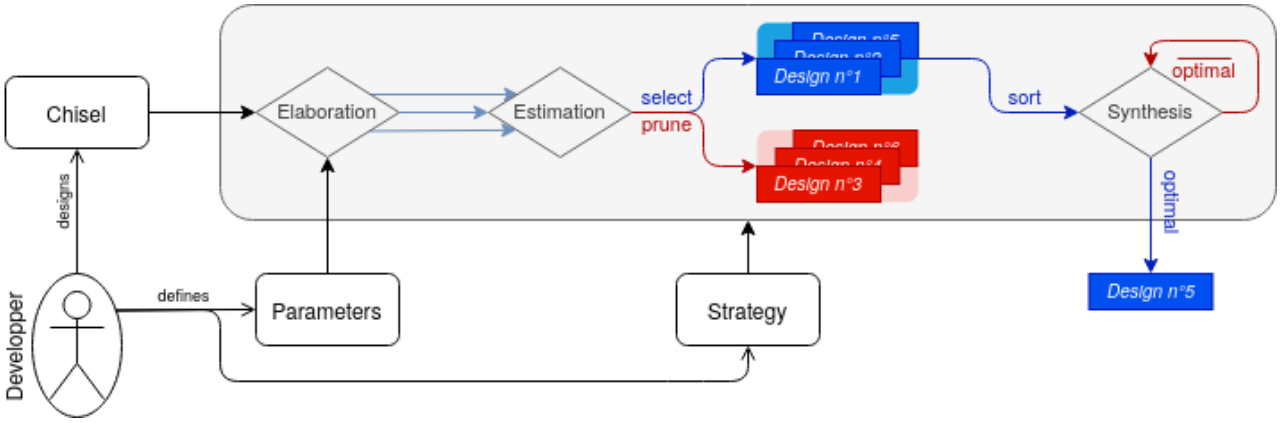
For those three strategies, the user needs to define the **Chisel** generator, and instrument it by annotating its constructor in order to define an explorable design space — which corresponds to the **meta design** methodology. They then need to specify how to browse the design space, by composing basic exploration steps in a functional way: it is the **meta exploration** methodology, which enables to build complex exploration strategies by composing simple ones.



(a) Simple exhaustive strategy



(b) Pruning based strategy



(c) Gradient descent based strategy

Figure 5.4: Building complex strategies

5.4 Discussions on the Proposed Design Space Exploration Methodology

5.4.1 Limitations of the Approach

Our approach strongly relies on the quality of the estimators to perform quick space traversals while achieving accurate estimations, with the objective to provide realistic solutions.

It is even more true when looking at the proposed frequency estimation methodology, as it remains as complex as what synthesis is — in an algorithmic meaning — and will probably not cope with the *Field-Programmable Gate Arrays* (**FPGA**) specificities, resulting in erroneous estimations after potentially longer processes. However, more accurate estimation methods exist at various levels of granularity which could improve the explorations quality, and our methodology would greatly benefit from proposing to its users both multi-level and multi-fidelity estimators [YHC+21][LC18].

On the other hand, more complex exploration schemes are yet to be proposed in order to achieve state of the art exploration performances, notably by using meta heuristics for *Multi-objective Optimization Problem* (**MOP**) solving (such as *Genetic Algorithms* (**GA**) [PCS21], Bayesian optimization [LC16] or simulated annealing [WAGM+19]), or supervised learning techniques [NKO19][FKA+20].

The proposed schemes are here introduced as a *proof of concept* of functional programming usage for efficient **DSE**, and the introduced library of basic strategies should be enhanced.

5.4.2 Synthesis on the Contributions

Using the estimation considerations that were exposed in Chapter 4, as well as the *Hardware Construction Languages* (**HCL**) features, we defined two complementary methodologies for *Design Space Exploration* (**DSE**) — namely **meta design** and **meta exploration**.

We put a particular focus on how functional programming can be used to define exploration strategies, by leveraging users expertise to build exploration processes step by step. We then exposed various basic schemes that can be used to build both application and target specific strategies.

The proposed methodologies are built considering naive use cases, and both new estimation methodologies and exploration strategies should be proposed in order to provide a more flexible approach.

6

Experiments and Results

This chapter introduces the experiments that were run to demonstrate the usability of both the estimation and the exploration methodologies that were described in the previous chapters.

We built a **Chisel**-based demonstrator as a *proof of concept* framework, as well as a benchmark of representative applications for analysis and comparison purposes. We then ran multiple experiments to exhibit how *Hardware Construction Languages* (**HCL**) can be used to improve the life of developers, with a particular focus on their usage for *Design Space Exploration* (**DSE**).

Table of contents

6.1 Building a Software Demonstrator 80
6.1.1 Implementation Details 80
6.1.2 Use Case: Exploring GEMM Implementations . . . 83
6.2 Application Benchmark 85
6.3 Experimental Setups 86
6.4 Quality of the Estimators 86
6.4.1 Resource Estimations 86
6.4.2 Timing Estimations 90
6.4.3 Quality of Service Estimations 91
6.5 Comparing the Exploration Strategies 93
6.5.1 Resource Estimation and Convergence Speed . . . 93
6.5.2 Quality of Service Based Explorations 97
6.5.3 Use Case: Exploring Black Scholes Designs 100
6.5.4 Results of the Black Scholes Exploration 105
6.6 Synthesis on the Experiments 109

6.1 Building a Software Demonstrator

In order to demonstrate the usability of the methodologies introduced in Sections 4 and 5, we built a **Chisel**-based framework — which is called *Quick Exploration using Chisel Estimators* (**QECE**) — and integrated it into the *Hardware Construction Framework* (**HCF**).

6.1.1 Implementation Details

QECE was built in a flexible and modular way, to allow users to easily define both custom estimators and exploration strategies. It leverages **scala** high level features to use software methodologies in hardware design processes.

Meta design

To do so, the first step was to enable users to exploit the **meta design methodology**, in order to expose the design space to be explored at module constructor level, as stated in Section 5.2.1. This was made possible by using the **java** annotation system (as can be seen in Listing 5.1 for example). We also added an annotation system to specify the impact of the parameters on given metrics, enabling to leverage the user knowledge at both implementation and exploration stages, by introducing a relationship between parameters and strategies. It is done by defining a new annotation class: `class ImpactMetric(name: String) extends StaticAnnotation` — where the `name` parameter is used to identify a particular metric.[1]

Integrating estimators

In a second time, we developed wrappers around the **FIRRTL** transforms to enable a seamless estimator integration, as exposed in Section 4.4.2.

We implemented both **resource** and **timing** estimators using linear interpolations and macro block replacements to allow an early estimation of the metrics in the exploration flows.

Some empirical *Quality of Service* (**QoS**) estimators were implemented by binding to the **Chisel** simulation backends, as shown in Figure 4.7. Doing so, one can retrieve some metrics from simulations, and can thus build their own test benches, defining both which metrics they want to get and how they are computed. As an example, we added helpers to compute the *Root-Mean-Square Errors* (**RMSE**) from the simulation results, by providing an initial workload and a software golden reference.

[1] The `@qualityOfService` annotation used in Listing 5.1 is a simple alias for `ImpactMetric("qualityOfService")`.

Finally, we enabled building any **analytical estimator** operating on pre-existing metrics in the flow — users can thus easily define analytical formulas directly in their strategies.

Building strategies

We then built a flexible way to enable **meta exploration** by implementing a generic `Strategy` class, which can then be used to define more or less complex strategies, as defined in Section 5.3.

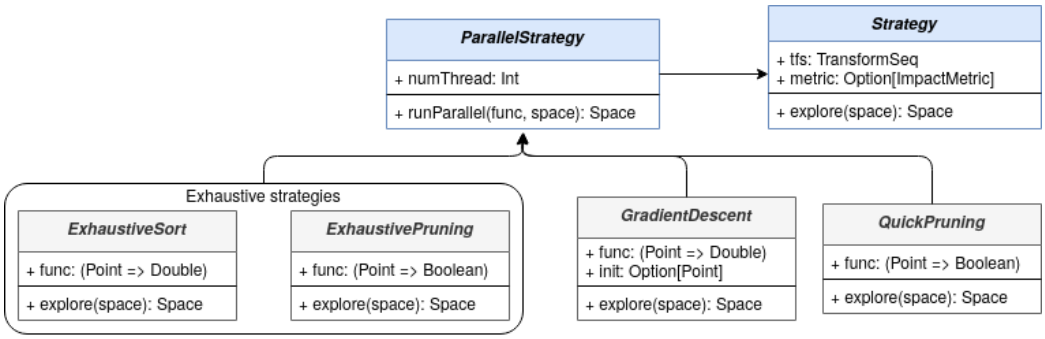


Figure 6.1: Object hierarchy for the implemented strategies

Figure 6.1 introduces the different strategies that were built for this demonstrator, which have already been discussed in Section 5.3.2. As we can see, every implemented strategy inherits from the base `Strategy` class through `ParallelStrategy`, as each can take advantage of parallel threads to speed-up the estimations. The strategies rely on custom defined parameters that allow the users to finely tune every step of the process — it includes the **estimation transforms** `tfs` to be run, the **number of parallel threads** available, or different ways to estimate and compare the explored implementations. For both **pruning strategies**, the users hence need to provide a **boolean function** which specify if a point should be removed from the space, while the two other strategies rely on a **cost function** which build a new metric from a set of existing ones.

The `ImpactMetrics` are considered at strategy level to reduce the design space dimension by removing the non impacting parameters if needed. The same mechanism is used for the dimension removal strategy that was exposed in Section 5.3.2, in order to allow users to explicitly state that a given dimension will no more be relevant for the remaining of an exploration process. Moreover, as the introduced estimation processes are independent, we also implemented some helpers for parallelism exploitation in order to speed up the whole processes — this is particularly helpful for time consuming estimation processes such as synthesis runs — and we used caching techniques in

the strategies to avoid multiple estimations of a same point. We finally implemented composition and strategy building helpers as a syntactic sugar to enable straightforward strategy definitions in a functional fashion — *i.e.* the helpers hide all the side-effects needed for the strategy building, and expose the exploration processes as simple functions operating over spaces.

Design space implementation

Building custom exploration strategies is thus made possible in a expressive way. However, implementing a new strategy often relies on particular operations over the input space, implying that the data structure used for its implementation might have a huge impact on the exploration performances.

In order to allow the users to define their own space structures, we propose a generic implementation for `Spaces` in Figure 6.2, with the basic operations introduced in Section 5.3 that are needed to explore them (*e.g.* `map`, `filter` or `getNeighbours` functions). It also includes operations to remove and add dimensions, for cases where the `ImpactMetrics` are used to discriminate some dimensions in the current exploration. We use a hierarchical, object-oriented approach to allow the users to easily integrate their own space structures if needed. As each implementation defines its own operations, some spaces might be more suited for a given usage, and the strategy builder should choose a structure fitted for their need. For the needs of this demonstrator, we implemented two different structures: `SeqSpaces` and `MatrixSpaces`.

A `SeqSpace` is a simple space based on the **scala** `Seq` collections, that can be used for sort purposes as it exposes an ordered structure. However, finding the neighbourhood of a multidimensional `Point` in a flatten representation of the space requires to scan it exhaustively, evaluating each distance and selecting only the nearest points. As a `SeqSpace` does not exhibit any pattern to directly access to the neighbourhoods, exploring the neighbours of a `Point` in a large space can thus result in an exploding complexity.

On the other hand, a `MatrixSpace` is based on the **scala** representation of matrices, built on multidimensional `Arrays`. Each dimension of the space is represented at a different level in the built **hyper matrix**, and the space also includes a dictionary mapping each dimension to all the possible values. This enables to easily define the neighbourhood of a point, with a complexity growing linear with the different dimensions of the space (which is ideal to define efficient **gradient descent based strategies**). This also means that the matrices may be sparse, for points that have already been pruned. However, such `Space` cannot be sorted as the structure does not exhibit an order, and the **scala** `Array`-based implementations can only be used for dimensions ranging from 1 to 5 — which is a limitation of the language — even if adding dimensions is possible in a custom way if needed.

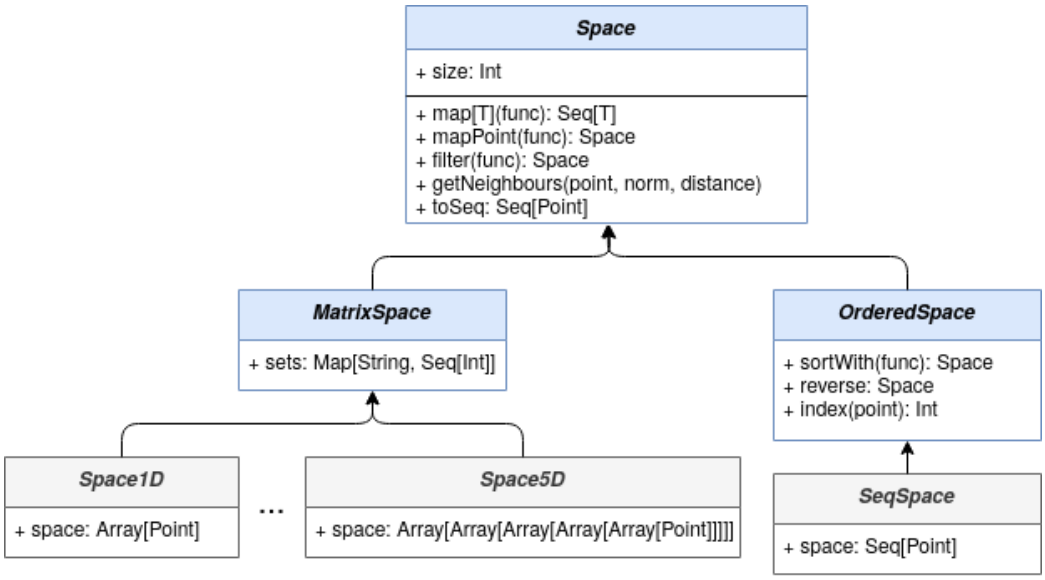


Figure 6.2: Proposing different space implementations

A functional demonstrator

Using those two space implementations and the five basic strategies introduced in Chapter 5, we built **QECE** as a *proof of concept* framework that will be used to demonstrate the advantages of the proposed methodologies.

We deliver **QECE** as an open-source solution, to enable using the introduced methodologies in any **Chisel**-based project [FMR21c].

6.1.2 Use Case: Exploring GEMM Implementations

We introduce a (simplified) example of the usage of **QECE** for a **gradient descent based exploration** on *General Matrix Multiply* (**GEMM**) kernels in Listing 6.1, implementing the strategy exposed in Figure 5.4c.

For each strategy, the user needs to define the set of **estimation transforms** `tfs` to be run, and some functions to specify the exploration behaviour:[2] for both `sort` and `gradient` steps, he needs to provide a function `func` to associate a cost to each implementation, as well as a comparison function `cmp` to expose an order relationship, while the `prune` step only requires a boolean function (also called `func`) to specify whether a point is to be removed or to be kept in the resulting design space.

[2]The `++` operation is used to build an ordered set of **estimation transforms**.

The different metrics at stake in this use case are `%lut`, `%dsp` and `throughput`, which respectively corresponds to the usage of **LUTs** and **DSPs**, and to the theoretical throughput of the considered implementations.

The defined strategy is based on three consecutive steps:

1. **lines 2-5:** Estimate the resource usages and **prune the implementations** that consume **too many resources**.[3]
2. **lines 7-10:** Sort the design space to **select the widest implementation** that still fits on the target board, after this first pruning.
3. **lines 12-17:** Use this implementation as the starting point of a **gradient descent algorithm** based on synthesis runs, aiming to find local optimum in an already pruned space (using the **throughput (GOp/s)** as the objective function to optimize).

As the **GEMM** algorithm is computation intensive, we chose to consider only the *Look-Up Tables* (**LUT**) and *Digital Signal Processors* (**DSP**) usages for step 1., as the wide implementations will probably saturate the available **LUTs**/**DSPs** before the other resources — it is important to note that this decision is based on our expertise about hardware design. For step 2., we then chose to define the **widest design** as the implementation estimated to consume the greatest amount of **LUTs** for similar reasons, with the hypothesis that the implementation with the maximal throughput — but that still fits on the target board — will be near this estimated widest implementation. This is then used in step 3., as the gradient strategy will use the first point of the sorted space as the starting point for the descent. We thus improve the convergence speed as we know that the implementations where both **LUTs** and **DSPs** are estimated to be low will probably not be the optimal ones.

We defined some aliases for the estimation transforms to be run, to keep the strategy concise and intelligible:

- `TransformSeq.resources` performs a **FIRRTL** estimation of the circuit resource usage, using the methodology introduced in Section 4.2.
- `TransformSeq.synthesis` runs a synthesis backend and performs a report analysis to provide both resource usage and operating frequency.
- `TransformSeq.throughput` computes the throughput using an analytical formula (Eq. B.3). The implementations that did not fit on the target board after the synthesis are marked as having a null throughput.

[3] Thresholds are arbitrary and shall depend on the accuracy of the used estimators.

Using those transforms, we can express the exploration flow schematized in Figure 5.4c in a readable, intelligible manner, by composing exploration steps in a functional fashion. Each exploration step — *i.e.* `prune`, `sort` and `gradient` — corresponds to a `Strategy` as defined in Section 6.1.1, defining the estimation transforms to be run and the `Space` operations to be performed, resulting in a complex, application specific strategy.

```
explore(
    // prune the designs using too much DSP/LUT
    prune(
        tfs  = TransformSeq.resources,
        func = (%dsp > x || %lut > y)
    ),
    // select the widest design to start descent
    sort(
        func = %lut,
        cmp  = (_ > _)
    ),
    // use gradient descent for exploration
    gradient(
        tfs  = TransformSeq.synthesis ++
               TransformSeq.throughput,
        func  = throughput,
        cmp   = (_ > _)
    )
)
```

Listing 6.1: Defining a gradient descent based strategy (Fig. 5.4c) in QECE

6.2 Application Benchmark

To demonstrate how **QECE** can be used to improve the developer productivity, we built an open-source benchmark of digital applications that are relevant for **FPGA**-based implementations [FMR21b].

Appendix B exposes how the **meta design** methodology has been applied to each kernel of the benchmark. More specifically, Table B.1 introduces its composition as well as all the exposed design spaces.

6.3 Experimental Setups

The experiments were run on two different experimental setups, as two servers were available. Their characteristics are introduced in Table 6.1.

Name	Mark	#Core	Frequency	RAM	Vivado
Server 1	†	12	3.46 GHz	78.8 GB	v2017.3
Server 2	‡‡	24	3.2 GHz	188 GB	v2021.1

Table 6.1: Experimental setups characteristics

It is important to note that *vivado* specifications claims that at most **8 GB** of **RAM** will be used when targeting a *VC709* board [Xil21a]. However, the memory consumption is given only for designs that consume $\approx 80\%$ of the available resources — meaning that, for wider designs, the memory usage may explode. To cope with this problem, we had to define a **2-hours synthesis timeout**, in order to keep the memory usage under constraint and avoid crashes. We empirically noticed that doing so keeps the memory usage under 20 GB for each synthesis, meaning that we can respectively run at most 4 and 9 synthesis at once on server 1 and 2. As the version of *vivado* can lead to different implementation choices, it is important to consider which version of the software suite was used for syntheses — it is also important to note that given the version and/or the synthesis heuristics used, synthesis results may vary a lot: the whole flow is highly dependent on the quality of the synthesis software used.

For the next sections, please refer to the column **Mark** to know which setup was used for a particular experiment.

6.4 Quality of the Estimators

In this section, we will discuss the quality of the estimation methodologies that were defined in Chapter 4, and show that our demonstrator is able to leverage imperfect yet exploitable estimators for both resource and **QoS** metrics. We will also tackle the usability of our approach for timing estimation, as well as limitations of such approach in quick exploration processes.

6.4.1 Resource Estimations

Figure 6.3 introduces a measure of the *Quality of Results* (**QoR**) of the **graph level resource estimators** — or **FIRRTL**-based estimators — that were

described in Section 4.2. The **QoR** is computed with respect to the synthesis results for 5 different **meta designs** (see Table 6.2).

For each implementation, we ran three types of resource estimators:[4]

1. **FIRRTL**-based estimation, without macro block (see Section 4.2.2)
2. **FIRRTL**-based estimation, using macro blocks (see Section 4.2.5)
3. **Synthesis** based estimation, used as baseline (*i.e.* reference value)

We then plotted the relative difference histograms of both estimations 1. and 2. with respect to 3., respectively in Figures 6.3a and 6.3b, to analyse the **QoR** of both estimation methodologies. To do so, for each implementation of each kernel, we tried to run those three estimators, computed the relative differences with respect to the synthesis results, and built relative difference classes for each resource type (**LUTs**, **FFs**, **DSPs** and **BRAMs**). Those classes were then plotted in order to exhibit the impact of each estimation methodology, for all the considered resources.

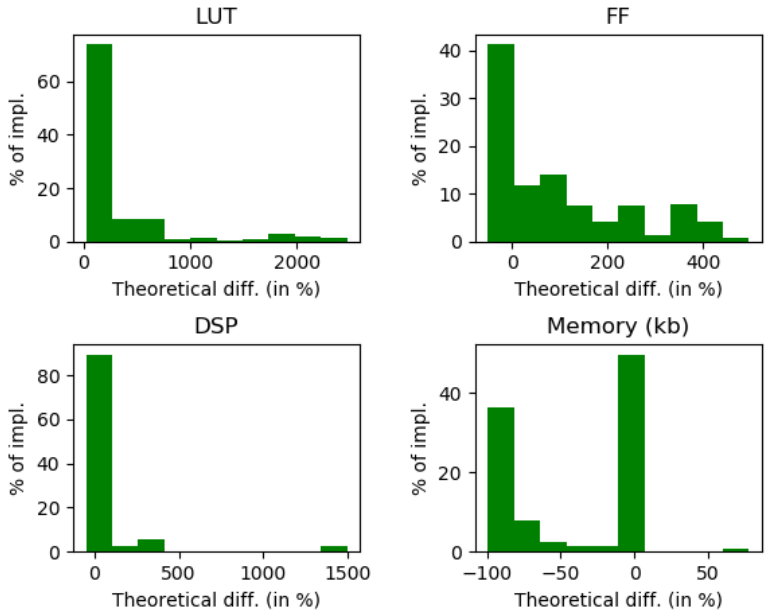
Kernel	Space	Estimations (*failures*)	Estimation time		
			Synthesis	Macro	No macro
GEMM‡‡	168	151 (46)	15h48m11s	2h00m42s	25m55s
Black Scholes‡‡	162	42 (3)	2h57m16s	10m19s	8m12s
Pi‡‡	162	90 (12)	39m50s	4m14s	1m05s
FFT‡‡	200	173 (22)	9h51m16s	3h14m06s	2h14m37s
Dot product‡‡	144	144 (0)	27m54s	29s	26s
Total	**836**	**600 (83)**	**29h44m27s**	**5h59m50s**	**2h50m15s**

Table 6.2: Design spaces and running times of resource estimators (Fig. 6.3)

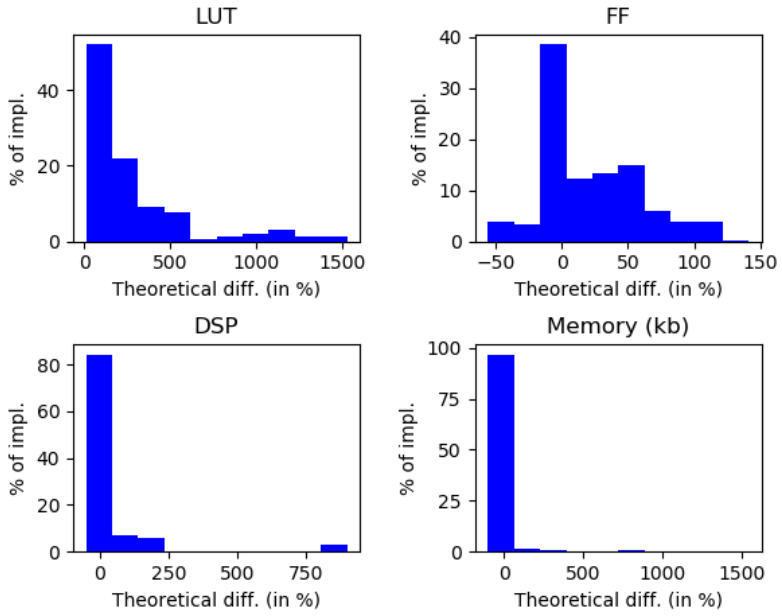
We observe that using the naive approach — *i.e.* without macro block replacement (Fig. 6.3a) — the estimation accuracy is quite variable:

- the **LUTs** are estimated within $[\![0\%, 1000\%]\!]$ of the real usage for more than 80% of the considered designs
- the **FFs** are estimated within $[\![0\%, 400\%]\!]$ of the real usage, and even under 200% for a majority of designs
- the **DSPs** are almost always estimated perfectly, but errors in the estimations may result in a 1500% overestimation
- the **BRAMs** are estimated within $[\![-100\%, 100\%]\!]$ of the real usage, and are perfectly estimated for a majority of designs

[4] To keep both resource usage and processing time acceptable, we set a **30 minute timeout** for both macro replacement and resource usage estimation transforms.



(a) Resource estimation (without macro block replacement)



(b) Resource estimation (with macro block replacement)

Figure 6.3: Average relative differences on 5 different **meta designs** between the resource estimations and the synthesis results.[‡‡]
The kernels are introduced in Table 6.2.

Based on the Figure 6.3b, we can then remark that using macro block replacements has a big impact on the estimations **QoR**. The **LUT** estimations are now mainly produced in a $[\![0\%, 500\%]\!]$ interval, while the **FF** estimations are now estimated within $[\![-50\%, 100\%]$. The **DSP** estimations are mainly as good as with the naive version, but extrema are divided by a factor 2, and the **BRAM** usage is now always perfectly estimated.

We can thus claim that those estimators are not perfect, and that the estimation variability could be considered too big to be exploitable. However, some tendencies can be exhibited here, that could be used to take decisions based on the estimations only:

- for most designs, using a factor 6 on the **LUT** estimation can enable to estimate whether a design fits on the target board or not
- similarly, using a factor 2 for **FFs** can lead to realistic decisions
- both **DSP** and **BRAM** estimations can be trusted for most designs

As for the estimation process speed, we can observe in Table 6.2 that both the naive and the macro based approaches are way faster than synthesis runs, which was the main objective of those techniques. We can also remark that using the macro block replacement methodology can consume a lot more time that the naive approach, and it is specifically true for large and complex designs such as **GEMM** or **FFT**. It is actually due to the implementation of the macro replacement technique, which is based on building another graph from the **FIRRTL** representation, and which does not seem to scale on large designs. Most of the time is due to timeouts of the graph building processes, and this could be mitigated by using a more optimized technique to do so.

We thus showed that the graph level techniques that were introduced in Chapter 4 can be used to perform an early resource estimation, and could be leveraged for early decision making from the developers. However, we still need to exhibit what impact it can have on those decisions, and it will be discussed later in this chapter.

Remark

While we did exhibit the **QoR** of the estimation methodologies on 5 different **meta designs**, one may want to analyse the quality of both methodologies for a particular kernel. For this purpose, we expose a disaggregated version of Figure 6.3 in Appendix C — except for the **GEMM** kernels, which results are exposed in Figure 6.6 as they will be used in later experiments.

6.4.2 Timing Estimations

As for the timing estimations, Figure 6.4 introduces some considerations on both the estimation processing times and the **QoR** for **GEMM** kernels, with respect to the synthesis results. Table 6.3 provides more information about the explored design space and the temporal behaviour of these experiments.[5]

Space size	Estimator	Successful estimations	Timeouts	Exploration time
168	Synthesis	102	49	15h48m24s
	RTL estimation (with macro)	94	57	11h20m15s
	RTL estimation (without macro)	-	-	*≈8h*

Table 6.3: Timing estimations over GEMM[‡‡] implementations (Fig. 6.4)

For those experiments, a **2 hours timeout** was used for both macro replacement and critical path estimation transforms, in order to keep the memory usage under constraints and avoid time consuming processes. Doing so, we remark that for both timing estimation approaches — *i.e.* with and without macro block replacement — the estimation time is mostly kept way under the synthesis time. However, for complex designs, it can be longer than the synthesis. This is mainly due to the complexity of the path building algorithm, which unrolls every possible path on the considered design, and is thus similar to the synthesis process — hence, it cannot be considered as a faster approach, as the algorithmic complexities are comparable.

Using only those considerations, we can already state that those approaches cannot be leveraged to build efficient design processes, as for more complex designs, actually synthesizing the circuit will result in a faster feedback for the users. Moreover, we can also remark that the frequency estimation is totally unrealistic, with differences of hundred of MHz between the estimations and the synthesis results.

Based on those results, we consider this timing estimation methodology to be unpracticable with both approaches, and will not consider those estimators to be usable in the rest of this work. However, some methods exist to provide more realistic estimations of the operating frequency [KC20][PCS21], and integrating them in **QECE** is considered, as they could bring both fast and interesting feedbacks to users and tools.

[5]For the naive approach — *i.e.* without macro block replacement — the timing estimations are not fully available, as the memory blocks cannot properly be estimated. However, the path building algorithm was run anyway to estimate the processing time.

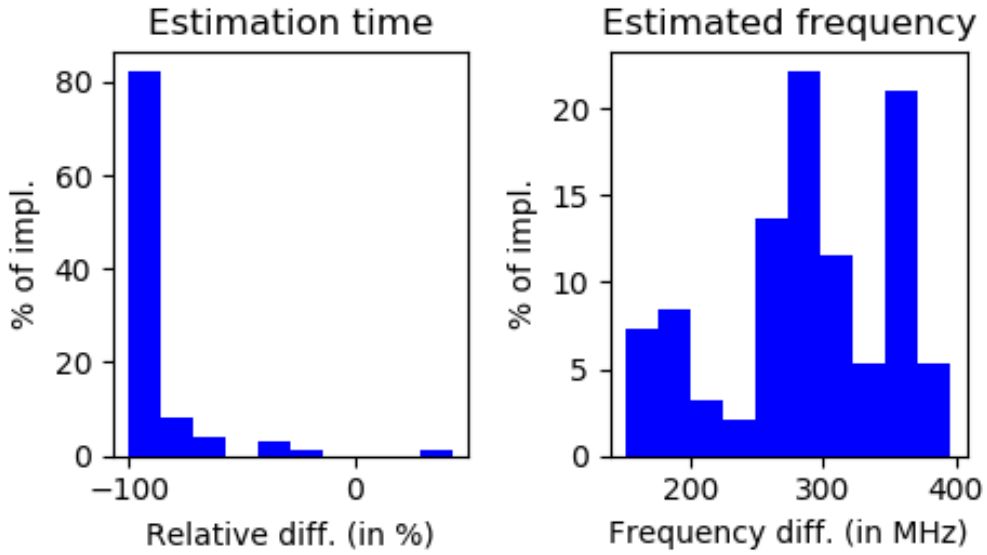


Figure 6.4: Relative difference between frequency estimations and synthesis results on GEMM **meta design**‡‡ (*using macro block replacement*)

6.4.3 Quality of Service Estimations

In order to estimate the **QoS** of a design, we propose an *Application Programming Interface* (**API**) to users in order to extract custom defined results from a simulation backend, as specified in Section 6.1.1. To define an estimator, one needs to provide a **simulation test bench** as well as a way to compute and extract the **QoS** of the design with respect to a given workload.

Figure 6.5 introduces a simple use case on the **dot product** kernel, using **RMSE** as an error metric. We here provide a simple analysis of the impact of the data representation on the result of a dot product, and display the error rate, showing that integrating a custom defined **QoS** metric is possible in **QECE**. To define the data representation, we use a fixed point format and consider two parameters: the number of bits used for the *dynamic* (*i.e.* the number of bits before the binary point), and the number of *precision* bits (*i.e.* after the binary point). As the *parallelism level* does not impact the **QoS**, this dimension is not considered, while the number of elements by input vector is fixed to 8, for visualization purposes.

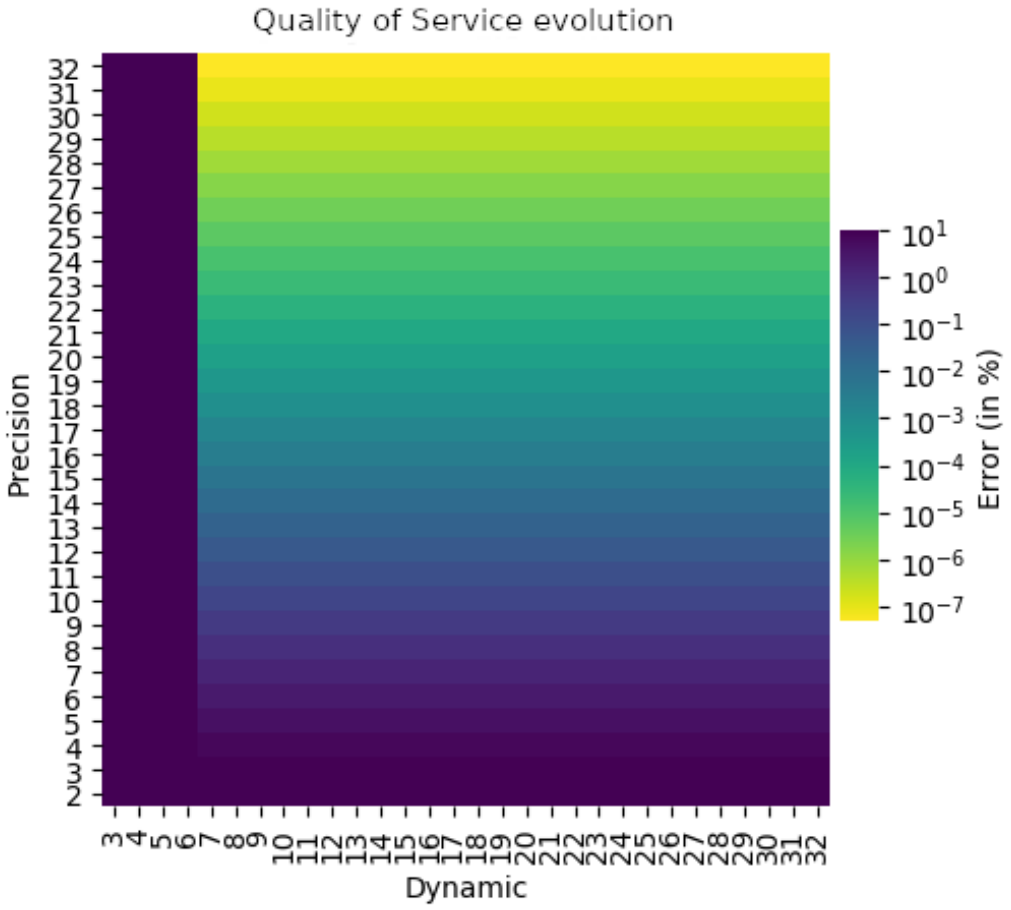


Figure 6.5: Heatmap of the Quality of Service estimations for the dot product **meta design**‡‡ (*vector size is fixed to 8 elements*)

The empirical approach for this estimator is based on comparing the simulation results with a software version of the same algorithm, based on a floating-point representation. Doing so, we can select only the designs which do not introduce a significant error in their results — on this use case, it actually enables to compare the different data representations to select the ones sufficiently wide to absorb the errors that are induced by the use of a simpler representation. As the hardware implementation of *Floating-Point Units* (**FPU**) requires a lot of resources and time, such approach can be used to select a more efficient way to implement an algorithm, if the users can precisely state which error rates are acceptable.

As can be seen on the heatmap, the implementations that use not enough *dynamic* bits are marked with a 10% error rate, as representation overflows occurred during the simulations, resulting in a large error.[6] Moreover, we can also observe the impact of the *precision* parameter on the **QoS**, from large representations where the error rate is under 10^{-7}% to more compact ones, where the error rate reaches 10%.

While this use case is quite simple, it demonstrates that one can easily integrate an empirical, application specific estimator to estimate the **QoS** of its circuits, providing him with an interesting feedback that can be leveraged to select the best implementation that satisfy its accuracy need.

For example, in Figure 6.5, we can (visually) state that if the applicative needs require an error rate lesser than 0.1%, any implementation with more than 6 bits of *dynamic* and 10 bits of *precision* is enough.

6.5 Comparing the Exploration Strategies

We will now discuss how the user expertise can be leveraged for an efficient **DSE** definition, using the methodologies introduced in Chapter 5.

For each strategy that will be compared in this section, the strategy definition is based on assumptions on both considered algorithm and target board, as our approach is based on providing users with ways to exploit their expertise instead of trying to define generic strategies for every algorithm.

6.5.1 Resource Estimation and Convergence Speed

In a first time, we want to demonstrate how high level estimations of the resource usage can be used for the quicker convergence of an exploration strategy. We consider the exploration of two different **meta designs** — **FFT** and **GEMM** — and compare the three exploration strategies that were introduced in Figure 5.4. The results are exposed in Table 6.4. For those experiments, the **QoS** is not considered, hence the element width is fixed by the developer (assuming that they can specify at exploration time that the chosen data representation is acceptable for the given use case), meaning that the *bit width* dimension is not considered in those explorations.

[6] The error rate introduced in Fig. 6.5 is capped at 10% and uses a logarithmic scale to properly visualize the smaller error rates, which are the region of interests in such explorations, as the acceptable error rate should be under 10% to be meaningful.

Kernel	Strategy	Best throughput	#(space)	#synth (#timeout)	Time	Speed-up
FFT128†	*Exhaustive* (Fig. 5.4a)	*1.767 Tb/s*	7	*7 (0)*	*00h22m45s*	-
	Pruning (Fig. 5.4b)	1.767 Tb/s		7 (0)	00h24m14s	×0.94
	Gradient (Fig. 5.4c)	1.767 Tb/s		3 (0)	00h19m51s	×1.15
FFT512†	*Exhaustive*	*5.479 Tb/s*	9	*9 (0)*	*02h11m51s*	-
	Pruning	5.479 Tb/s		9 (0)	03h17m52s	×0.66
	Gradient	5.479 Tb/s		3 (0)	02h18m29s	×0.95
GEMM†	*Exhaustive*	*231.334 GOp/s*	41	*41 (19)*	*13h51m56s*	-
	Pruning	231.334 GOp/s		26 (7)	08h52m00s	×1.5
	Gradient	231.334 GOp/s		6 (1)	03h21m06s	×4.1

Table 6.4: Comparing different exploration strategies with no quality of service concerns. *(Exhaustive strategies are used as baselines)*

Defining a pruning function

For both kernels, we used a simple hypothesis to define the pruning function:

Hypothesis 6.1. *Both FFT and GEMM kernels are computation intensive.*

This means that the computational resources — *i.e.* **LUTs** and **DSPs** — will be saturated first by the synthesis tool, and that removing too wide designs can be done by considering only those metrics.

We thus defined the following **pruning function** to be applied in both **pruning** and **gradient** strategies:

$$prune_{est.}(p) = LUT_{est.}(p) > 200\% \vee DSP_{est.}(p) > 100\% \tag{6.1}$$

The thresholds were chosen with respect to the considerations that were discussed in Section 6.4.1 and to the **QoR** of the resource estimations on **GEMM** kernels, as presented in Figure 6.6b.

Exploring *Fast Fourier Transform* implementations

For the **FFT** explorations, the design spaces are reduced as the size of the algorithm (see Table B.1 for the instantiation parameters) is considered fixed by the developers at exploration time — meaning that **FFT** implementations of different sizes are not compared during the exploration processes. This results in a single dimension exploration space to be explored, for each possible **FFT** size. We only considered two different sizes — 128 and 512 — and we remark that for both explorations, the **pruning strategy** is slower than the baseline, as no actual pruning is done in the design space. In fact, we actually applied the exhaustive strategy after running resource estimations, resulting in a consequent overhead with respect to the baseline.

On the other hand, we remark that the **gradient strategy** results in less synthesis needs for the same best fit finding — meaning that even if the exploration time is not much impacted by the strategy choice, we can reduce the

global processor time needed for parallel syntheses. In fact, for the **FFT**512, the actual convergence time is biased by synthesis timeouts, meaning that improving the resource estimators could enable to prune more efficiently the design space, and could help us avoiding long and non converging syntheses.

These use cases can be used to discuss the importance of the adequacy between the chosen exploration strategy and the explored kernel. Indeed, applying a preliminary pruning function to those use cases is not useful to speed-up explorations, and just results in a useless time overhead, while the gradient approach is practical and results in a speed-up that compensates that overhead.

Exploring *General Matrix Multiply* implementations

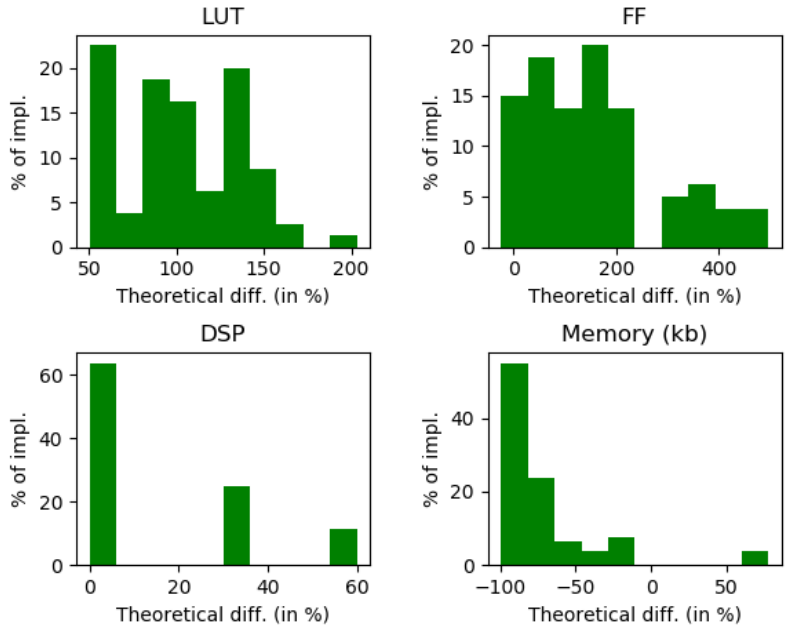
For the **GEMM**-based explorations, we assumed that the defined pruning function (Eq. 6.1) is more relevant, as it is based on an analysis of the **QoR** of the estimators on the **GEMM** implementations. We observe that applying this pruning function enables to reduce the number of implementations to be synthesized in the **pruning strategy** by more than a third. More importantly, it reduces the number of synthesis timeouts by more than a half, as some non fitting designs that would cause long or non converging synthesis processes are not considered in this approach.

In addition to this, we observe that using the **gradient strategy** — *i.e.* leveraging both high level resource estimations and clever space traversal — results in $\times 6.8$ less synthesis runs (6 instead of 41) and a $\times 4.1$ exploration speed-up.

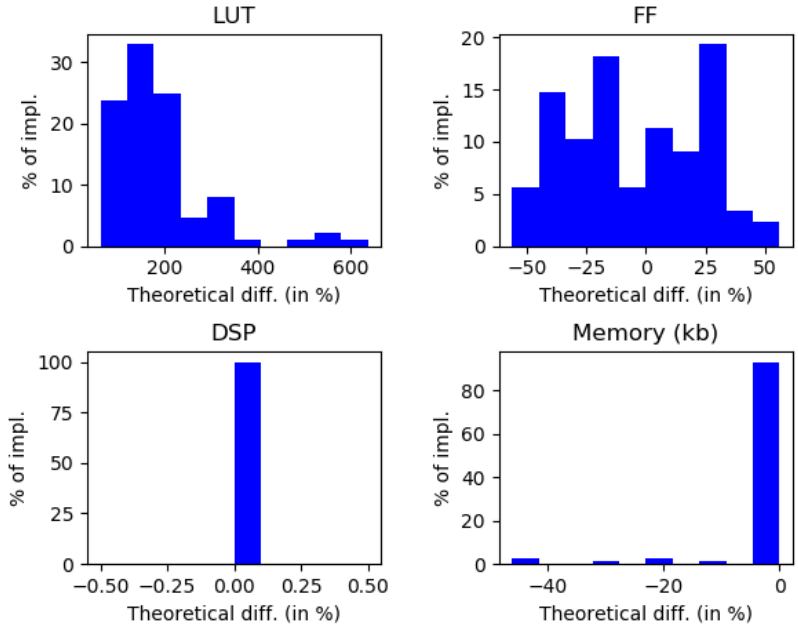
We thus showed that building an application specific, user defined strategy enables to speed-up exploration processes while producing comparable solutions for this use case.

Disclaimer on the definition of the pruning function

One could argue that defining the pruning function after synthesizing the whole design space to analyse the **QoR** of the resource estimators is not practical, as the goal of this methodology is to avoid such exhaustive exploration. This is mainly due to the fact that the applied estimation methodology is application specific — as the **meta designs** characteristics have a heavy impact on the resource estimator **QoR** (see Appendix C for more information) — and could be addressed by providing more accurate and reliable resource estimators in the framework. This could be used to define more generic pruning thresholds that would not require to run application specific syntheses to define it. We could also run syntheses on a subset of the target design space to define this pruning function.



(a) Resource estimation (without macro block replacement)



(b) Resource estimation (with macro block replacement)

Figure 6.6: Relative difference between resource estimations and synthesis results on GEMM implementations[‡‡]

6.5.2 Quality of Service Based Explorations

We now consider the **QoS** of the generated designs, and demonstrate how the proposed methodologies can enable to define multi-concern exploration strategies for custom *Multi-objective Optimization Problem* (**MOP**) solving. In contrast to the previous explorations, we do not make any assumption over the data representation — particularly over the bit widths — and delegate the responsibility of choosing a relevant data type to the exploration flow.

Comparing different pruning strategies

To begin with, we analyse the impact of the pruning strategies over both quality and speed of the pruning. In other words, we will compare the **exhaustive** and the **quick pruning** strategies defined in Section 5.3.2 to check that the **quick pruning** strategy prunes the same implementations that the exhaustive approach does, while exhibiting the impact of the implementation details on the run time.

First of all, we use the **dot product meta design** to measure the **QoS** of each implementation using empirical transforms. To select only the implementations that are compliant with the accuracy needs, the user provides an **error threshold** — here accepting a 1% error rate — and uses the provided heuristic for **quick pruning** (Fig. 6.7) to partition the different spaces more efficiently than by applying an exhaustive filtering. The heatmap represents the results of the exhaustive approach to illustrate the distribution of the error rates, while the grey squares constitute the frontier that was built in the Algo. 5.2, and is used to build the pruned design space in the end (*i.e.* the implementations that are "above" the frontier). We remark that both strategies produce the same resulting space, meaning that the **quick pruning strategy** can be used for quicker convergence of **QoS**-based explorations, at least for kernels that are assumed to comply with Hypothesis 5.2.

Figure 6.7: Quick pruning strategy for Quality of Service based exploration on the dot product **meta design**‡‡ (*considering vectors of 8 elements*)

Optimization	Simulated impl.	Time	Speed-up
Exhaustive pruning	*3844*	*12m22s*	—
Space reduction + exhaustive pruning	961	03m07s	×3.97
Space reduction + quick pruning (SeqSpace)	255	05m03s	×2.44
Space reduction + quick pruning (MatrixSpace)	255	00m49s	×15.14

Table 6.5: Pruning results with respect to the used optimization on the dot product **meta design**‡‡

We now consider the impact of both strategies and of the implementation details on the runtime. Table 6.5 exposes 4 different ways to prune the design space, that are based on the composition of three techniques:

- Space reduction — *i.e.* reducing the number of explored dimensions using the users knowledge about the parameters impact (Section 5.2.3)
- Strategy choice — either the exhaustive or the quick pruning strategy
- Space structure choice — either using a `SeqSpace` or a `MatrixSpace`

For all the considered techniques, we empirically checked that the resulting design spaces are identical, meaning that the same pruning is performed by those 4 techniques. The impact of the space reduction technique on the exploration time is obviously linear, as the number of estimated implementations is reduced in a linear way. In this context, as the *parallelism* parameter is not considered for **QoS**-based explorations, we reduce by a $\times 4$ factor the number of considered implementations (as this parameter can take 4 different values). Moreover, the number of implementations to simulate (column **Simulated impl.** is also impacted by the chosen exploration strategy, as using the **quick pruning** heuristic can lead to a relevant partition with less simulations to run, with respect to the **exhaustive pruning** strategy.

We can also remark that the performance of the **quick pruning strategy** heavily relies on the chosen space structure: using a `SeqSpace` representation, the quick pruning is actually slower than the exhaustive strategy, while using a `MatrixSpace` results in a $\times 3.8$ ($= 15.14/3.97$) speed-up, with respect to the strategy using both exhausting pruning and space reduction.

We thus showed on this simple example that the **quick pruning strategy** can be used to speed-up a **QoS**-based exploration strategy. Moreover, we provide further analysis about the **QoS** estimation and the quick pruning strategy in Appendix D.

6.5.3 Use Case: Exploring Black Scholes Designs

We now consider a more realistic use case to demonstrate how the proposed methodologies can be leveraged to build a complex, application specific exploration strategy. To do so, we use a **Black Scholes meta design** — for which the implementation details are provided in Appendix B — and define an iterative exploration strategy based on our expertise of **FPGA** design.

As exposed in Appendix B, the **meta design** exposes 5 parameters: the data representation parameters (*i.e.* *dynamic* and *precision*), the number of iterations of the **Monte Carlo** method, the number of parallel cores available, and the number of iterations of the **Euler-Maruyama** method.

Defining the objectives of the exploration

The first thing to do to define an efficient exploration process is to clearly state the goal of the target flow. In this **Black Scholes** based use case, we decided to try to ***maximize the throughput of the generated accelerators, under the constraints that they fit on the target board and produce results with a controlled error rate***.

$$T_{estimation.s^{-1}} = \frac{freq}{\Delta_c} \tag{6.2}$$

$$\Delta_c \approx \frac{nbIter \times nbEuler}{nbCore} \tag{6.3}$$

The throughput — expressed in number of estimations by second — can be approximated easily for these kernels, as they periodically produce a result: it is then computed as the ratio between the **frequency** and the **latency** of a given implementation (Eq. 6.2). A theoretical latency (in number of cycles) — *i.e.* the (fixed) period between the production of each result — can be computed using the Equation 6.3, as a huge majority of the computation cycles are spent in the **Monte Carlo** cores iterations.[7]

We also define an **area** metric to verify that a design fits on the target board, using the maximum values of the usage percentage for the four considered metrics (**LUTs**, **FFs**, **DSPs** and **BRAMs**).

Enhancing the design space

The next step to take in the introduced methodology is to instrument the **meta design** with information on the metrics that will be used in the flow.

[7] This result has been verified empirically, proving that the impact of the control flow variations on the latency is negligible with respect to the actual computation cycles.

```
0 class BlackScholes(
1     @resource @qos @linear(8, 32)  dynamic: Int,
2     @resource @qos @linear(8, 32)  precision: Int,
3     @qos @pow2(5, 10)              nbIteration: Int,
4     @qos @pow2(1, 6)               nbEuler: Int,
5     @resource @pow2(2, 10)         nbCore: Int
6 ) extends Module with Explorable
```

Listing 6.2: Expertise-based design space for Black Scholes **meta design**

For this exploration, we will sequentially consider two concerns: a first stage will be based on the circuits **QoS**, and a second one will work over their resource usage. We hence need to define, for both of those metrics, which parameters have an impact on them, and which does not, based on our knowledge about the algorithm. The analysis to do so is introduced below, and is used to enhance the initial design space as exposed in Table B.1, leading to the one introduced in Listing 6.2.

Due to the probabilistic nature of the **Monte Carlo** method, most of the exposed parameters have an impact on the **QoS**, either because they act on the data representation (as it is the case for both *precision* and *dynamic* parameters), or because they impact the number and the precision of the sampling in the Black Scholes equation (Eq. B.7), as do the *number of iterations* and the *number of Euler-Maruyama iterations*. In contrast with these statements, the *number of cores* does not impact the circuits **QoS**, as it only modifies the temporal behaviour of the designs, and not their functionalities — this parameter is hence not annotated as impacting the quality of service (using the `@qos` annotation).

As for the resource usage, we now try to identify which parameters impact the size of the generated designs. We can use our knowledge about the meta design structure to state that both *iteration parameters* (the global number of **Monte Carlo** iterations, and the number of inner **Euler-Maruyama** iterations) does not significantly impact the amount of resources used in a design. As a matter of fact, the number of iterations will act on the global latency of the design (Eq. 6.3), but will have a small impact on the resource usage, which mostly depends on the number of parallel cores and their size.[8] We thus annotate the data representation parameters and the *nbCore* parameter with `@resource` to guide the exploration steps.

[8]The number of iterations actually impacts some counters in the control flow of the designs — however, the impact is negligible with respect to the resources needed to implement the computation cores.

Outlining the estimation transforms

Once the design space has been enhanced to guide the exploration, we need to define the different steps that are to be taken in the process. To build a compact definition of this exploration strategy, we define some helpers in Listings 6.3 and 6.4 to build the transforms that are to be used in the flow.

Listing 6.3 demonstrates how empirical **QoS** estimators can be integrated in **QECE**, in a condensed way. We define `QualityOfService.simulation` as a transform calling the simulation backend to estimate the **RMSE** of a given implementation. The different **generation parameters** of the simulated implementation are needed to instantiate the simulation test bench, and are thus retrieved from the **point metrics** (the `m` variable at line 1., which is used to define the parameters in lines 5-9). Moreover, some other parameters can be provided by the users when they are building such **estimation transform** — *e.g.* in this use case, they provide the number of software iterations (line 10.) to compute the reference value for the **Black Scholes** estimation, as well as the number of hardware simulations to run on each implementation to provide a significant result (line 11.). They also provide a *workload* (line 12.), which is the data distribution to be used in the test benches.

Remark: The circuits **QoS** are estimated using an empirical approach (Section 4.3.3), as we use it to demonstrate the practicability of using the simulation backend in the estimation methodologies. However, one could also define an analytical formula to specify the relationship between the number of iterations of a given implementation on one hand, and the accuracy of the **Black Scholes** approximation on the other. While it would not be used to estimate the impact of the data representation over the **QoS**, it could allow to eliminate the parameters relative to the number of iterations from the design space, at least for the exploration steps that rely on simulations.

As for Listing 6.4, it implements the analytical formulas from Equations 6.2 and 6.3. The transforms are easily defined, by specifying a pair $(n \rightarrow f : \{m_x, x \in [\![0, p]\!]\} \Rightarrow m_{x_{p+1}})$ with n the name of the generated metric, and f a function which operates on a set of metrics $\{m_x, x \in [\![0, p]\!]\}$ to generate a new metric named n to be propagated in the exploration process.

As can be seen in lines 1-4, the **latency** is estimated in the **pre-elaboration** stage (with respect to the **API** introduced in Section 4.4.2), as it only relies on the generation parameters.

On the other hand, both **area** and **throughput** are estimated in the **post-elaboration** stage (lines 5-12). This is due to the fact that the `Transforms.throughput` method is going to be called in the same exploration step that the synthesis flow: as the formulas rely on the synthesis results, they must be computed after the elaboration process.

The `area` metric is computed by retrieving each resource percentage metric from the current implementation, using the maximal value to check if a design fits on the target board. To artificially eliminate the non fitting designs, we then use a temporary metric `_throughput` to store the theoretical throughput as computed using Equation 6.2 (lines 8-9), but only assign it to the actual `throughput` metric if the `area` metric is under 100% (lines 10-11).

```
0  object QualityOfService {
1    val simulation = TransformSeq.simulation(m =>
2      (
3        c =>
4          new BlackScholesTester(
5            m("dynamic").toInt,
6            m("precision").toInt,
7            m("nbIteration").toInt,
8            m("nbCore").toInt,
9            m("nEuler").toInt,
10           nbSoftIteration,
11           nbSimulation,
12           workload
13         )(c)
14       )
15     )
16 }
```

Listing 6.3: Defining an helper for empirical quality of service estimations

```
0  object Transforms {
1    val latency = TransformSeq.preElab(
2      "latency" ->
3        (m => (m("nbIteration") * m("nbEuler")) / m("nbCore"))
4    )
5    val throughput = TransformSeq.postElab(
6      "area" ->
7        (m => max(m("lut"), m("ff"), m("dsp"), m("mem"))),
8      "_throughput" ->
9        (m => m("freq") / m("latency")),
10     "throughput" ->
11       (m => if (m("area") > 1.0) 0.0 else m("_throughput"))
12   )
13 }
```

Listing 6.4: Transform helper for the exploration

Defining the exploration strategy

After defining the transforms that are going to be used in the exploration process, the last step is to define the steps to be taken. We define a **five steps** strategy for this use case, for which the code is exposed in Listing 6.5:

1. **lines 1-5:** the design space is pruned for implementations that are estimated to induce an error of more than **5%** (line 3.).

   For this step, we use the **quick pruning** algorithm (Algo. 5.2) to quickly partition the space in this pruning step (line 1.). Moreover, as we only consider the **QoS** of the different implementations, we specify that the framework can eliminate all the dimensions that are not annotated with `@qos` (line 4.). Indeed, we use the `QualityOfService.simulation` transform from Listing 6.3 to compute the **QoS** (line 2.).

2. **line 6:** this line specifies how we reduce the dimensions of the design space for the remaining exploration steps. All the **meta design parameters** (see Fig. 6.2) that are not annotated with `@resource` do no act on the resource metrics, and are thus removed from the dimensions. They are in fact the parameters acting on the number of iterations: *nbIteration* and *nbEuler*. Actually, we can state two things at this point: all the remaining designs are acceptable with respect to the needed **QoS** in this use case, and reducing the number of iterations can only benefit to improve both the resource usage and the throughput.[9]

   The boolean parameter of the `context.reduceDimension` method specifies that the dimension removal will project the parameters on the **minimal values** in the space — as we want to keep the number of iterations as low as possible among the remaining designs.

   By removing those two dimensions from the design space, the number of remaining implementations is hence heavily reduced.

3. **line 7:** we compute the latency of each remaining implementation.

4. **lines 8-12:** we select the "minimal point" of the remaining design space (the point with the minimal sum of parameters) and put it in front of the design space, as the next step will use it as a starting point.

5. **lines 13-17:** we use the **gradient descent** algorithm (Algo. 5.1) to find a local optimum with respect to the objective of the exploration.

[9] Increasing the number of iterations can only improve the **QoS** at the cost of **latency** increase (Eq. 6.3), yet we already have a sufficiently low error after the pruning step.

In this step, we run syntheses in order to provide a realistic estimation of both resource usage and operating frequency, and it is thus necessary to adopt a clever strategy to limit the number of costly process runs. We hence use neighbourhood explorations to iteratively find an acceptable solution to the users.

```
0  val strategy = context.buildStrategy(
1    context.quickPrune[BlackScholes](
2      QualityOfService.simulation,
3      _.error > 0.05,
4      metric = Some(new qos)
5    ),
6    context.reduceDimension[BlackScholes](new resource, true),
7    context.map[BlackScholes](Transforms.latency),
8    context.sort[BlackScholes](
9      TransformSeq.empty,
10     m => m("dynamic") + m("precision") + m("nbCore"),
11     (_ < _)
12   ),
13   context.gradient[BlackScholes](
14     TransformSeq.synthesis ++ Transforms.throughput
15     func = _("throughput"),
16     cmp = (_ > _)
17   )
18 )
```

Listing 6.5: Expertise-based exploration strategy for Black Scholes implementations

We hence defined a complex, expertise-based exploration strategy which takes the best of the users knowledge to guide the framework and to speed-up the traversal process. In order to provide an analysis on the relevance of such strategies, we ran multiple exploration processes to expose the interests of the **meta exploration** choices that led to the definition of Listing 6.5.

6.5.4 Results of the Black Scholes Exploration

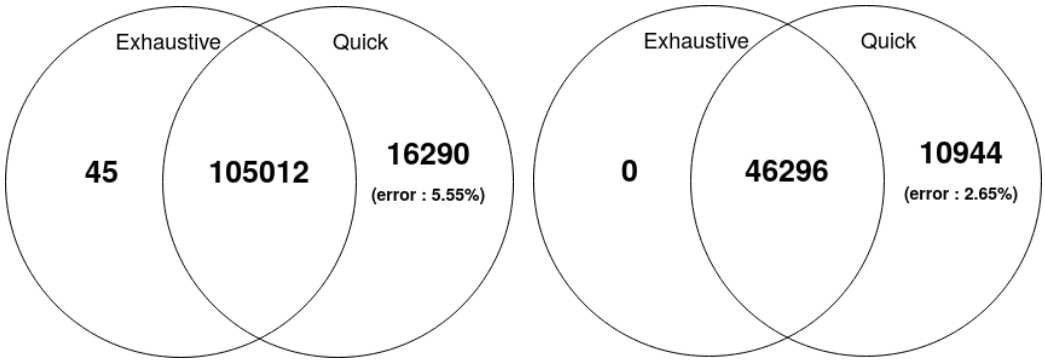
In this last section, we will analyse the results of the exploration strategy defined for the Black Scholes use case, to exhibit the advantages of defining a custom strategy in a functional fashion.

Comparing the pruning strategies

A first analysis can be done about the quality of the pruning processes, by comparing the **exhaustive pruning** and the **quick pruning** strategies on the Black Scholes use case.

Error threshold	Number of impl.	Strategy	Exploration time	Speed-up
5%	202500	Exhaustive pruning	46h29m19s	-
		Quick pruning	32h59m29s	×1.40
2%	202500	Exhaustive pruning	46h21m42s	-
		Quick pruning	48h46m53s	×0.95

Table 6.6: Comparing the pruning strategies over Black Scholes kernels‡‡



(a) 5% error threshold **(b)** 2% error threshold

Figure 6.8: Comparing the accuracy of the pruning strategies on Black Scholes‡‡

Table 6.6 introduces the results of four different explorations processes, to compare both strategies using two different **error thresholds** for the pruning. In those experiments, only the pruning step (lines 1-5 in Listing 6.5, compared to an exhaustive version of it) is considered, and we compare the remaining implementations in Figure 6.8.

As we can see in the table, depending on the error threshold, the **quick pruning strategy** can lead to a faster pruning of the space (as it is the case with a 5% threshold), or a slower one if the frontier is difficult to estimate and requires a lot of neighbourhood exploration steps to do so.

We can also see in Figure 6.8 that the quick pruning heuristic is not perfect: for example, using an acceptable error rate of 5% (Fig. 6.8a), **105012 remaining implementations** were common to both strategies, while **45** were pruned by the quick strategy and not by the exhaustive one, and **16290**

were pruned by the exhaustive strategy, and not by the quick one (on a total of **202500 different implementations**). If we consider the exhaustive pruning to be the baseline — as estimating the **QoS** of each implementation leads to a more realistic pruning of the design space, even if the empirical approach can induce erroneous data — we can remark that the quick pruning strategy tends to prune less implementations that the exhaustive one.

This shows that this strategy can lead to erroneous choice from the exploration tools — however, the average errors in the designs that should have been pruned by the quick heuristic but were not are also displayed in Figures 6.8a and 6.8b (with average errors of respectively 5.55% and 2.65%). These errors are computed on the points that were estimated in the frontier building algorithms, through the neighbourhood exploration: those implementations are thus near to this limit, and this is why they are not pruned as they would have been in an exhaustive process.

Remark: among those points, some errors are approximated instead of being estimated, in order to keep the number of simulations low. This is done on the points that are considered "above" the built frontier, but that were not considered in the neighbourhood exploration process used to build this frontier. In order to produce an error metric anyway, even if not actual **QoS** estimation is performed, an artificial metric is built by copying the error value of one of the points on the frontier to add it to every point that was not estimated in frontier construction process. This means that the average **error** values introduced in Figure 6.8 relies on this approximation, and should be considered carefully.

Exploiting a complex strategy for exploration

We now consider a full exploration process based on the strategy exposed in Listing 6.5, to demonstrate the usability of **QECE** on a complex use case.

Rank	**Parameters**	**Error**	**Throughput** ($est.s^{-1}$)	**Area** Max %	**Area** Resource	**Frequency**
1	[12, 21, 64, 2, 64]	5.34%	125.06	25%	**DSP**	250.13 MHz
2	[12, 20, 64, 2, 64]	4.6%	125.06	25%	**DSP**	250.13 MHz
3	[12, 22, 64, 2, 64]	6.54%	125.03	25%	**DSP**	250.06 MHz
4	[13, 21, 64, 2, 64]	6.06%	125.03	25%	**DSP**	250.06 MHz
5	[12, 22, 64, 2, 32]	5.34%	62.53	12.5%	**DSP**	250.13 MHz

Table 6.7: Best implementations found using the exploration strategy from Listing 6.5‡‡. The whole exploration process took approximatively 38 hours to synthesize the 27 more interesting candidate implementations — through a gradient-based approach — and sort them.

Table 6.7 introduces the five best implementations found at the end of the exploration process — after 37 hours and 47 minutes. The **parameters** are compressed in the table, and correspond respectively to the **dynamic**, the **precision**, the **number of iterations**, the **number of Euler-Maruyama iterations** and the **number of cores**. A total of 27 different implementations have been considered in the last step of the process, which we will denote as being the **synthesis step**.

The first thing we can remark is that the pruning criteria is not strictly respected, as was shown in the previous section — however, the error overhead can be considered acceptable for some use cases, as it barely reaches 30% of the target error threshold.

We can also see that the four best implementations use 64 parallel cores, resulting in a resource usage of 1/4 of the target board, and achieving a throughput of 125 estimations by second.

One could thus argue that we could use up to 256 parallel cores, to fill the targeted **FPGA**. However, after the pruning step, we found that 64 iterations (with 2 Euler-Maruyama inner iterations) were enough to ensure a satisfying **QoS** on the implementations shown in the table. As the number of cores cannot be larger than the number of iterations by design — it is an inner constraint of the Black Scholes meta design used in this use case, and could be addressed by modifying the **Chisel** description — the gradient algorithm did not fully explore the **number of core** dimension, resulting in a sub optimal solution. Nevertheless, as the kernels are independent, this exploration showed that we could fit **four Black Scholes accelerators** in parallel on the target board, using the resulting generation parameters.

As for the temporal behaviour of this experiment, we cannot expose a baseline similar to the previous ones, as an exhaustive exploration of the remaining design space would be too long (approximatively 6000 different implementations to synthesize, which would take up to 500 days to complete, with respect to the 2 hours timeout used).

However, we could cope with this problem by using a more hierarchical approach to build the baseline. For example, some approaches in the literature uses a preliminary **random sampling** to identify **regions of interest** in the design space, *i.e.* sub spaces where the solutions are expected to be the best ones [AGMP21]. This strategy could be leveraged for two usages in our experiment. First of all, we could use partial sampling of the design space to run syntheses and state if the local approach of the gradient algorithm caused a sup optimal choice in the space. Moreover, we could also use such sampling to select the starting point of the gradient descent, instead of arbitrarily selecting a point as it was done (lines 8-12 of Listing 6.5).

6.6 Synthesis on the Experiments

Experimental contributions

In this chapter, we exposed the implementation details of a software demonstrator (named *Quick Exploration using Chisel Estimators* (**QECE**)) for the design methodologies that have been introduced in Chapters 4 and 5. Doing so, we built a generic and modular framework for *Design Space Exploration* (**DSE**) processes targeting *Field-Programmable Gate Arrays* (**FPGA**).

We also introduced a benchmark of algorithms that are representative of the usage of **FPGAs** as hardware accelerators, in order to demonstrate the usability of **QECE** in realistic use cases.

We finally ran multiple explorations as experiments to analyse the advantages and the limitations of both our methodologies and our framework over the hardware development processes. We explored the design spaces of three different **meta designs** — *General Matrix Multiply* (**GEMM**), *Fast Fourier Transform* (**FFT**) and Black Scholes — and considered different objectives and constraints to show that one can define custom strategies for their exploration processes, based on their expertise about both the algorithm being implemented and the **FPGA** board being targeted.

Synthesis on the results

In order to demonstrate how one can leverage its expertise about both the application being explored and the **FPGA** board being targeted, we considered two different types of objectives to be considered at exploration time.

To begin with, we used both **GEMM** and **FFT** **meta designs** to expose how a high level estimation of the resource usage can lead to the building of efficient exploration strategies, and demonstrated that such strategies can lead to speeding up exploration processes by a significant factor, reducing the exploration time from 13 hours to 4 hours only for **GEMM**.

We then used a **Monte Carlo** based design, the **Black Scholes** pricing **meta design**, to show how one can consider the *Quality of Service* (**QoS**) of the developed circuits to build strategies that sequentially consider different metrics to optimize. We implemented a pruning heuristic that can be used to more or less efficiently partition a design space without having to simulate exhaustively the implementations that compose it, with speed-ups up to 40% with respect to the exhaustive strategy, and exposed a complex, expertise based **meta exploration strategy** that can be used to find efficient architectures for the given use case. Doing so, the number of implementations to synthesize is heavily reduced, and only 27 different designs are considered in those long processes, in a total design space of approximatively 6000 different circuits, where an exhaustive exploration is indeed impracticable.

Discussion on the approach

While we did show that exhibiting an expertise based strategy — that uses custom defined metrics and estimation methodologies — can lead to a quicker convergence of the exploration processes toward an acceptable solution (if not an optimal one), we also showed that an inadequate strategy can lead to suboptimal solutions, and can take more time that the standard exploration flows that rely on heavy synthesis tools. The strategies that are being compared here are quite naive and simple, and **QECE** would greatly benefit of the implementation of more advanced heuristics, in order to provide users with a more furnished library of configurable exploration steps.

However, we demonstrated the practicability of a novel way to consider the exploration processes, using the functional programming feature to define such flows in a concise yet intelligible way for the users, by considering an exploration process as a succession of basic steps. We also exhibited the interests of the emerging *Hardware Construction Language* (**HCL**) paradigm for hardware development, proving that a potentially consequent overhead on the *a priori* analysis of the algorithm can lead to reusable hardware generators, which can then be easily adapted to new use cases.

7

Conclusion and Perspectives

While the life of the software developers has greatly benefited from emerging paradigms and new development methodologies in the past decades, developing a digital circuit remains a daunting task, which requires both expertise and time. Initiatives have thus been taken in order to increase the productivity of hardware designers, mostly by providing tools that are able to make the simplest decision in their place, to allow them to focus on the most complex tasks that really requires their skills and knowledges.

Among those initiatives, we propose a new design methodology, which takes the best of an emerging hardware development paradigm — the hardware construction paradigm — to build reusable and adaptable designs. We put a particular focus on how this new paradigm, which comes with emerging features from the software world itself, can bring more expressivity to the designers, and demonstrate its usage on a key challenge of digital design: the design space exploration, *i.e.* the exploration of the different implementation choices that a designer can make in the process of building a circuit.

To simplify our approach of the problem, we consider exploration processes that target reconfigurable circuits — more precisely, FPGA boards — as each implementation technology has its specificities. However, the methodologies introduced in this work could be, with few modifications, applied to the development of other digital circuits, such as ASIC, and could consider various levels of granularity, ranging from the development of basic operators to the exploration of complex, multi-core systems.

After providing an analysis of the domain literature, which focuses on three main concerns when it comes to design space exploration — namely the space exposition, the metric definition and estimations, and the exploration strategies — we introduce the main contributions of this work.

First contribution

First of all, we consider the different metrics of interests that can be used to define the quality of a design, in order to help the users — or an exploration tool — to make clever decisions in their design processes. We provide some insights about what are the common metrics, and introduce a generic approach to define both application specific and non specific concerns, in order to offer an intelligible yet usable approach of the estimation processes. We introduce some estimation methodologies for the key metrics of interests of a hardware implementation: the spatial and temporal aspects, *i.e.* the resource usage and the operating frequency of a FPGA-based implementation.

We also introduce the quality of service as a domain specific metric, as it can only be used in the fields where result approximations can be used to improve the performances of a circuit at the cost of accuracy, and demonstrate how the proposed tool architecture can be leveraged to integrate exotic yet useful metrics in the development flow.

Second contribution

The second and main contribution of this thesis is then addressed, with the introduction of two complementary design space exploration methodologies, namely **meta design** and **meta exploration**. We use the generation feature of the hardware construction languages — which refers to the action of describing a circuit generator instead of a dedicated, non adaptable accelerator — and instrument the framework of **Chisel**, an emerging HCL, to expose clever and concise design spaces and integrate them directly in the code. We then provide a formalization of what is a design space exploration strategy, to demonstrate how the functional programming feature — which is now accessible to the hardware developers, thanks to **Chisel** — can be used to build custom and controllable strategies that takes the best of the users knowledge and expertise about both the application and the target board. A set of basic strategies are provided as a basis, to demonstrate how an iterative approach of the design space exploration problem can lead to an intelligible and concise description of different exploration processes.

Experiments and results

In order to demonstrate how this approach can be used in the daily life of the hardware developers, we developed a **scala**-based framework named **QECE** (*Quick Exploration using Chisel Estimators*), as **Chisel** is a meta language that is built on **scala**. We also provide a set of **Chisel**-based generators for algorithms that are representative of the usage of FPGAs as hardware accelerators. The quality of results of the different estimation methodologies is then analysed, using multiple experiments over so-defined benchmark, and the limitations of those methodologies — which are introduced as a *proof of concept* rather than as usable tools — are discussed. We then expose different exploration use cases, and we demonstrate through various experiments that the defined approach can be used to efficiently solve the design space exploration problem, showing that using the users expertise can lead to faster processes, with no significant deterioration of the quality of the solutions.

However, as the work of a single thesis could roughly be enough to address this challenge in a generic way, multiple limitations as well as some ways to overcome them have been identified.

Limitations and perspectives of evolution

To begin with, the quality of the proposed estimation methodologies is questionable, as we do not reach a state of the art quality. The introduced framework would thus greatly benefit from integrating performant estimators at various levels of granularity and fidelity, to provide the users with a configurable library of estimation processes that one could use and adapt for a specific use case. Recent initiatives have been taken to provide quick and accurate estimations to the users, notably by using machine learning techniques to extract informations from prior estimations [KC20], or statistical approaches to limit the number of syntheses to be run [PCS21], and one should be able to choose and integrate a specific estimation methodology to build their own exploration flow.

Moreover, while we implemented some basic strategies as a *proof of concept*, **QECE** should also offer a library of configurable exploration steps, leveraging emerging approaches and algorithms to build efficient exploration strategies. Doing so, we could provide further genericity and modularity for the users, and the framework could even be leveraged to evaluate the quality of new exploration strategies, providing quick prototyping for exploration processes. For even more modularity, we should also provide a way to integrate external tools at different levels of the framework, for example by using an external software to guide the exploration, as does the E-𝕀DE𝔸 framework with Bellerophon, an evolutionary-based search engine [BTBB21].

Last but not least, we should provide the users with an easy way to visualize the results of an exploration, as solving a multi-objective optimization problem in a multi dimensional space can be very difficult to understand and interpret. To do so, Paletti et *al.* [PCS21] propose to integrate some visual models from the literature, such as the roofline model [WWP09] or the LogCA model [AW17]. As a matter of fact, the users should be able to visualize both the exploration results and the interesting metrics of a given circuits, as they are the ones that will be able to use quick and accurate feedbacks to make clever decisions that an automatic tool could not.

On another side, to demonstrate the usability of the proposed methodologies, we could benefit from a wider benchmark to analyse different classes of algorithms, showing the generic nature of the approach. While we did expose different use cases to exhibit how one can leverage its expertise to define custom exploration strategies relying on different types of metrics of interests (and the corresponding estimation methodologies), more experiments on other scenarios — and other application domains — could demonstrate the impact of the algorithms specificities on a given strategy, for example.

More specifically, some people at TIMA lab are working on different approaches to efficiently implement neural network on **FPGAs**, from the usage of ternary data representations [PBBP+17] to the integration of **LUT**-based logarithm multipliers to our own implementation of a configurable *Multilayer Perceptron* (**MLP**) (Appendix B). Moreover, an industrial collaboration is ongoing with OVHcloud to use **Chisel** to bring more expressivity for the hardware developers [BHM+21]. Such local initiatives could benefit from our exploration framework to enable a quick exploration of the architectural space on different use cases.

Some other applicative domains which relies on heavily parametrized templates for both generation and exploration purposes could also benefit from our work. For example, Delomier et *al.* [DLGCJ20] exposes a model-based generator for the implementation of successive cancellation polar decoders on **FPGA**, and uses **HLS** to generate efficient accelerators from such model, based on architectural and algorithmic parameters. One could thus use our exploration framework to efficiently explore the so-defined design space, after defining the accelerators templates using **Chisel**. In a similar way, Mkhinini et *al.* [MMLT17] leverages **HLS** to accelerate the computations of modular polynomial multiplications in the context of **Fully Homomorphic Encryptions**, exposing high-level parameters to generate the different architectures. Once again, using our exploration methodology and framework could help the users to improve the exploration performances by leveraging their expertise to build efficient strategies.

Synthesizing the contributions

Regardless of the limitations that we identified for this work, we provide a modular framework for the estimation and the exploration of design spaces, which relies on an emerging paradigm. More generally, we propose an initiative to enhance the expressivity of the hardware developers, giving them the opportunity to use powerful features such as object-oriented and functional programming to describe not only the accelerators to be generated, but also the processes to build and explore them.

QECE, a **Chisel**-based framework for estimation and exploration, as well as a benchmark of applications that were used to demonstrate its usage, are provided as open-source projects [FMR21c][FMR21b]. It is important to note that the need for a flexible framework for both estimation and comparison of **Chisel**-based designs has already been expressed in the community, and that we plan on communicating on our solution as soon as possible, notably by proposing different use cases to integrate in the literature.

8

Résumé

Ce chapitre propose un bref résumé du contenu de ce manuscrit, en français. Il s'agit d'une synthèse du travail qui a été effectué durant cette thèse, et qui est décrit en anglais dans les chapitres précédents.

Ce résumé commence par un exposé des motivations à l'origine de ces travaux, et présente ensuite les travaux existants dans l'état de l'art quand au domaine de l'exploration d'espace de conception. Les deux sections suivantes concernent les contributions de cette thèse, respectivement la définition et l'estimation de métriques d'intérêts pour la conception numérique, et une méthodologie novatrice d'exploration d'espace de conception basée sur l'utilisation des langages de construction matérielle, à savoir la **méta exploration**. Ensuite, les expérimentations qui ont été menées pour démontrer l'utilisabilité des méthodologies proposées sont présentées, conjointement avec un démonstrateur logiciel spécialement développé pour cette thèse, nommé *Quick Exploration using Chisel Estimators* (**QECE**). Finalement, une synthèse sur les contributions de cette thèse est proposée, et les limitations de ces travaux sont mises en avant afin demettre l'accent sur les pistes de recherches qui restent ouvertes à la fin de cette thèse.

Table of contents

8.1 Motivations 116
8.2 État de l'art 116
8.3 Métriques et méthodologies d'estimation 119
8.4 Exploration d'espace de conception 120
8.5 Expérimentations et résultats 123
8.6 Conclusion 125

8.1 Motivations

Les motivations qui se cachent derrière ce travail de thèse sont à contextualiser par rapport à l'évolution historique des flots de conception numérique. En effet, les méthodologies de conception de circuit ont drastiquement évoluées au cours des dernières décennies, des méthodes de dessin de masque jusqu'aux techniques de synthèse de haut niveau qui sont désormais utilisées dans l'industrie.

Cependant, les alternatives actuelles qui s'offrent aux développeurs reposent soit sur des langages de description matérielle, comme VHDL ou Verilog, et requièrent un niveau d'expertise ainsi qu'un temps de développement conséquent, ou sur des méthodologies qui impactent les performances des circuits conçus, comme la synthèse de haut niveau ou les langages à domaine spécifique.

Dans ce contexte, nous nous sommes intéressés à l'utilisation d'un paradigme de développement matérielle qui a récemment vu le jour : les langages de construction matérielle, qui, comme leur nom l'indiquent, proposent une approche plus constructiviste de la conception numérique. Se faisant, ces langages permettent de décrire des générateurs de circuits plutôt que des circuits dont la fonctionnalité et les possibilités sont figées à la conception.

Plus particulièrement, nous proposons dans ce travail d'étudier l'utilisation de ces langages dans le contexte de l'exploration d'espace de conception, c'est à dire le processus — manuel, assisté ou automatique — qui consiste à faire varier des architectures équivalentes en terme de fonctionnalité, afin de les comparer entre elles et de sélectionner celle qui répond le mieux à un cas d'utilisation donné.

Les fonctionnalités de génération propres aux langages de construction matérielle se trouvent donc être une opportunité intéressante pour proposer des flots d'exploration d'espace de conception qui, contrairement à leurs équivalents de plus haut niveau (notamment ceux utilisés dans les techniques de synthèse de haut niveau), permettent aux développeurs de contrôler les circuits générés tout en guidant, à l'aide de leurs expertises, les outils pour converger rapidement vers une solution adaptée au cas d'utilisation initial.

8.2 État de l'art

Afin de positionner nos travaux par rapport à la littérature existante dans le domaine de l'exploration d'espace de conception, nous nous proposons d'analyser les différents outils et méthodologies d'exploration sous trois angles

: l'exposition de l'espace à explorer, les métriques d'intérêt à considérer pour estimer la "qualité" d'une implémentation en particulier, et la façon dont les architectures en jeu sont comparées entre elles afin de sélectionner la ou les meilleure(s) solution(s).

L'exposition de l'espace de conception est un point clef pour définir des processus efficaces d'exploration. En effet, si l'espace considéré inclut trop de solutions, notamment des solutions qui sont trivialement sous optimales (comme cela peut être le cas si un outil est utilisé pour générer un tel espace), le processus d'exploration sera peu efficace, car il devra comparer trop d'architectures différentes (chaque comparaison pouvant s'avérer très coûteuse). D'un autre côté, si l'espace considéré est trop restreint, le développeur risque de ne pas considérer une solution qui serait "meilleure" que celles qu'il a étudiée, pour un cas d'utilisation donnée. Ainsi, les approches d'exposition d'espace existantes se positionnent sur un spectre continu de solutions, qui va d'espaces composés de variations qui sont toutes manuellement définies par le développeur (qui contrôlent donc totalement les circuits qui sont considérés) à des espaces où l'ensemble de ces variations est inféré par un outil de façon implicite (par exemple, dans le cas de la synthèse de haut niveau, les outils décident eux mêmes de comment les différentes primitives de programmation logicielle seront traduits dans le paradigme de description matérielle). La plupart des solutions existantes se situent en réalité quelque part sur ce spectre, et combinent donc des paramètres explicites pour contrôler les variations fines d'architecture, afin d'optimiser la performance des circuits générés, et des inférences implicites qui permettent aux développeurs de ne pas s'attarder sur certains aspects simples des circuits pour lesquels les heuristiques d'inférence fonctionnent bien. On notera ici la nécessité pour le développeur de pouvoir intervenir dans le processus d'exposition d'espace de conception, afin de pouvoir s'il le souhaite proposer des variations qui lui semblent pertinentes pour améliorer la qualité des architectures proposées, mais aussi le processus d'exploration en lui même (par exemple en supprimant certaines inférences si elle ne semblent pas intéressantes).[1]

Une fois l'espace de conception défini pour un cas d'utilisation donné, le développeur doit définir les métriques d'intérêts, ainsi que les méthodologies d'estimation adéquates. Ce faisant, il fournit en réalité les bases de fonctions

[1]Cette philosophie — s'appuyer sur l'expertise des développeurs et développeuses matériels pour améliorer leurs productivités — est d'ailleurs à la base de ces travaux, qui cherchent à proposer un flot de conception totalement personnalisable qui permette à ces utilisateurs de contrôler les aspects de conception qui leur semblent importants tout en reposant sur de simples inférences pour les décharger des tâches les plus simples et ainsi améliorer leurs expériences de conception.

objectifs à optimiser, en spécifiant une façon de mesurer la qualité des architectures de façon automatique. Ces métriques sont, pour les plus usuelles d'entre elles, omniprésentes dans le travail quotidien des développeurs — il peut s'agir de l'utilisation des ressources disponibles, de la fréquence d'opération ou encore de la bande passante utilisée par les architectures considérées. Nous proposons également de considérer des métriques plus "exotiques", comme la qualité de résultat (qui peut se définir, par exemple, par le taux d'erreur induit sur les résultats d'un circuit par rapport à une référence) ou encore la durée de vie des variables ou la sécurité des architectures. L'accent est particulièrement mis, encore une fois, sur l'intérêt qu'ont les développeurs à pouvoir définir eux même à la fois les métriques considérées, mais aussi les techniques pour estimer ces métriques, ce qui peut notamment influer sur la précision de ces estimations et sur leur rapidité.

Enfin, nous proposons de considérer un dernier aspect clef dans la définition de méthodologies d'exploration d'espace de conception : la stratégie d'exploration. Cette stratégie désigne l'algorithme qui va être utilisé lors du procédé d'exploration afin de comparer les différentes implémentation entre elles, cet algorithme étant le plus souvent implémenté dans l'outil d'exploration, mais correspond également au processus d'optimisation bien connu des concepteurs numériques, qui le plus souvent itèrent sur la description matérielle jusqu'à obtenir une solution qui satisfassent leurs contraintes et objectifs. Pour ce troisième point, nous appuyons notre analyse sur différentes taxonomies de stratégie d'exploration existantes dans la littérature. Tout d'abord, nous considérons trois types de stratégie possible pour explorer un espace de conception : les stratégies hiérarchiques, les stratégies itératives et les stratégies séquentielles. Bien que les stratégies hiérarchiques et itératives semblent très prometteuses dans le domaine de l'exploration, notamment pour leurs capacités de passage à l'échelle, nous nous sommes concentrés sur les stratégies séquentielles dans le cadre de ces travaux, c'est à dire des algorithmes qui s'appuient sur un ensemble d'étapes consécutives qui opèrent sur l'espace de conception afin de répondre à un cas d'utilisation. La deuxième taxonomie considérée dans cette approche classifie ces algorithmes en quatre catégories : les méta-heuristiques, les heuristiques dédiées, les méthodes basées sur l'apprentissage supervisé et les méthodes basées sur des analyses de graphe. Ces différentes catégories mettent en avant l'immense variété de possibilité qui s'offrent au développeur qui souhaite explorer de façon efficace son espace de conception, et appuient une fois de plus notre proposition de méthodologie générique et flexible reposant sur les connaissances et l'expertise du développeur pour guider le processus d'exploration.

Dans cette analyse de la littérature, nous montrons donc que le domaine de l'exploration d'espace de conception est un domaine très vaste, qui comporte de nombreuses contributions et méthodologies. Cependant, l'analyse de l'état de l'art en trois point qui est proposée est, au vue de nos connaissances actuelles, novatrice dans son approche du problème d'exploration, ces considérations d'exposition, d'estimation et d'exploration n'étant usuellement pas décorrélées mais plutôt considérées comme un tout. De plus, cette analyse met en avant l'intérêt d'une méthodologie d'exploration qui soit flexible et basée sur l'expérience des développeurs, là où la plupart des initiatives actuelles cherchent plutôt à permettre à des utilisateurs non experts de réaliser cette tâche d'exploration, et ce le plus souvent au coût de la performance des circuits générés.

8.3 Métriques et méthodologies d'estimation

Dans ce chapitre, nous explicitons l'intérêt de la tâche de définition des métriques d'intérêt dans le cadre de l'exploration d'espace de conception. En effet, il est nécessaire de définir une sorte de relation d'ordre entre les différentes implémentations qui composent cette espace : factuellement, cela revient à expliciter les critères de qualité qui sont à considérer pour définir si une implémentation est préférable à une autre dans le cas d'utilisation donné.

Il est à noter que cette étape de définition explicite des métriques d'intérêt s'intègre encore une fois dans notre approche flexible du problème de l'exploration d'espace de conception : dans la plupart des outils, ces métriques sont au mieux définies "en dur" dans les outils (et l'utilisateur peut choisir laquelle il souhaite considérer), et au pire ne sont même pas paramétrables. A contrario, nous faisons le choix dans ce travail de laisser au développeur la liberté de définir les métriques d'intérêts pour son cas d'utilisation, mais aussi de définir les méthodologies d'estimation qui seront utilisées pour définir ces métriques.

En effet, après avoir défini une métrique de qualité pour les implémentations qui composent l'espace à explorer, il est nécessaire de préciser comment cette métrique doit être calculée : cela peut être par des formules analytiques, des analyses de la représentation interne du circuit dans le flot de compilation Chisel, ou encore l'interfaçage avec des outils externes. Par exemple, pour estimer la consommation de ressources d'une implémentation, on peut décider d'utiliser une simple estimation (rapide) basée sur le comptage des primitives de calcul qui apparaissent dans la représentation du circuit, mais on peut aussi utiliser les résultats d'un outil de synthèse logique (bien plus long, mais bien plus précis) afin de fournir à

l'outil d'exploration des métriques plus réalistes. Ainsi, nous proposons une méthodologie générique à la fois de définition et d'estimation des métriques d'intérêt, qui s'appuie sur une interface de programmation flexible qui permet d'intégrer ces estimateurs à différents niveaux d'abstraction, en jouant sur un compromis implicite entre la qualité d'un estimateur et sa rapidité.

Nous proposons également dans ce chapitre l'implémentation de plusieurs estimateurs comme autant de preuves de concept de l'utilisabilité de cette approche. Nous implémentons ainsi tout d'abord un estimateur rapide de la consommation de ressource FPGA, basé sur une analyse hiérarchique de la représentation intermédiaires des circuits générés par Chisel. Un estimateur similaire pour le calcul des chemins critiques des circuits est proposé, mais la difficulté de l'approche ainsi que sa complexité algorithmique résultent en une solution qui est à la fois trop imprécise et trop lente pour être utilisable en pratique. Afin de fournir des estimations réalistes, nous proposons une interface simple pour les logiciels de synthèse logique propriétaire, permettant d'estimer à la fois la consommation de ressources et les chemins critiques d'un circuit. Cette estimateur est donc relativement lent, mais très utile dans la mesure où les résultats de ces outils de synthèses devront être considérés tôt ou tard dans le processus de conception d'un circuit. Enfin, nous implémentons un estimateur plus original, qui consiste à estimer la qualité de service des différents circuits générés. Cette métrique d'intérêt, qui peut s'avérer cruciale dans des domaines tel que celui des calculs approximés, est rarement considérée dans les flots d'exploration classique, et cette proposition permet de démontrer l'intérêt de la définition de métriques personnalisées par le développeur, pour des cas d'utilisation très particuliers. L'estimation de la qualité de service est basée sur deux approches distinctes : le développeur peut fournir un modèle analytique pour estimer le taux d'erreur introduit dans le circuit par rapport à ses paramètres de génération, ou peut utiliser une approche empirique, basée sur les résultats de simulations RTL des circuits, afin d'estimer ce même taux d'erreur.

Dans ce chapitre, nous proposons donc à la fois une approche générique pour la définition et l'estimation de métriques d'intérêt dans le contexte de l'exploration d'espace de conception, mais aussi des estimateurs "preuves de concept" qui s'intègrent dans le flot de conception Chisel à travers l'interface de programmation que nous avons définie.

8.4 Exploration d'espace de conception

Dans ce chapitre, nous proposons un approche complémentaire à celle de la définition des métriques d'intérêt afin de définir des processus d'exploration d'espace de conception flexible, basés sur l'expertise des développeurs.

Dans un premier temps, nous proposons une méthodologie de conception de générateurs de circuits basée sur l'utilisation des langages de construction matérielle, dont le principe repose sur une analyse a priori de l'algorithme implémenté, couplé à une connaissance poussée dans le domaine de la conception numérique et sur la technologie cible. Cette méthodologie, appelée **méta conception**, est utilisée pour concevoir des générateurs de circuits dont l'exploration fait sens, c'est à dire dont les paramètres de génération ont un impact contrôlé sur les implémentations générées, et dont l'espace de conception induit comporte un maximum d'implémentations intéressantes, c'est à dire des implémentations non trivialement sous optimale pour le cas d'utilisation en jeu. La Figure 8.1 introduit les étapes qui constituent cette méthodologie de **méta conception**, de l'analyse de l'agorithme et de la cible à l'implémentation matérielle de la fonctionnalité et sa validation. En instrumentant un système d'annotation des générateurs produits, la méthodologie proposée permet en outre de spécifier l'ensemble des valeurs possibles pour chaque paramètre, exhibant ainsi l'espace de conception abordé ci-dessus.

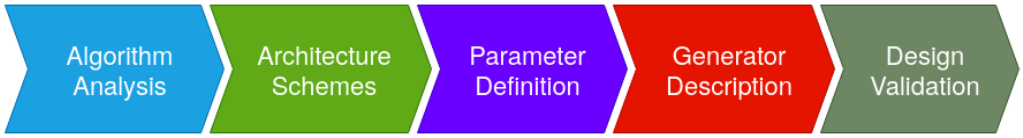


Figure 8.1: Méthodologie de méta conception

Dans un deuxième temps, nous nous intéressons au processus d'exploration de ces espaces de conception, et proposons la **méta exploration**, une méthodologie d'exploration qui se basent sur la **méta conception**. Cette méthodologie repose encore une fois sur l'expertise des développeurs afin de maximiser la flexibilité et la généricité de l'approche (Fig. 8.2). Nous discutons de l'utilisation du paradigme de programmation fonctionnelle[2] pour la définition des stratégies d'exploration d'espace de conception. Plus précisément, nous nous intéressons à une sous classe des stratégies d'exploration, les stratégies séquentielles, et proposons d'utiliser ce paradigme de programmation pour capturer la nature même de composition de ces stratégies. Ainsi, nous proposons de considérer les stratégies d'exploration comme une composition d'étapes basiques qui opèrent sur des espaces de conception, et qui peuvent dont être composées pour construire des stratégies plus complexes. Chaque stratégie basique repose à la fois sur les métriques d'intérêt à considérer, sur la façon à utiliser pour les es-

[2]Chisel étant un langage basé sur scala, proposant donc des fonctionnalités de programmation fonctionnelle

timer et sur comment comparer les différentes implémentations de l'espace de façon efficace (cela peut reposer sur une comparaison exhaustive de toutes les implémentations afin d'optimiser les critères d'intérêt, mais aussi sur des heuristiques moins complexes afin de réduire la durée des processus d'exploration produits ainsi).

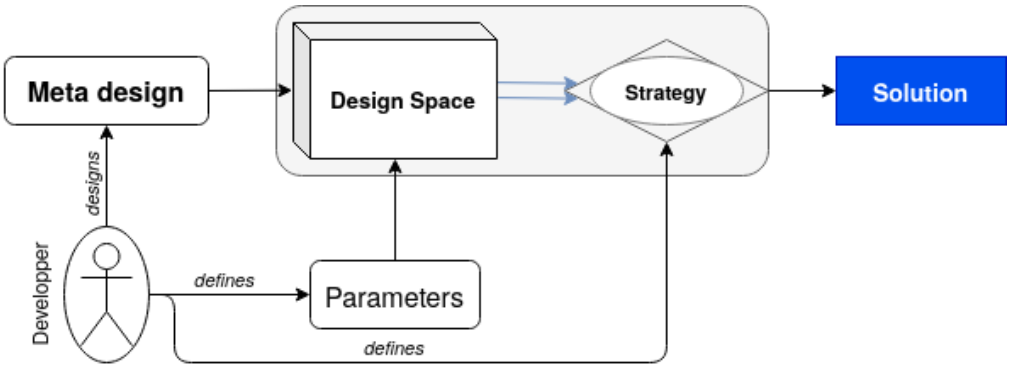


Figure 8.2: Méthodologie de méta exploration

Une formalisation mathématique de cette approche est proposée, afin de faire le lien entre la théorie méthodologique et l'utilisation de la programmation fonctionnelle pour l'implémenter, et plusieurs stratégies basiques sont définies comme preuves de concept. Parmi elles, des méthodes classiques d'opération sur des collections (les collections étant ici les espaces de conception à explorer) sont introduites, à savoir les méthodes *map* (application exhaustive d'une fonction sur l'ensemble des implémentations qui composent un espace), *sort* (un tri de l'espace basé sur l'application d'estimateurs sur l'ensemble des implémentations et sur une relation d'ordre explicite) et *prune* (un élagage de l'espace, lui aussi basé sur l'application exhaustive d'une fonction binaire qui défini quelles implémentations doivent être retirées de l'espace de conception, et quelles implémentations doivent rester). Nous proposons également deux stratégies plus complexes, basées sur une exploration des voisinages directs des implémentations dans les espaces de conception, afin d'accélérer les processus d'exploration pour les cas d'utilisation où une application exhaustive est infaisable dans un temps raisonnable Ces stratégies peuvent donc être utilisées pour réaliser respectivement un tri rapide et un élagage rapide des espaces de conception.

Dans ce chapitre, nous proposons donc une méthodologie duale d'exploration d'espace de conception qui repose sur l'utilisation de l'expertise des développeurs matériel, permettant de définir des processus d'exploration flexibles et paramétrables, reposant sur la programmation fonctionnelle afin de maximiser l'expressivité des utilisateurs.

8.5 Expérimentations et résultats

Afin de démontrer l'utilisabilité de la méthodologie d'exploration d'espace de conception présentée dans les chapitres précédents, un démonstrateur logiciel nommé **QECE** (*Quick Exploration using Chisel Estimators*) est proposé dans ce chapitre. De plus, la structure interne du logiciel ainsi que les détails intéressants d'implémentation sont décrits pour permettre aux utilisateurs de mieux appréhender les possibilités qu'un tel outil peut apporter au monde de la conception numérique.

Toujours dans le but de démontrer l'utilisabilité de la méthodologie proposée, un banc d'applications représentatives de l'utilisation des FPGAs en tant que dispositifs d'accélération matérielle est proposé. Chaque noyau applicatif composant ce banc a été développé en appliquant la méthodologie de **méta conception**, les détails d'implémentations pouvant être trouvés en annexe du manuscrit en langue anglaise. Des expérimentations d'estimation et d'exploration ont été menées sur trois noyaux de ce banc (*GEMM*, un algorithme de multiplication de matrices, *FFT*, un algorithme provenant du monde du traitement du signal, et *Black Scholes*, un modèle de calcul de la valeur d'une action basé sur la méthode de Monte Carlo), afin de mesurer la qualité des estimateurs introduits en Section 8.3, mais aussi des stratégies d'exploration proposées en Section 8.4.

Tout d'abord, nous quantifions l'erreur introduite par les différents estimateurs, démontrant que la méthodologie d'estimation de la consommation des ressources basée sur l'analyse de la représentation intermédiaire introduit une erreur importante par rapport aux résultats des logiciels de synthèses, mais que les métriques estimées peuvent tout de même être utilisée pour inférer la qualité des implémentations considérées en première approche. En revanche, nous démontrons que la méthodologie d'estimation des chemins critiques des circuits, basée elle aussi sur une analyse de la représentation intermédiaire, produit des résultats très éloignés de la réalité (à savoir, les résultats du logiciel de synthèse logique), très peu fiables, et en un temps qui peut être plus élévé que les outils propriétaires que l'on cherche à approximer. De ce fait, cette approche est laissée de côté comme piste d'amélioration du logiciel proposé, et les estimations de chemins critiques et de fréquences maximale d'opération utilisées dans la suite de nos travaux sont basées sur l'analyse des rapports des outils de synthèse logiques, bien plus fiables mais pouvant être coûteux en temps et en ressources de calcul. Enfin, nous démontrons l'utilisabilité de la méthodologie empirique d'estimation de la qualité de service des circuits sur un exemple simple (calcul d'un produit scalaire), afin de montrer comment l'instrumentation des différents moteurs

de simulation disponibles dans le flot de conception Chisel peuvent permettre d'estimer des métriques intéressantes, pour certains cas d'utilisation.

Après avoir abordé l'aspect qualitatif des différents estimateurs proposés, et démontré leurs utilisabilités, nous nous intéressons maintenant à démontrer l'intérêt de l'approche fonctionnelle pour résoudre le problème de l'exploration d'espace de conception, en proposant et en comparant différentes stratégies de **méta exploration** comme définies dans le Section 8.4. Tout d'abord, nous comparons trois stratégies d'exploration sur le noyau applicatif *GEMM*, démontrons qu'une approche basée sur l'utilisation de l'expertise du développeur peut permettre de réduire d'un facteur 4 le temps passé pour explorer un espace restreint (42 implémentations) de façon efficace, comparé à une solution par "force brute" consistant à synthétiser toutes les implémentations disponibles. Nous démontrons ensuite, sur le noyau applicatif de Transformée de Fourier Rapide (*FFT*), qu'une des limites de cette approche d'exploration est la possibilité d'exprimer des relations d'ordres claires entre les implémentations qui composent l'espace de conception, c'est à dire utiliser les métriques d'intérêts disponibles pour définir comparer la qualité des implémentations dans un cas d'utilisation donné. Sans modèle applicatif adéquat, il est impossible de comparer, par exemple, des noyaux qui considèrent différentes tailles d'élements dans les calculs, puisque des éléments plus grands vont augmenter la qualité de service des circuits générés, mais également la consommation de ressources — c'est donc au développeur de spécifier exactement cette relation d'ordre, qui ne peut être implicite. Ainsi, sur ce noyau applicatif, nous démontrons que les stratégies avancées d'exploration peuvent s'avérer moins performantes (en terme de temps) que la stratégie par "force brute", dans le cas où le modèle applicatif fourni est trop faible pour exposer des espaces de conceptions suffisamment large pour bénéficier de cette approche. Enfin, et pour palier ce problème, nous proposons une stratégie d'exploration de l'espace de conception des noyaux *Black Scholes*, basée sur une approche en deux temps: tout d'abord, l'utilisation d'une approche empirique pour estimer la qualité de service (l'erreur introduite par l'implémentation matérielle par rapport à une référence logicielle) et discriminer toutes les implémentations qui introduisent une erreur trop importante, avant de parcourir les implémentations restantes pour sélectionner celle qui proposent le meilleur débit possible, pour la carte FPGA cible. Une telle stratégie peut être définie en une dizaine de ligne dans le logiciel **QECE**, démontrons à la fois l'intérêt et la concision de l'approche fonctionnelle pour l'exploration d'espace de conception.

Dans ce chapitre, nous démontrons l'utilisabilité de la méthodologie de **méta exploration**, contribution principale de cette thèse, à travers l'usage d'un logiciel démonstrateur (**QECE**) que nous avons développé, et de mul-

tiples expérimentations sur des noyaux applicatifs également développés par nos soins. L'accent est mis sur l'utilisation de l'expertise des développeurs matériels pour proposer des stratégies intelligentes qui convergent rapidement pour les cas d'utilisation considérés.

8.6 Conclusion

Dans ce travail de thèse, nous nous intéressons au paradigme émergent des langages de construction matérielle, dont Chisel est un exemple, afin d'améliorer la productivité des développeurs matériel. Plus précisément, nous proposons une approche originale pour résoudre le problème de l'exploration d'espace de conception, qui se base sur une méthodologie innovante de conception couplée à l'utilisation du paradigme de programmation fonctionnelle afin de définir des stratégies d'exploration intelligentes, car définies par les développeurs eux mêmes.

Nous démontrons l'utilisabilité d'une telle méthodologie à l'aide d'un logiciel développeur, nommé **QECE**, que nous avons développé pour l'occasion, qui propose une bibliothèque de méthodologies d'estimation de métriques d'intérêt pour les circuits considérés, et une bibliothèque de stratégies basiques d'exploration qui peuvent être composées pour construire des stratégies plus complexes et adaptables aux différents cas d'utilisation. Cette approche permet une grande flexibilité d'utilisation, afin d'être réutilisable et adaptable à de nouveaux cas d'utilisation, améliorant ainsi la productivité des concepteurs numériques.

APPENDIXES

A

Chisel Basics

This Appendix provides motivations about technological choices that were made for this thesis, as well as some insights about basic usage of **Chisel**, the chosen *Hardware Construction Language* (**HCL**).

Doing so, we aim at providing more information about the usage of **HCLs** for hardware development. A particular focus is put on highlighting some notable differences with the standard *Hardware Description Languages* (**HDL**) such as *VHDL* or *verilog*, in order to help readers that may be familiar with such languages to comprehend the improvements that this paradigm enables.

Motivation

Hardware Construction Languages (**HCL**) are a novel paradigm which enables the development of parametrized hardware generators instead of dedicated hardware accelerators, as described in Section 2.1.4. Such initiatives are based on high level languages such as **python** [LZB14], **Haskell** [BKK+10] or **scala** [BVR+12], which enable leveraging novel software constructs for hardware development.

In this context, we chose to base this work on the usage of *Constructing Hardware in a Scala Embedded Language* (**Chisel**), an **HCL** developed at Berkeley since 2012. Since its creation, **Chisel** has been widely adopted by both academic and industrial worlds, with initiatives such as the Rocket Chip generator [AAB+16], an in-order generator of RISC-V cores, the *Berkeley Out-of-Order Machine* (**BOOM**) generator [CPA15], an out-of-order version, or the development of Google *Tensor Processing Unit* (**TPU**) [LT18].

Moreover, Chisel has also been used as a basic tile of the Chipyard project [Ber21b], a set of tools used to create an agile framework for hardware development, along with tools such as Hammer [WIS+18], which decouples physical design concerns, logical design concerns, tool concerns and technology concerns, and Diplomacy [CTL17], which enable automatic negotiation of parameters when generating complex Systems-on-a-Chip (SoC). It proves that Chisel ever growing community works to ever improve the language features, and that it can be used in wild, complex initiatives.

Basic usage

To begin with, a simple cheat sheet is available in the project, in order to give some insights about available primitives for hardware generation [Ber21a].

However, as an exhaustive comprehension of available constructs is not mandatory to understand the content of this work, examples of **Chisel** usage will be provided thereafter that should be sufficient to comprehend the improvements brought by **HCL** usage.

Chisel compilation flow

The **Chisel** *Hardware Construction Framework* (**HCF**) structure is based on a separation of concerns inherited from the structure of software compilers, where the entry point (**frontend**) and the tool output (**backend**) are separated by an *Intermediate Representation* (**IR**). Such architecture allows to implement support for various input and target languages, while exhibiting a modular structure for the developers.

An example of a **Chisel**-based generation of a parametrized increment module is introduced in Figure A.1, where two simple parameters are exhibited: the width of the module output, and its initialization value.

Three main steps are taken during the generation of this module:

1. The **Chisel** parameters are resolved during the elaboration, meaning that the parameters are fixed for the remaining of the flow. This is similar to the *verilog* elaboration process where the explicit module parameters are resolved and propagated in the circuit description.

   A **high level FIRRTL** representation is here generated as a first **IR**.

2. Multiple transforms (including verifications such as combinatorial loop detection) are run over the **FIRRTL** representation, progressively lowering the abstraction level, until a **low level** representation is produced.

   At the end of this process, all the data types and widths have been resolved, the conditional statements have been replaced, and it is guaranteed that every component is connected exactly once.[1]

   One can remark that, at this step, the parameters are fixed (*e.g.* the width of the output is fixed to 8 bits) and implicit control signals — *i.e.* the clock and the reset — have been generated automatically.

[1] As the example module is very simple, no significant differences are exhibited between **high** and **low level FIRRTL** representations.

3. The last step is a simple translation, where the **low FIRRTL** is translated to *verilog*.

   As the abstraction level remains the same, this process is quite straightforward, with the simple goal of providing an output language that can be used by most tools in the industry.

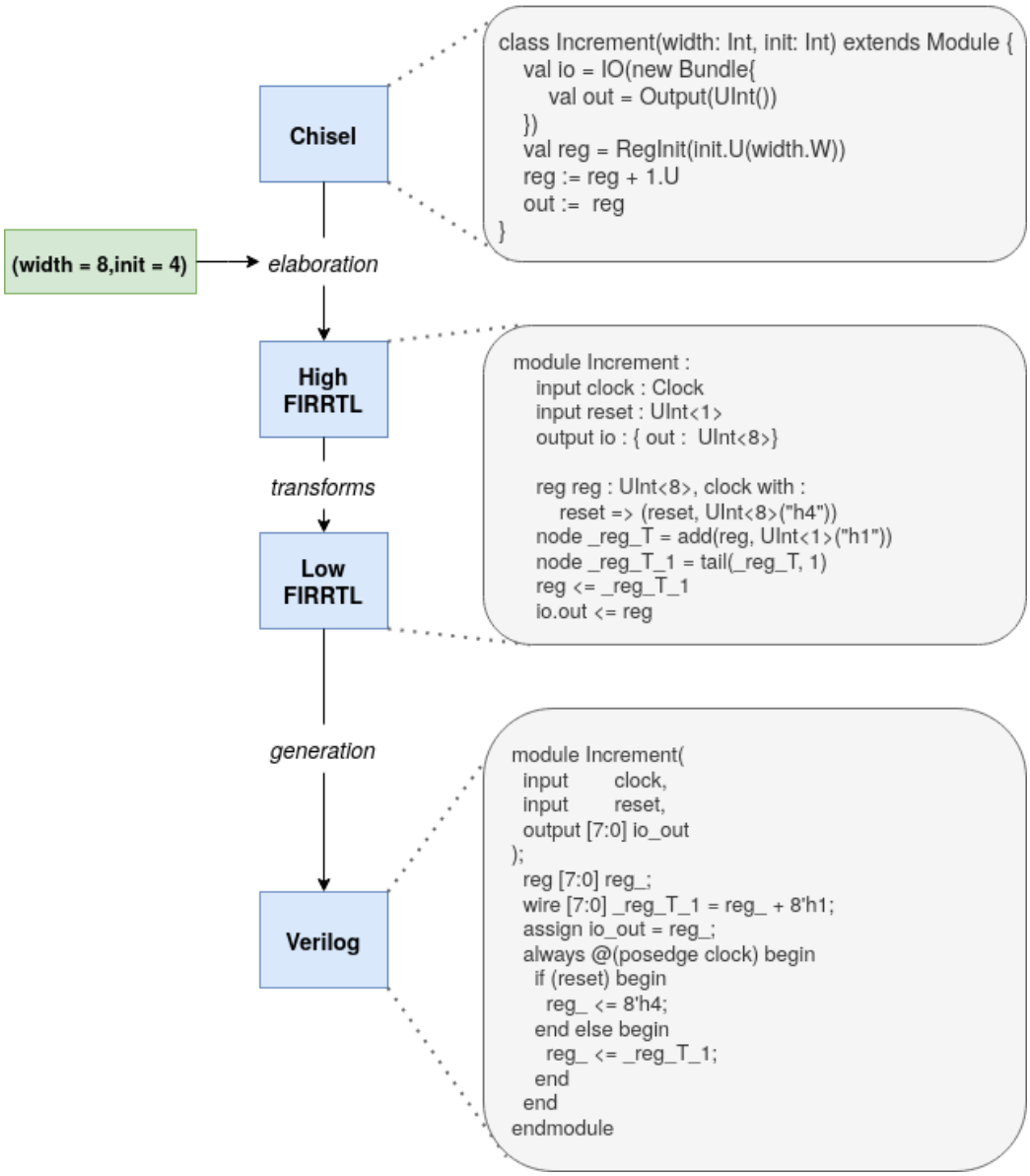


Figure A.1: Implementing an increment module using Chisel

Exposing high level parameters at constructor level

One of the main differences of **HCLs** with standard **HDL** is the possibility to leverage *Object-Oriented Programming* (**OOP**) features — such as module inheritance — for hardware construction. While evolutions of those languages have come with a way to define simple `parameter`s for module generation, as well as `generate` features which enable loop based generation of hardware, it remains difficult to leverage complex parameters that could bring a lot for hardware generation.

A simple example of a **Chisel** generator is introduced in Listing A.1, implementing a type parametric adder generator. It leverages a **scala** constructor for module definition, allowing to define how each constructor parameter is used for adder generation — meaning that the type of `op1`, `op2` and `res` ports will be defined at elaboration time only. This code can then be used to generate variations of an adder module, allowing to instantiate an adder on 32 bit wide unsigned integers (line 14.) or on 8 bit fixed point numbers (line 15.) with the same description.

Used parameter is here defined using the template feature from **scala**, using a `T` type for generation. This type is then checked at elaboration, to verify that it does implement the `Data` type (the basic type for any data in **Chisel**) as well as the `Num` trait (which ensure that a given type does implement basic arithmetic operations such as addition and multiplication). Doing so, one can leverage *polymorphism* to generate modules operating on any type that fits the requirements, enabling to build a library of easy to use components that can be adapted to multiple use cases.

While this example is intentionally kept simple for the reader to understand the provided code, it should be noted that providing a similar module using standard **HDL** would require either to copy and adapt the code for every possible type, or search and replace type specific operations in the code. Moreover, more complex parameters can be defined, such as high-order functions which can be used as module parameters to change the behaviour of a circuit in functional fashion.

Using functions as block generators

Another interesting feature of **Chisel** is the ability to generate hardware using functions to provide block abstractions — in other terms, calling a function can result in **RTL** code generation at the end of the elaboration.

This includes **constructor methods**, meaning that the instantiation process can be leveraged to generate behavioural description of circuit. A simple example is provided in Figure A.2, where a *verilog* description of a

```
0 import chisel3._
1 import chisel3.experimental.FixedPoint
2
3 class Adder[T <: Data with Num[T]](tpe: T)
4     extends Module {
5     val io = IO(new Bundle({
6         val op1 = Input(tpe)
7         val op2 = Input(tpe)
8         val res = Output(tpe)
9     })
10
11    io.res := io.op1 + io.op2
12 }
13
14 val uintAdder = new Adder(UInt(32.W))
15 val fpAdder = new Adder(FixedPoint(8.W, 3.BP))
```

Listing A.1: Type parametric adder generator in Chisel

register (List. A.2) is compared to its **Chisel** counterpart (List. A.3).

As can be observed here, the semantic for describing a register in standard **HDL** is based on simulation needs, as those languages were originally designed for simulation purposes instead of hardware description. One then needs to declare a logic signal (line 6.), and a process sensitive to the clock signal to either update the value, or reset it (lines 7-11).

While every hardware developer has grown used to such way to describe registers, it remains a verbose way to describe a basic component that will be instantiated hundreds of times in a single circuit. Moreover, this semantic is quite confusing for beginners, as *verilog* exposes a `reg` keyword for signal declaration, but that does not actually differ from the `wire` keyword from a semantic point of view.[2]

On the other hand, **Chisel** leverages block generation through function calls, enabling users to simply instantiate basic classes for register description (line 6.). Once it is done, assigning a signal to the register value (*i.e.* `myRegister` value at line 6.) will change the next value (as does changing the value on port `D` in List. A.2), while using the variable as a right-hand operand (line 7.) is equivalent to reading the content of the register (`Q` port).

[2]The only way to describe a register is to use specific patterns such as the one in Listing A.2, that will be recognized and translated by synthesis tools to instantiate actual hardware registers.

```
0  module register (parameter WIDTH) (
1      input clk,
2      input reset,
3      input [WIDTH-1:0] D,
4      output [WIDTH-1:0] Q);
5
6      reg [WIDTH-1:0] myregister;
7      always @(posedge clk) begin
8          if (reset)
9              Q = 0;
10         else
11             Q = D;
12 endmodule
```

Listing (A.2) Behavioural description of a register in Verilog

```
0 class MyRegister[T: Data](tpe: T) extends Module {
1     val io = IO(new Bundle{
2         val dataIn = Input(tpe)
3         val dataOut = Output(tpe)
4     })
5     // register width is inferred from dataIn type
6     val myRegister = RegNext(dataIn)
7     dataOut := myRegister
8 }
```

Listing (A.3) Building a register using Chisel constructs

Figure A.2: Building a register: HDL *vs* HCL

This simple semantic is less error prone, as it does not rely on a copy of a particular pattern for instantiation, and allowing easier instantiation of basic component is a key feature to allow developers to spend more time on actual design problems, thus improving their productivity. We also see here that using high level parameters as exposed in previous section can enable to build highly reusable component generators, by defining a type parametric register module for example.

Remark: Both `clk` and `reset` signals are not provided for register instantiation in the **Chisel** example. In fact, those signals are implicit in any class inheriting from `Module`, the basic class for hardware circuits. They are used for instantiations of components that require them, and are being propagated to every instantiated sub module in order to expose coherent time zones to users. If needed, they can be overwritten manually to build different time zones — *e.g.* for circuits using multiple clock domains.

Leveraging high level constructs for generation

The last feature that we will consider here is the usage of high level functionalities for hardware generation.

As **Chisel** is based on **scala** language, features from this language can be used to define module generators and bring more expressivity for developers.

```
0  class DotProduct(width: Int, nElem: Int) extends Module {
1      val dataType = UInt(width.W)
2
3      val io = IO(new Bundle{
4          val op1 = Input(Vec(nElem, dataType))
5          val op2 = Input(Vec(nElem, dataType))
6          val out = Output(dataType)
7      })
8
9      io.out := ((io.op1 zip io.op2)
10         .map{ case (a, b) => a * b })
11         .reduceTree(_ + _) // use balanced tree
12 }
```

Listing A.4: Building a dot product kernel using Chisel

Listing A.4 introduces a **dot product** generator parametrized by elements width and number. This generator thus accepts two vectors of elements, which elements are multiplied two by two, before being summed to produce the output. Such operation can be captured using the **map-reduce** paradigm, as it would be done in a software implementation leveraging functional programming, and hardware designers should also benefit from such programming patterns.

An example of the target architecture is provided in Figure A.3, to help the reader to understand the gap between the final circuit, and what the designer actually requires to describe using either **Chisel** or *verilog*.

Lines 9. to 11. expose how **Chisel** can be leveraged to build a generator for such implementations: input vectors are **zipped** — *i.e.* elements of each vectors are paired — before multiplication is applied to each pair. Resulting vector is then reduced through add operations, using a binary tree structure to force elaboration to produce a balanced adder tree.

A *verilog* counterpart is proposed in Listing A.5, to exhibit the complexity of building such generator in a traditional **HDL**. To do so, we require to expose a multidimensional array of elements and populate it according to some

```
module dot_product #(
    parameter NELEM,
    parameter WIDTH,
  ) (
    input [N-1:0] [WIDTH-1:0] op1,
    input [N-1:0] [WIDTH-1:0] op2,
    output        [WIDTH-1:0] out,
  );

  localparam  STAGES = clog2(NELEM);
  localparam  NPADDED = 2**(STAGES);

  wire  [STAGES:0] [NPADDED-1:0] [WIDTH-1:0] tab;

  generate
    // Init loop with mul (required padding for NELEM not power of 2)
    for (i=0; i<NPADDED; i=i+1) begin:init
      // 0 padding is fine for add reduce
      assign tab[0][i] = (i < NELEM) ? op1[i] * op2[i] : '0;
    end

    // main reduce loop with tree
    for (stage=0; stage<STAGES; stage=stage+1) begin:stages
      for (couple=0; couple<2**(STAGES-stage-1); couple=couple+1)
        begin:couples
          localparam first  = couple * (2**(stage+1));
          localparam second = first + (2**stage);
          assign tab[stage+1][first] =
            tab[stage][first] + tab[stage][second];
      end
    end
  endgenerate

  assign out = tab[STAGES][0];
endmodule
```

Listing A.5: Dot product generic implementation in Verilog

specific patterns on the indexes, through the `generate ... for` construct. This pattern is hence recognized by the synthesis tool, which eliminates the useless elements of the array, and build the correct adder tree.

A last remark can be done on the differences between those two paradigms: while changing the data path of the *verilog* module — *e.g.* to add registers after multipliers and adders — requires to modify the whole code, it can be done in a functional way in the **Chisel** description.

This is possible because both `map` and `reduceTree` functions accept high-order functions as parameters to define which operations are to be performed, which is known in the software world as **functional programming**. In fact, the `reduceTree` construct even accepts a second function as parameter, to define what to do on non balanced trees (when the number of elements is not a power of two), allowing to define a simple delay using a `RegNext` in the case of pipelined additions, for instance.

This final example thus exhibits how the introduced features enable to build efficient hardware generators which are easier to design, understand and adapt to new use cases.

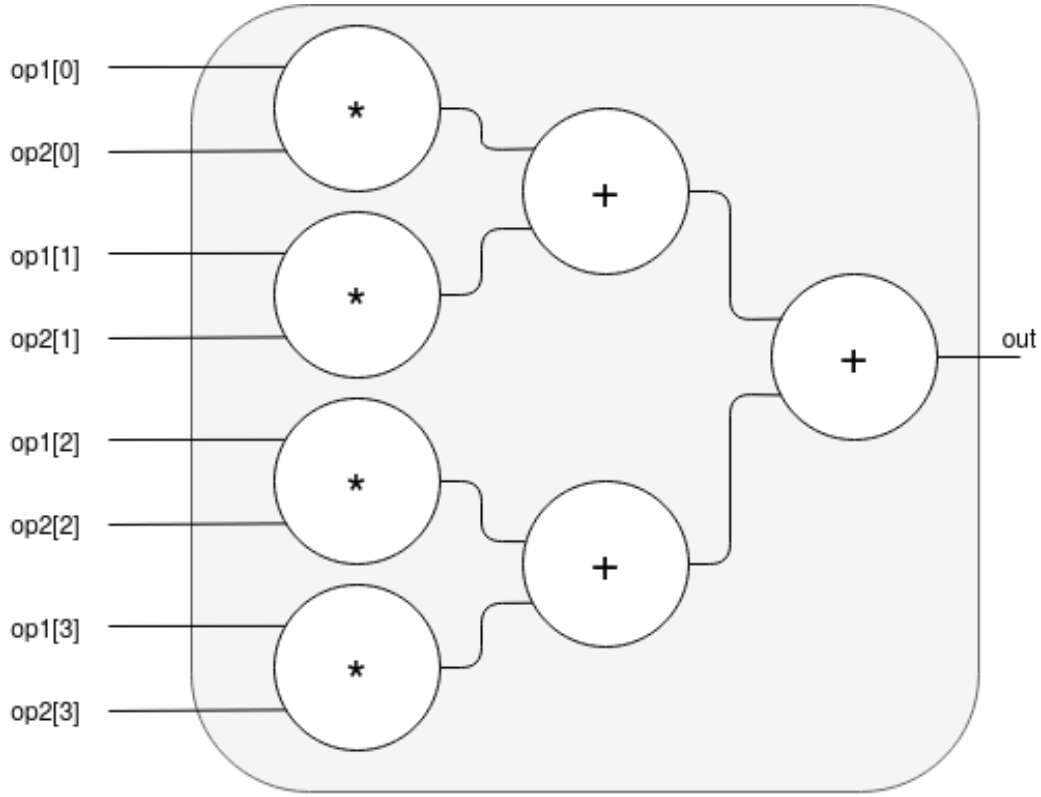


Figure A.3: Example of a dot product architecture to generate (*using vectors of 4 elements*)

B

Benchmark Composition

In order to analyse how the defined methodologies can be leveraged to improve the life of hardware developers, we implemented **7 computation kernels** using **Chisel**, which are introduced in Table B.1.

Each kernel has been developed by applying the **meta design methodology** introduced in section 5.1.1, and this appendix also exposes — in addition to the benchmark description — which considerations were taken into account, for each kernel, in order to provide an explorable meta design from a prior algorithmic analysis.

This benchmark is proposed as an open-source project [FMR21b].

Kernel model

The kernels were built to be integrated in a simple programming model, by exposing a unified interface. To do so, we defined a **Chisel Role** and **Shell** based architecture, as can be seen in emerging *Field-Programmable Gate Array* (**FPGA**) projects that aim at separating the accelerators implementation from their interfaces [CCP+16].

The proposed interface is kept simple, as **Role** simply exposes a bi-directional **Ready/Valid** interface with a parametric *bandwidth* (Fig. B.1), that can be then be integrated in more or less complex **Shells**, ranging from simple mapping of this *Input/Output* (**IO**) to the communication *Intellectual Property* (**IP**) that may be available on the target board, to complex communication protocols.

However, such structure is quite simple, and evolutions should consider integrating multi-lane communications for configuration purposes — which are currently done using the same **IO** bus, meaning that the accelerators must consider configuration in their data communication protocols.

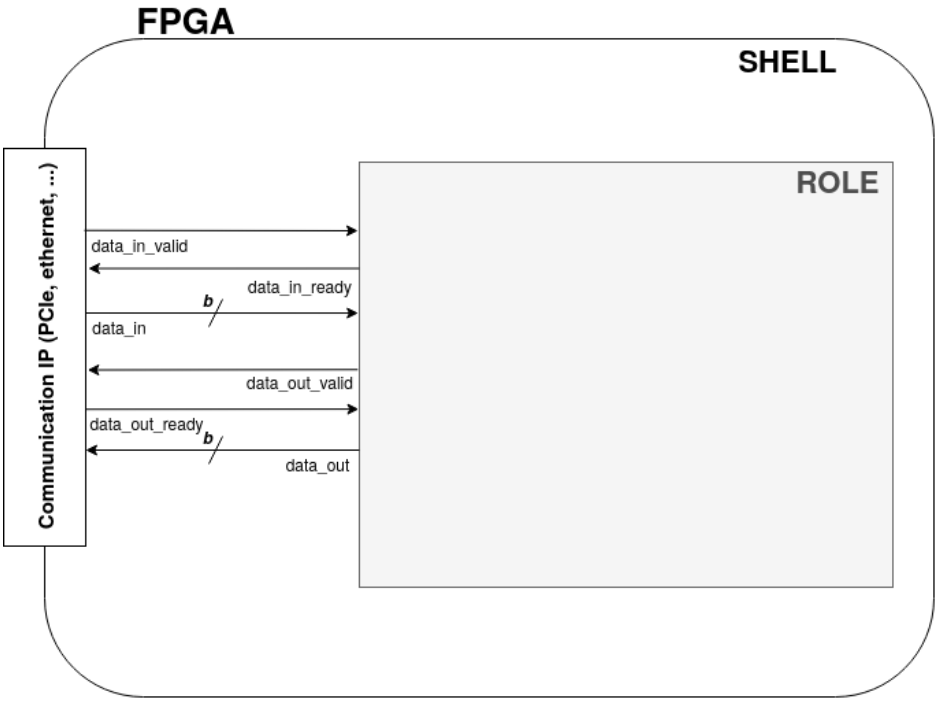


Figure B.1: Simple Role and Shell model used

Dot Product

The dot product generator has been developed as a simple example, and does not expose a heavy design space. Moreover, an example of the target architecture is introduced in Figure A.3.

Given two vectors a and b of dimension n, we compute:

$$c = \sum_{i=0}^{n-1} a_i * b_i \tag{B.1}$$

The *Vector width* is used to define the algorithmic complexity of the implementation, while both *dynamic* and *precision* define the element type to operate on. In order to take advantage of an **FPGA** implementation, we aim at exploiting the parallelism of the algorithm, while keeping the resource usage under given constraints. The dot product can easily be expressed in a functional way, using the **Map-Reduce** pattern, implying we can extract a maximum parallelism level of n, by performing all the multiplications in parallel before performing reduction through additions. We thus expose *parallelism* as a parameter, allowing the user to easily increase the throughput at the cost of functional unit replication.

General Matrix Multiply

The *General Matrix Multiply* (**GEMM**) algorithm is a generalization of the matrix multiply algorithm.

Given three matrices A, B and C in $\mathcal{M}_n$, the set of the matrices of dimension $n \times n$ (that we consider square for simplification) and two values α and β, we compute:

$$C = \alpha \cdot A \times B + \beta \cdot C \tag{B.2}$$

A meta design for the **GEMM** algorithm has been introduced in prior work [FMR20] — from which Equation B.3 is extracted — and was manually explored in order to demonstrate how both **meta design** and **meta exploration** can be leveraged to increase designers productivity [FMR21a].

The **GEMM** generator has been developed in order to maximize the **IO** usage, targeting a temporal behaviour as represented in Figure B.2. Such prior analysis enables to build a generic generator that uses both *Block Random-Access Memory* (**BRAM**) and *Multiply and Accumulate* (**MAC**) to build adaptable designs that will compute partial results on the fly, resulting in a theoretical maximal usage of the bus.

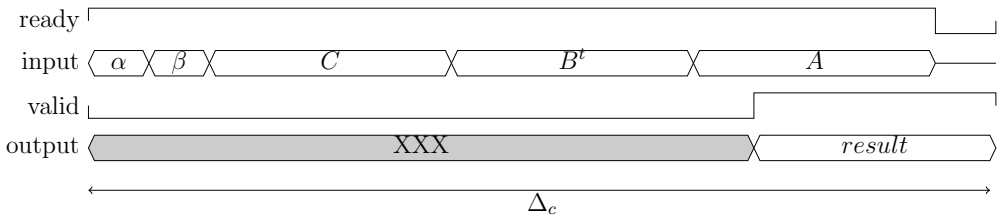


Figure B.2: Targeted chronogram for an efficient GEMM implementation

Equation B.3 introduces the theoretical throughput (in $operation \cdot second^{-1}$) of the designs generated from this description, and is used to define a cost function to be maximized while exploring the **GEMM** design space. One can remark that doing this enable to normalize a cost metric through different matrices dimensions, as performing **one GEMM computation** over $\mathcal{M}_n$ is comparable to performing **eight** $\mathcal{M}_{\frac{n}{2}}$ **computations**.

$$T_{GOp/s} = \frac{f}{\Delta_{cycle}} = \frac{2 \times n^3 \times f \times b}{3 \times n^2 \times e} \tag{B.3}$$

As a result, we define three parameters for **GEMM** generation: the *bus bandwidth*, which defines the capacity of the **IO** bus, the *element bit width*, which defines the size of each element in the considered matrices — meaning that $\frac{bandwidth}{bitwidth}$ element can be sent and received each cycle — and finally the *dimension* (or n) of the matrices in $\mathcal{M}_n$.

Fast Fourier Transform

The *Fast Fourier Transform* (**FFT**) algorithm is used in signal processing to provide frequency analyses from temporal data.

Given $x_0, ..., x_{n-1} \in \mathbb{C}$, **FFT** is computed using the following formula:

$$f_j = \sum_{k=0}^{n-1} x_k e^{-\frac{2\pi i}{n} jk} \tag{B.4}$$

The **FFT** generator was built to maximize the **IO** bus usage, resulting in a pipelined implementation inspired from Gerez [Ger12].

The pipeline is based on the *Radix-2 Multi-Path Delay Commutator* (**R2MDC**) technique [RG75], using multiple Radix-2 stages to reduce the **FFT** problem size by 2 at each stage, in a *divide and conquer* fashion. We use the *Decimation In Frequency* (**DIF**) mode in order to consume input data (temporal samples) in a **FIFO** fashion, resulting in a need to reorder the output data (frequency samples) at the end of the pipeline. This is done by using a **Ping Pong buffer**, which enables to provide the frequency data in a coherent order, even if the **DIF**-based implementations produce out-of-order frequency samples — moreover, using another **Ping Pong buffer** on the **R2MDC** inputs enables to exploit them at 100%. The **Twiddle factors** — *i.e.* the trigonometric coefficients — are computed *a priori* and stored in *Read-Only Memories* (**ROM**) to fasten the computations.

This pipelined implementation allows to maximize the **IO** bus usage as we consume input data on the fly, while maximizing the resource usage.

The **FFT** generator relies on three parameters: the *parallelism* level, which defines the number of lanes used in the **R2MDC** model — *i.e.* the number of inputs that can be absorbed in one cycle in the generated accelerator — the *element bit width*, which defines the size of each complex element used for computation, and the *size* of the **FFT** problem being solved. As for the *element bit width*, the **FFT** uses **Fixed Point** representation for the computations, and one could want to explore this representation by defining both *dynamic* and *precision* parameters. However, as we do not focus *Quality of Service* (**QoS**) based exploration on this kernel, we chose to define the data representation using only one parameter — we use $\frac{bitwidth}{2}$ as values for both *dynamic* and *precision*.

We can remark that, in contrast with the **GEMM** implementations, two **FFT** implementations of different sizes cannot be compared easily, as the first one cannot be expressed by composing instances of the second one. We here solve different applicative problems, hence the developers either need to

specify which size to use, or to allow the exploration process to make this decision through applicative information (*e.g.* **QoS**).

Finite Impulse Response Filter

Finite Impulse Response Filter (**FIR Filter**) is a standard digital processing algorithm, based on the application of a finite number of coefficients to temporal samples in order to modify an input signal.

The filter response is computed using a discrete convolution between an input signal $x[t]$ and the function represented by $c_k, k \in [\![0, n-1]\!]$ coefficients:

$$f(t) = \sum_{k=0}^{n-1} c_k \times x[t-k] \tag{B.5}$$

Once again, we aim at exploiting the **IO** bus at its maximal capacity by absorbing the whole input bandwidth as temporal samples at each cycle. To do so, a structured buffer is built to enable windowed accesses to the buffer data: if we consider that k elements are presented on the input bus at each cycle, then those k elements $e_i, i \in [\![0, k-1]\!]$ are fed to a **FIFO**. On read, this structure enables to access k different vectors for inputs ($[\![e_i, e_{i+k-1}]\!], i \in [\![0, k-1]\!]$), meaning that we can compute k successive response of the filter $f(t+i), i \in [\![0, k]\!]$ in parallel. Doing so, we hence built a parametrizable, fully pipelined **FIR Filter** generator that takes the best of the available **IO** capacities.

So built **FIR Filter** generator uses three parameters: the *bus bandwidth*, which defines the capacity of the **IO** bus, the *element bit width*, which defines the data representation used in the computations, and the *tap number*, which defines the number of coefficients used in the filter. A fourth parameter allows to define whether coefficients are hard-coded — *i.e.* stored in **ROMs** — or provided in the communication protocol. However, no exploration process should consider both kinds of implementations as they are not comparable and depend on both applicative needs and target constraints.

Here again — as for **FFT** implementations comparison through multiple sizes — it is difficult to compare **FIR Filter** implementations if the *tap numbers* differ. However, one could define application specific metrics to allow such comparison, for example by defining applicative needs — *e.g.* the **filter type** (*low-pass*, *band-pass*, ...) or **cut-off frequency** — and running empirical simulations to choose which implementation best fit one's need.

Monte Carlo

Both *Pi* and *Black Scholes* kernels are based on a parametric **Monte Carlo** generator, for which the meta architecture is introduced in Figure B.3.

In order to provide random samples following normal distributions for the different cores — which are generated using different seeds — we use a parametric Tausworthe generator to feed pre-computed **Box-Muller ROMs** [Box58]. Using so defined samples, each core performs the given computations, and the partial results are averaged to provide a statistical value for the result — for example, we estimate the value of π by checking if random points are inside a circle of radius one before computing the proportion of points that are within this disk. This can be used to determine the area of the disk, hence we can empirically approximate π.

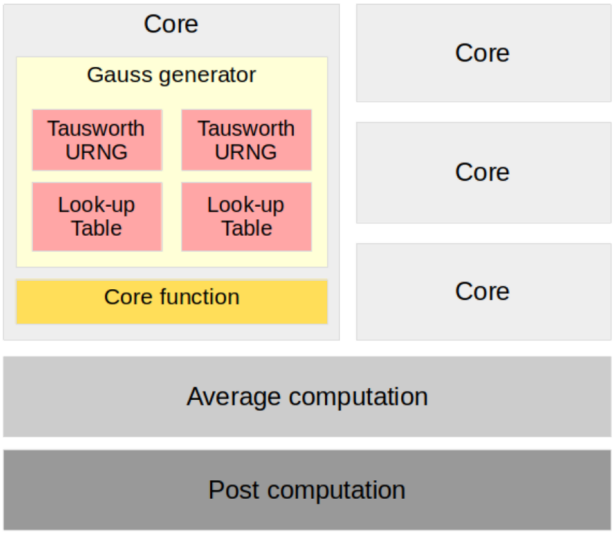


Figure B.3: Monte Carlo kernel meta architecture

We use a **Factory pattern** to build different **Monte Carlo** kernels, fully exploiting the *Object-Oriented Programming* (**OOP**) features of **Chisel** for such generators. This enables to build both *Pi* and *Black Scholes* kernels by defining specific inner functions while allowing code reusability by exploiting inheritance.

For both kernel generators, we thus define four parameters: the *dynamic* and the *precision* to define the Fixed Point representation for the computations, the *iteration number*, which defines the number of experiments to be run before performing both average and post processing to provide a result, and the *core number* which defines the number of parallel cores to run

iterations.

As for the *Black Scholes* generator, we aim at computing the price of an action at time t, given the following equation — μ, σ and T being respectively the **risk free rate**, the **volatility** and the **maturity** of the considered option:

$$S(t) = S(0) \times e^{(\mu - \frac{1}{2}\sigma^2)T + \sigma\sqrt{T}\mathcal{N}(0,1)} \tag{B.6}$$

However, as hardware-based computations of the exponential function are costly, we leverage **Euler-Maruyama** method to sequentially approximate the formula (Eq. B.6):

$$S_{\Delta t} = S_0((1 + (\mu - \frac{1}{2}\sigma^2)\Delta t) + \sigma\sqrt{\Delta t}\mathcal{N}(0,1) \tag{B.7}$$

To exploit this approximation model, the *Black Scholes* kernels are generated using a fifth parameter, namely the *Euler iteration number*, which is used to define how many iterations are taken for each **Euler-Maruyama** based approximation. Moreover, other parameters could be considered for exploration, as this meta implementation uses hard coded values for the *Black Scholes* specific parameters (μ, σ and T), and for the **Monte Carlo** parameters, such as the Tausworthe and the Box-Muller configurations. Such values may also be integrated in the communication protocol, in order to build dynamically programmable accelerators.

For both **Monte Carlo** based kernels, we hence expose heavy design spaces as the **QoS** is considered for exploration — we aim at maximizing the design efficiency (*i.e.* performance *vs* cost ratio) while insuring that the generated designs does not generate significant errors with respect to a given workload model.

Multilayer Perceptron

The *Multilayer Perceptrons* (**MLP**) are simple neural network models that are based on the original Perceptron model as introduced by F. Rosenblatt in 1958 [Ros58]. They consist on a given number of **fully connected** neuron layers — meaning that each neuron of a layer n is connected to every neuron of the layers $n-1$ and $n+1$.

Such model has been used for more than twenty years for multiple uses, notably for image classification over the **MNIST database** (handwritten digits) [LBBH98]. The **MLP** implementations are costly, as all the necessary connections result in a complex interconnection model, which led researchers to develop more compact neural network for *Deep Learning* (**DL**) — especially for image processing and recognition domains — such as *Convolutional*

Neural Network (**CNN**) models. However, some specific domains still require to use fully connected networks — *e.g.* personalization and recommendation systems [NMS+19] — and given their simple structure, we chose to implement a **MLP** generator using **Chisel**.

We chose not to base developed generator on the **Role** model as introduced in Figure B.1, in order to allow a multi-channel interface model, notably to be able to configure the accelerator through a distinct **IO** bus. In order to take the best of **FPGA** specificities, we chose to target a **fully unrolled** implementation, meaning that each neuron in the network will be an independent entity with its own memory and computation units — in contrast to most hardware implementations which are based on scheduling the different neuron tasks on a topology-independent amount of hardware neurons — as existing works proved that unrolling networks on **FPGAs** can lead to efficient implementations [PBBP+17].

Most of the design effort was done at **neuron level**, in order to provide efficient basic units for computing neuron outputs. Let $N(i,j)$ be the j^{th} neuron of the i^{th} layer of a given network. $N(i,j)$ got n different inputs $x_0, ..., x_{n-1}$ resulting from the preceding layer, n different weights $w_0, ..., w_{n-1}$ that result from a prior training of the network and are used for configuration purposes, and a bias b which is also provided by the training phase. An **activation function** f is also applied to each neuron output, and is defined at network level — thus it does not need to be configured at neuron level as it is defined at elaboration time and can be hard-coded.

The output $o(i,j)$ is then computed using the following formula:

$$o(i,j) = f(\sum_{k=0}^{n-1} x_k w_k + b) \tag{B.8}$$

A simplified version of the chosen neuron architecture is introduced in Figure B.4. Each neuron is composed of an embedded memory — which should leverage **BRAMs** for Xilinx boards — to store network weights, and a computation unit to perform *Multiply and Accumulate* (**MAC**) operations. As **MAC** operations can easily be parallelized in a way similar to the dot product, we hence expose a *parallelism* parameter at neuron level to easily define the *Functional Unit* (**FU**) replication factor. In order to properly schedule the output propagation, a *scan-chain* system is inserted after the activation function, to manage the synchronization with the next layer.

Each neuron also includes a parallel data bus for configuration, which uses a simple *Finite State Machine* (**FSM**) to write weights in a **BRAM** — weights are written in a vector-like structure, enabling to access to k weights $w_0, ..., w_{k-1}$ each cycle, in order to feed k **MAC** units in parallel.

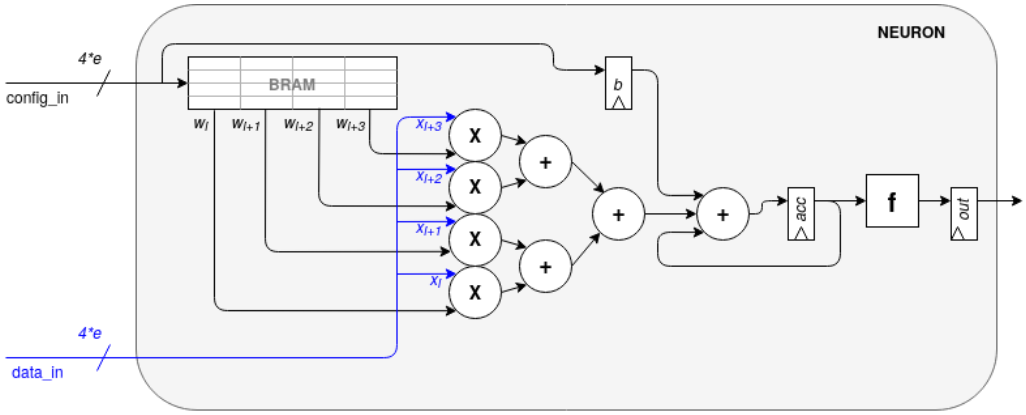


Figure B.4: Simplified schematic for a neuron implementation

Using this basic neuron generator, we thus define the whole network as a composition of layers, each layer being defined as a composition of neurons, with small **FSMs** for the control flow — including two different **IO** buses, respectively for the data and the weights.

We chose to use MNIST as a use case for so defined **MLP**, as it is both a simple example and a standard reference in image processing.

Beside the *parallelism* parameter, which allows to explore a trade-off between the neuron throughput and its resource usage, we defined three parameters: *#neuron(*1^{st} *layer)* and *#neuron(*2^{nd}*)* respectively to define the number of neurons in 1^{st} and 2^{nd} layers, while the *element bit width* once again defines how data are represented in the design. We here only explore three layered networks, where the first layer is composed of as many neurons as there are pixesl in an input picture, and the last layer is composed of as many neurons as the number of possible classifications (here, 10 classes for 10 different digits). Whereas this means that both input and output layer widths are defined at application level, it also means that this generator can be adapted to any network, and that any number of layers can be put between input and output layers, resulting in a possibly very wide design space to explore. Moreover, multiple **activation functions** could be explored using **scala** functional programming features, enabling to compare either different activation functions or multiple implementations of a same activation function.

In order to really take advantage of the **FPGAs** specificities, multiple improvements are considered — including considering hard-coding the network configuration parameters to reduce memory footprint, various arithmetic possibilities that can be leveraged by changing the data type, and an automatically searching for the best topology.

Kernel name	Description	Domain	Parameters	Space example	Impact on QoS
GEMM	Generic Matrix Multiply	Image processing	I/O bandwidth Element bit width Matrices dimensions	@pow2(5, 10) @pow2(3, 6) @pow2(4, 10)	no
FFT	Fast Fourier Transform	Signal processing	Parallelism Element bit width FFT size	@pow2(1, 10) @pow2(3, 6) @pow2(7, 11)	no
MLP	Multilayer Perceptron applied to MNIST recognition	Image processing	#neuron(1^{st} layer) #neuron (2^{nd} layer) Parallelism Element width	@enum(784) @pow2(6, 10) @pow(0, 4) @pow2(0, 4)	no
Dot Product	Dot product parallel implementation	Miscellaneous	Dynamic Precision Vector width Parallelism level	@linear(16, 32) @linear(16, 32) @pow2(2, 16) @pow2(0, 4)	**yes** no
FIR	Finite Impulse Response filter	Signal processing	I/O bandwidth Dynamic Precision Tap number	@pow2(5, 10) @linear(16, 32) @linear(16, 32) @pow2(5, 10)	no **yes**
Pi	Monte Carlo method applied to pi computation	Miscellaneous	Dynamic Precision Iteration number Core number	@linear(16, 32) @linear(16, 32) @pow2(8, 17) @pow2(2, 10)	**yes** no
Black Scholes	Monte Carlo method applied to Black Scholes equations	Finance	Dynamic Precision Iteration number Euler iteration Core number	@linear(16, 32) @linear(16, 32) @pow2(8, 17) @pow2(1, 6) @pow2(2, 10)	**yes** no

Table B.1: Benchmark composition

C

Quality of Resource Estimation

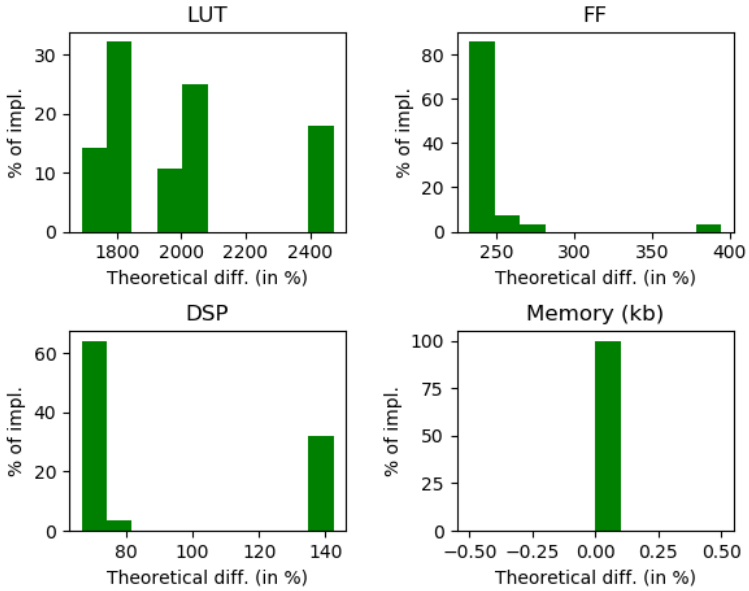
This Appendix introduces a more detailed version of the histograms of relative differences that were presented in Figure 6.3.

For each of the considered kernels — namely *Black Scholes*, *Pi*, *FFT* and *Dot Product* — we exhibit the relative differences of the resource estimations with respect to the synthesis results, for both macro and non-macro estimation methodologies that were introduced in Section 4.2. As for the *GEMM* kernel, the relative differences are shown in Figure 6.6, as they are used for exploration purposes in Section 6.5.1.

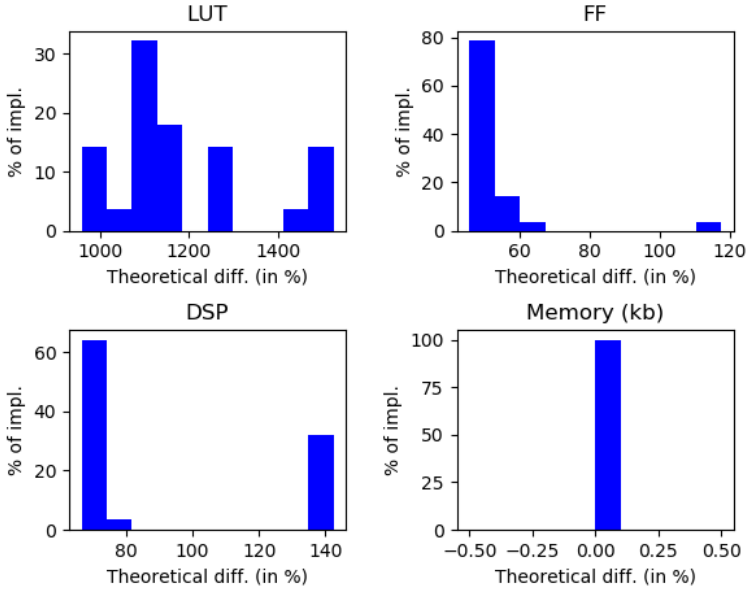
Please refer to Table 6.2 for both the experimental setups and the temporal considerations of these experiments.

Figures

C.1 Quality of resource estimation on Black Scholes 150
C.2 Quality of resource estimation on Pi 151
C.3 Quality of resource estimation on FFT 152
C.4 Quality of resource estimation on dot product 153

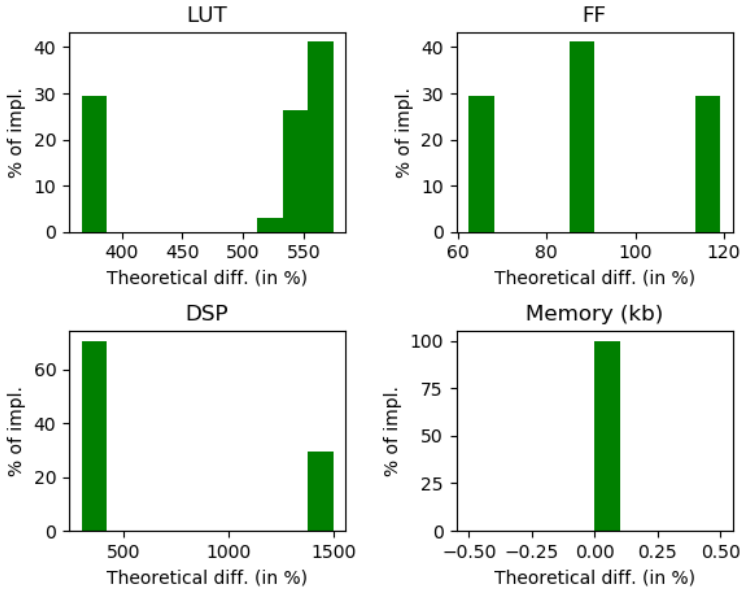


(a) Without macro block replacement

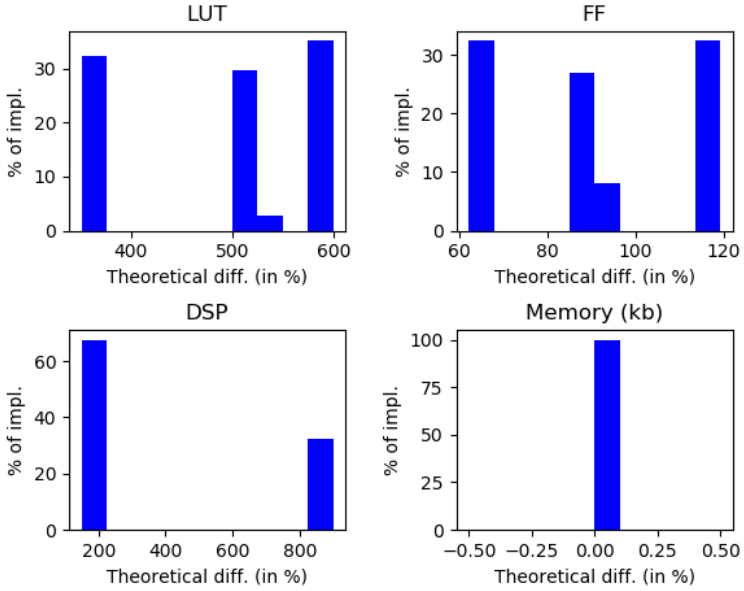


(b) With macro block replacement

Figure C.1: Relative differences between the resource estimations and the synthesis results on Black Scholes kernels

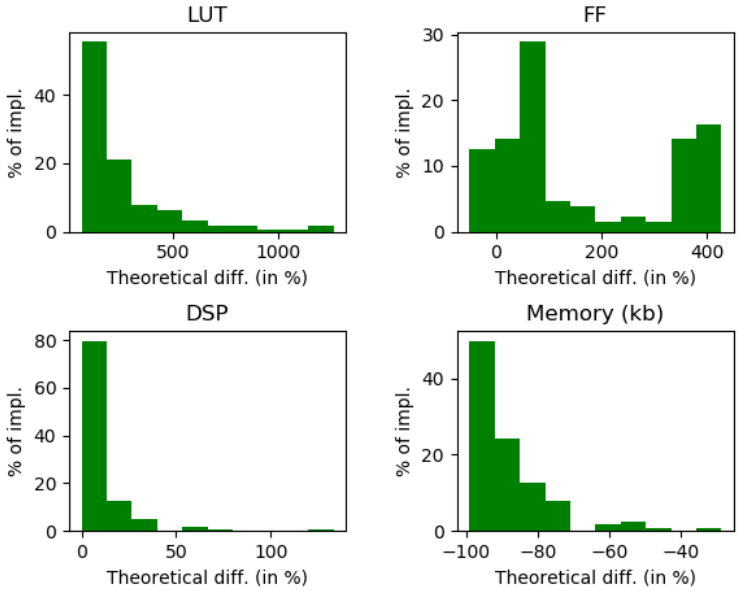


(a) Without macro block replacement

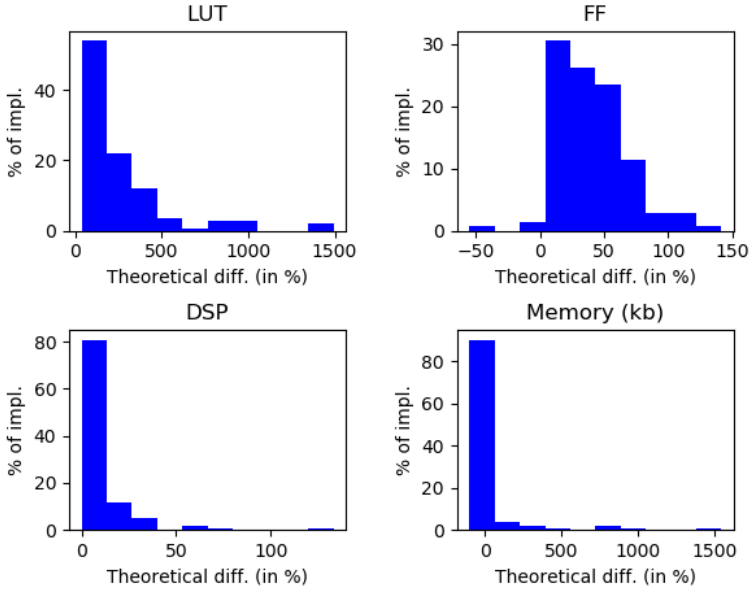


(b) With macro block replacement

Figure C.2: Relative differences between the resource estimations and the synthesis results on Pi kernels

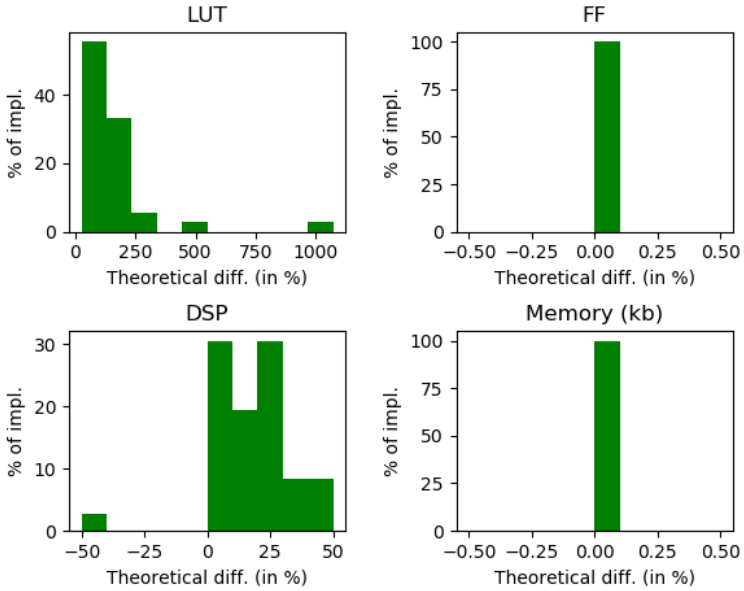


(a) Without macro block replacement

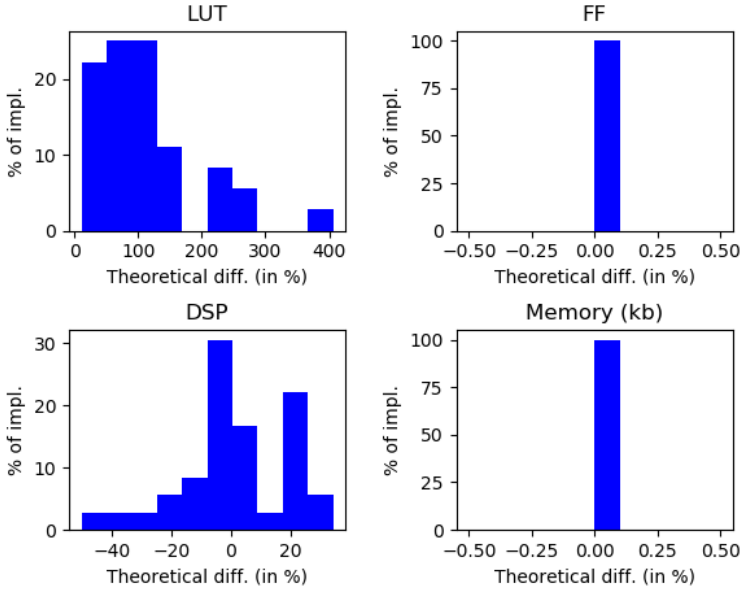


(b) With macro block replacement

Figure C.3: Relative differences between the resource estimations and the synthesis results on Fast-Fourier Transform kernels



(a) Without macro block replacement



(b) With macro block replacement

Figure C.4: Relative differences between the resource estimations and the synthesis results on dot product kernels

D

Comparing the Pruning Strategies

This Appendix provides further data on the empirical *Quality of Service* (**QoS**) estimations that were introduced in Section 6.1.1. It shows the impact of the different pruning strategies for partitioning a given design space, with a particular focus on the quick pruning strategy that was defined in Algorithm 5.2. For an easier representation of the results, we fix the *vector width* parameter to 8, as was done in Section 6.4.3.

In a similar way to what was done in Table 6.5, we consider 4 different strategies for these experiments:

- **Strategy 1:** exhaustive pruning
- **Strategy 2:** exhaustive pruning, with space reduction
- **Strategy 3:** quick pruning, with space reduction (using `SeqSpace`)
- **Strategy 4:** quick pruning, with space reduction (using `MatrixSpace`)

Figures

D.1 Quality of service evolution ($\mathcal{N}(0,1)$, 10 simulations) . . . 158
D.2 Quality of service evolution ($\mathcal{N}(0,1)$, 50 simulations) . . . 158
D.3 Quality of service evolution ($\mathcal{N}(0,1)$, 100 simulations) . . 159
D.4 Quality of service evolution ($\mathcal{N}(0,1)$, 1000 simulations) . . 159
D.5 Quality of service evolution ($\mathcal{N}(32,10)$, 10 simulations) . . 160
D.6 Quality of service evolution ($\mathcal{N}(32,10)$, 50 simulations) . . 160
D.7 Quality of service evolution ($\mathcal{N}(32,10)$, 100 simulations) . 161
D.8 Quality of service evolution ($\mathcal{N}(32,10)$, 1000 simulations) 161

Table D.1 introduces 8 different experiments for comparing those strategies — see Table 6.5 for a detailed explanation of these experiments — along with heat map representations of the **QoS** evolution. For those experiments, we set the acceptable error threshold to 1%, similarly to what was done in Figure 6.7.

As can be observed, the speed of the exhaustive strategies does not seem to be correlated with the initial distribution or with the number of simulations. In fact, it seems that each simulation runtime is negligible with respect to the static cost of launching the simulator, resulting in a null gain when reducing the number of simulations. However, as **dot products** are quite simple kernels, it is possible that for more complex ones, the simulation runtime would scale, meaning that reducing the number of simulations could bring better performance.

On the other hand, we remark that changing the distribution (*i.e.* the simulation workload) can have a direct impact on the number of implementations that are explored to build the **pruning frontier**, thus impacting the duration of each strategy — as the pruned space width reduces, it is easier to build that frontier and thus the pruning process becomes faster. Moreover, we can use those results to confirm the impact that the space structures have on the exploration runtime. We can remark that using a linear `SeqSpace` can be more time-consuming than an exhaustive exploration of the space, while using a more complex `MatrixSpace` implementation can result in an acceleration factor of $\times 5.4$, depending on the size of the pruned search space.

Finally, we can state that reducing the number of dimensions before the exploration is indeed an easy way to reduce the search time, under the assumtion that the user is aware that a particular dimension has no impact on a given metric for the considered exploration step.

As a conclusion, we can state that for **QoS**-based pruning, it seems to be better to use the **quick pruning strategy** with a `MatrixSpace` **structure**, after performing **space reduction**.[1] One should also adapt the number of simulations in order to cope with the **law of large numbers** and provide meaningful **QoS** results.

[1] Here, only the *parallelism* dimension is being ignored at simulation time as it does not impact the implementations **QoS**. Moreover, as the *vector width* is fixed to 8, the exploration is thus in fact done in a **two dimensional** space.

#	Distribution		Simulation	Strategy 1		Strategy 2		Strategy 3		Strategy 4	
	Mean	Std dev.	number	#impl.	Time	#impl.	Time	#impl.	Time	#impl.	Time
D.1	0	1	10	3844	12m26s	961	3m08s	275	5m23s	275	54s
D.2	0	1	50	3844	12m24s	961	3m08s	270	5m17s	270	52s
D.3	0	1	100	3844	12m22s	961	3m09s	260	5m06s	260	49s
D.4	0	1	1000	3844	12m22s	961	3m07s	255	5m03s	255	49s
D.5	32	10	10	3844	12m24s	961	3m09s	181	3m23s	181	35s
D.6	32	10	50	3844	12m25s	961	3m06s	181	3m23s	181	35s
D.7	32	10	100	3844	12m32s	961	3m08s	181	3m24ss	181	35s
D.8	32	10	1000	3844	12m20s	961	3m07s	181	3m25s	181	35s

Table D.1: Comparison of the pruning strategies on dot product implementations

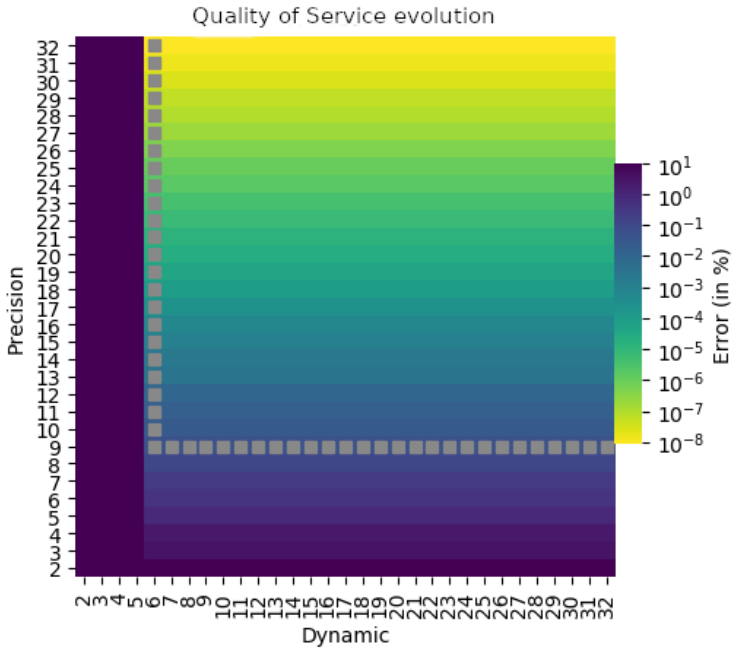


Figure D.1: Quality of service evolution ($\mathcal{N}(0,1)$, 10 simulations)

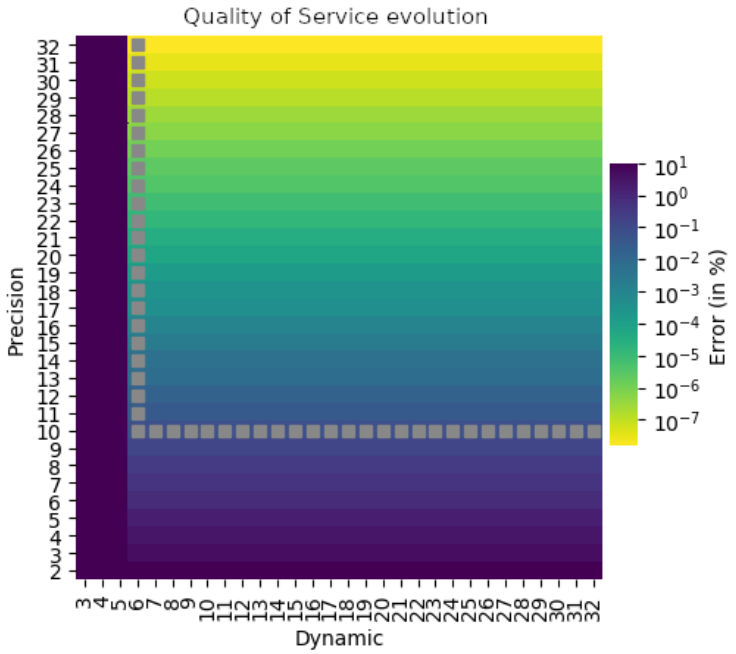


Figure D.2: Quality of service evolution ($\mathcal{N}(0,1)$, 50 simulations)

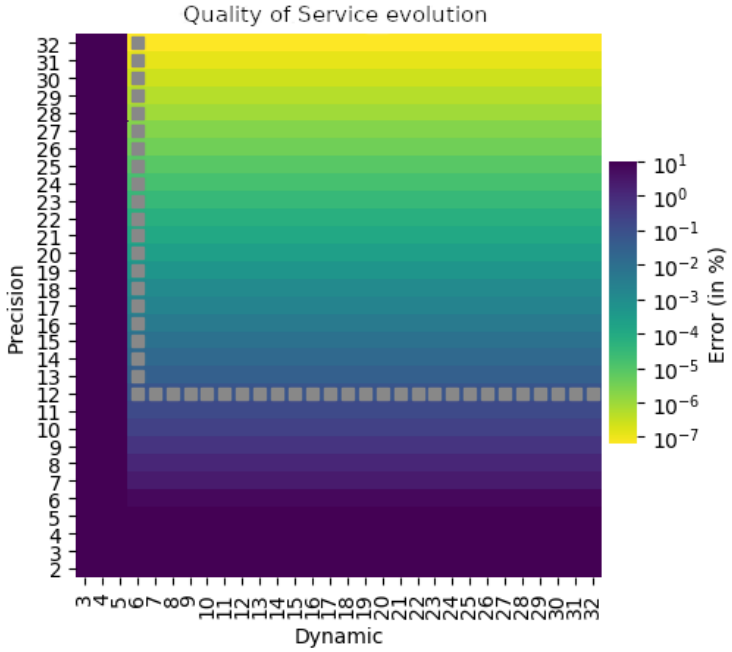


Figure D.3: Quality of service evolution ($\mathcal{N}(0,1)$, 100 simulations)

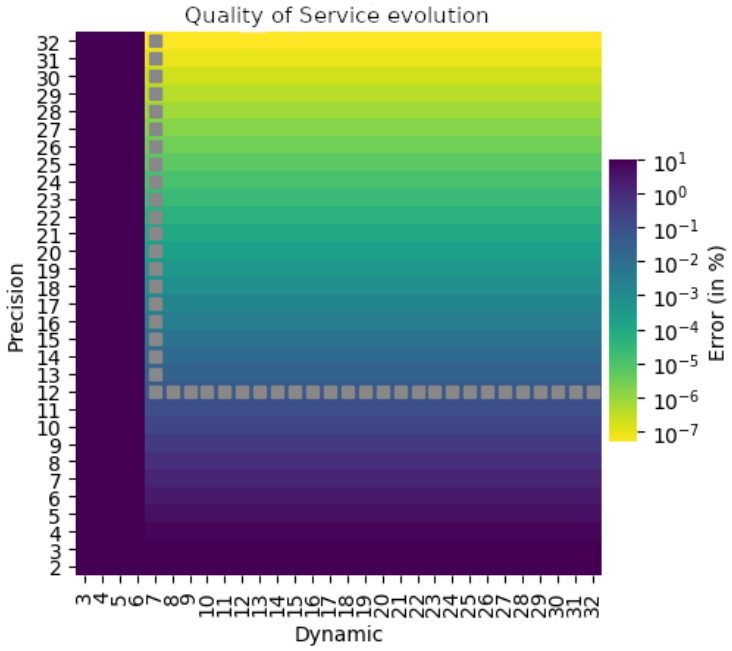


Figure D.4: Quality of service evolution ($\mathcal{N}(0,1)$, 1000 simulations)

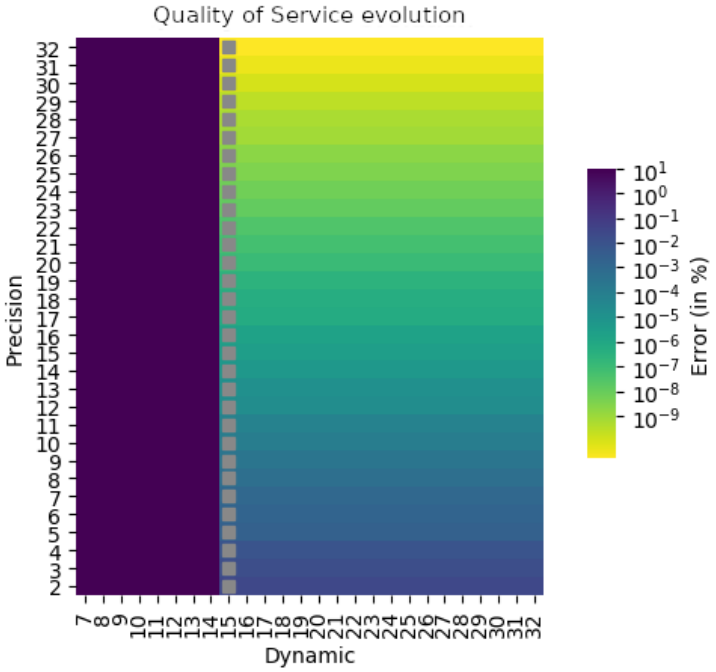


Figure D.5: Quality of service evolution ($\mathcal{N}(32, 10)$, 10 simulations)

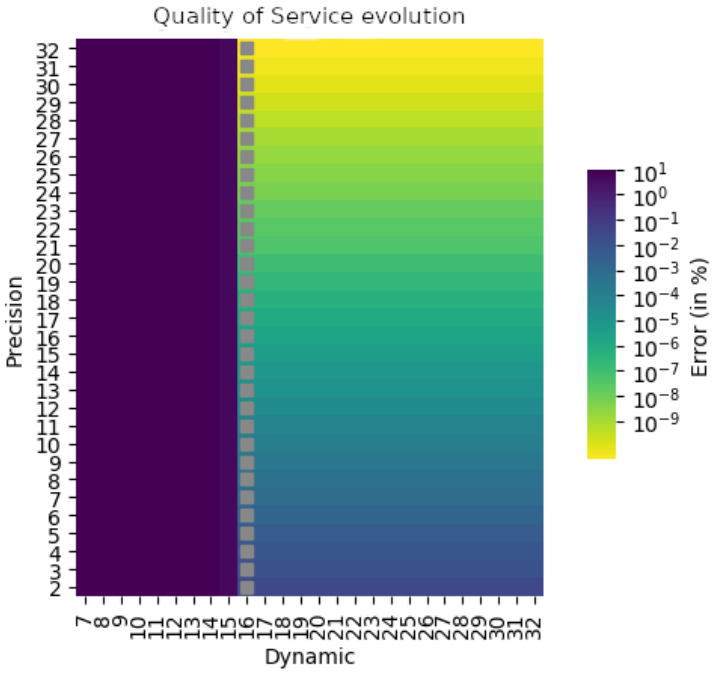


Figure D.6: Quality of service evolution ($\mathcal{N}(32, 10)$, 50 simulations)

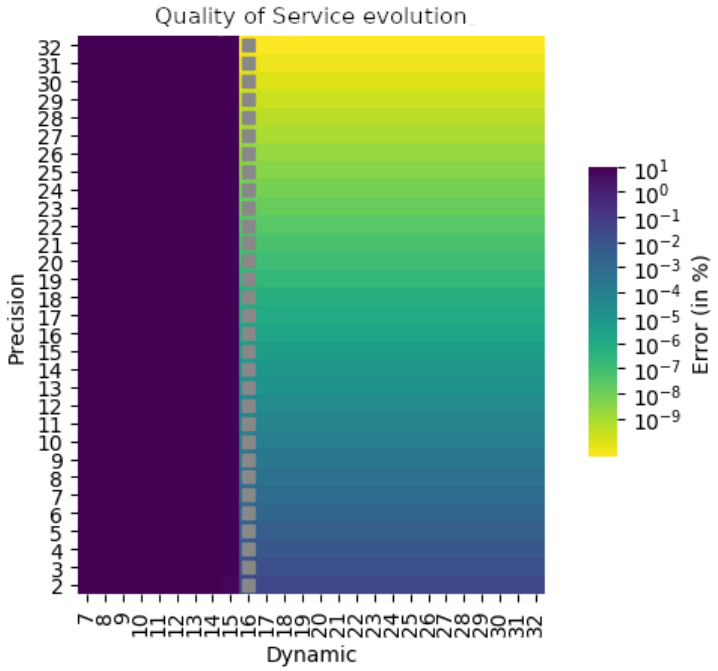


Figure D.7: Quality of service evolution ($\mathcal{N}(32, 10)$, 100 simulations)

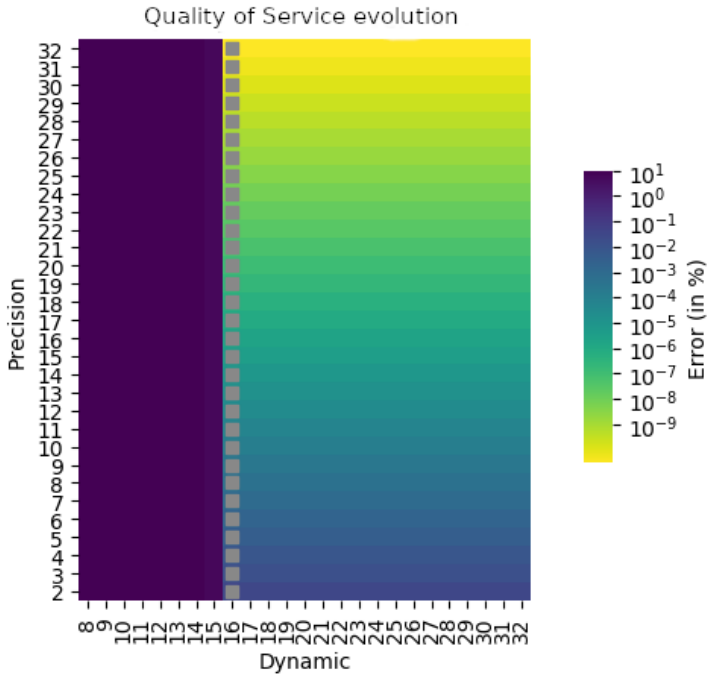


Figure D.8: Quality of service evolution ($\mathcal{N}(32, 10)$, 1000 simulations)

Publications

International publications

ARC2020 Toledo, Spain (short paper + poster) [FMR20]
Chisel Usecase: Designing General Matrix Multiply for FPGA

RSP'2021 *Virtual event* [FMR21a]
Integrating Quick Resource Estimators in Hardware Construction Framework for Design Space Exploration

Glossary

API *Application Programming Interface.*
An interface proposed by the developers of programming tools to their users.

AxC *Approximate Computing.*
Application domain relying on a simple assertion: most of the applications are redundant enough to allow approximation in the intermediate results, thus some resources can be saved by changing the data representation and using approximate **FUs**.

ASIC *Application-Specific Integrated Circuit.*
An integrated circuit customized for a specific application.

BOOM *Berkeley Out-of-Order Machine.*
A **Chisel**-based generator of RISC-V out-of-order cores [CPA15].

BRAM *Block Random-Access Memory.*
A memory block embedded in the **FPGA** itself, allowing an access that is quicker than the external memories that may be available.

Chisel *Constructing Hardware in a Scala Embedded Language.*
A **scala**-based **HCL** developed at Berkeley since 2012 [BVR+12].

CLB *Configurable Logic Block.*
Basic blocks for Xilinx **FPGAs**, including both computation resources (**LUTs**) and memory resources (**FFs**).

CNN *Convolutional Neural Network.*
A neural network model where each neuron is connected to a bounded amount of preceding and succeeding neurons, in contrast to fully connected networks.

CPU *Central Processing Unit*

DL *Deep Learning.*
Machine learning methods based on multiple layered networks.

DIF *Decimation In Frequency.*
A standard approach for **FFT** computations, which enables consuming the temporal data in a **FIFO** fashion.

DG *Directed Graph*

DSE *Design Space Exploration.*
A manual or automatic methodology for the exploration of a design space, in order to find the best fit for an algorithm hardware implementation.

DSL *Domain Specific Language.*
A programming language targeting a specific domain, and thus embedding very specific features which are keys for the usual computations in this field. **DSLs** can thus be accelerated using some specific components, and non experts can take advantage of such acceleration.

DSP *Digital Signal Processor.*
A computing unit dedicated to digital processing applications.

FIFO *First-In First-Out.*
A data structure in which data are consumed in their order of arrival.

FIR Filter *Finite Impulse Response Filter.*
A class of representative algorithms of signal processing. It is a digital filter using a finite number of coefficients.

FIRRTL *Flexible Intermediate Representation for* **RTL**.
The *Intermediate Representation* (**IR**) used by **Chisel**.

FF *Flip Flop.*
Basic units of memorization of one bit— often assimilated to the registers.

FFT *Fast Fourier Transform.*
A representative algorithm in the signal processing field. It converts input data from the time domain to the frequency domain.

FPGA *Field-Programmable Gate Array.*
A reconfigurable circuit that is able to behave as any sequential or combinatorial circuit.

FPU *Floating-Point Unit.*
Computation units dedicated to floating-point computations, based on the IEEE-754 standard.

FSM *Finite State Machine*

FU *Functional Unit.*
Computation units used as the basis of a given architecture.

GA *Genetic Algorithm.*
A class of evolutionary algorithms that can be used for optimization and search problem resolution.

GEMM *General Matrix Multiply.*
A representative algorithm in the field of linear algebra. It is a generalization of the basic matrix multiplication algorithm.

GPU *Graphical Processing Unit*

HCF *Hardware Construction Framework.*
A framework used to compile an entry **HCL**-based code to a **RTL** description that can be fed to any low level toolchain.
HCFs are similar to standard software compilers in their design, using a frontend/transforms/backend separation.

HCL *Hardware Construction Language.*

A hardware language enabling the definition of hardware generators instead of hardware designs, to ease the re-utilization of the code, thus speeding-up the hardware development processes.

HDL *Hardware Description Language.*

Standard **RTL** languages, such as *verilog*, **system-***verilog* or *VHDL*.

HLS *High Level Synthesis.*

A design methodology based on the compilation of algorithmic specification toward a hardware description, to ease and speed-up the hardware development.

IO *Input/Output*

IR *Intermediate Representation.*

An internal representation used by a compiler to abstract concerns from both the entry language and the target machine.

IP *Intellectual Property (core).*

A reusable unit of logic, often subject to intellectual property laws, used as a functional block in **ASICs** and **FPGAs**.

JSON *JavaScript Object Notation.*

A JavaScript-based format for representing textual data.

LUT *Look-Up Table.*

Basic electronic components, able to model any boolean function for a given amount of inputs (usually 4 or 6). Notably used in the design of recent **FPGAs**, due to this generic feature.

MAC *Multiply and Accumulate.*

A basic pattern of operation used in many domains such as signal processing, image processing or machine learning. It relies on computing a joint addition and multiplication.

ML *Machine Learning*

MLP *Multilayer Perceptron.*
A model of fully connected neural networks derived from the original Perceptron model [Ros58].

MOP *Multi-objective Optimization Problem.*
As defined by Barone et *al.* [BTBB21], it is the process of "finding, for some *decision variables*, a set of values satisfying *imposed constraints*, while optimizing a set of *objective functions*" [Osy85].

NoC *Network-on-Chip.*
A design paradigm which aims at integrating the communication system directly on the chip.

OOP *Object-Oriented Programming*

QECE *Quick Exploration using Chisel Estimators.*
An estimation and exploration framework built as a *proof of concept* of the usage of **HCLs** for **DSE** [FMR21c].

QoR *Quality of Results.*
In the context of this thesis, we consider the **QoR** to be the quality of the proposed estimators with respect to some reference values about circuit properties. A distinction must be done between **QoR** and **QoS**, as the second one is an actual property of the circuit while the first one provides insights about the adequacy of the estimation flow with respect to the development environment.

QoS *Quality of Service.*
A property of a circuit that exhibits the errors that were introduced by the implementation with respect to the initial algorithm. It is particularly relevant in the context of *Approximate Computing* (**AxC**), where one can build more performant designs at the cost of accuracy.

RAM *Random-Access Memory*

R2MDC *Radix-2 Multi-Path Delay Commutator.*

A standard technique for **FFT** optimization. The proposed implementation is based on a presentation from Gerez [Ger12] (itself based on the literature [RG75]).

RMSE *Root-Mean-Square Error.*

A standard metric defined by computing the relative differences between theoretical and experimental values.

RTL *Register-Transfer Level.*

The abstraction level used for hardware development, abstracting low level consideration to only consider signal interactions.

ROM *Read-Only Memory*

SoC *Systems on a Chip.*

A single integrated circuit (**ASIC**) embedding all the needed components - such as memories, **IO** ports or microprocessors - for a given application.

TPU *Tensor Processing Unit.*

An Artificial Intelligence dedicated **ASIC**.

VLSI *Very Large Scale Integration.*

Design processes used to integrate millions of transistors in a chip.

Bibliography

[AAB+16] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. The Rocket Chip Generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016. (Cited on pages 11, 23, and 129.)

[AAPLP21] Ayaz Akram, Venkatesh Akella, Sean Peisert, and Jason Lowe-Power. Enabling Design Space Exploration for RISC-V Secure Compute Environments. *Lawrence Berkeley National Laboratory*, 2021. (Cited on page 30.)

[ACP04] Giuseppe Ascia, Vincenzo Catania, and Maurizio Palesi. A GA-Based Design Space Exploration Framework for Parameterized System-On-A-Chip Platforms. *IEEE Transactions on Evolutionary Computation*, 2004. (Cited on page 19.)

[AGMP21] Muhammad Awais, Hassan Ghasemzadeh Mohammadi, and Marco Platzner. LDAX: A Learning-based Fast Design Space Exploration Framework for Approximate Circuit Synthesis. In *Proceedings of the 2021 on Great Lakes Symposium on VLSI*, Virtual Event USA, June 2021. ACM. (Cited on pages 30, 33, and 108.)

[ALS15] Yan Lin Aung, Siew-Kei Lam, and Thambipillai Srikanthan. Rapid Estimation of DSPs Utilization for Efficient High-Level Synthesis. In *International Conference on Digital Signal Processing (DSP)*, Singapore, Singapore, July 2015. IEEE. (Cited on page 28.)

[AW17] Muhammad Shoaib Bin Altaf and David A Wood. LogCA: A High-Level Performance Model for Hardware Accelerators. *ACM SIGARCH Computer Architecture News*, 2017. (Cited on page 113.)

[Ber21a] Berkeley. Chisel3 Cheat Sheet. https://github.com/freechipsproject/chisel-cheatsheet/releases/latest/download/chisel_cheatsheet.pdf, 2021. [ONLINE] Last accessed on 10th november, 2021. (Cited on page 130.)

[Ber21b] Berkeley Architecture Research. Chipyard project. https://chipyard.readthedocs.io/en/latest/index.html, 2021. [ONLINE] Last accessed on 10th november, 2021. (Cited on page 129.)

[BHM+21] Jean Bruant, Pierre-Henri Horrein, Olivier Muller, Tristan Groleat, and Frederic Petrot. Towards Agile Hardware Designs with Chisel: a Network Use-case. *IEEE Design & Test*, 2021. (Cited on pages 5 and 114.)

[BIM16] Mario Barbareschi, Federico Iannucci, and Antonino Mazzeo. Automatic Design Space Exploration of Approximate Algorithms for Big Data Applications. In *2016 30th International Conference on Advanced Information Networking and Applications Workshops (WAINA)*, Crans-Montana, Switzerland, March 2016. IEEE. (Cited on page 32.)

[BKK+10] Christiaan Baaij, Matthijs Kooijman, Jan Kuper, Arjan Boeijink, and Marco Gerards. ClaSH: Structural Descriptions of Synchronous Hardware Using Haskell. In *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, Lille, France, September 2010. IEEE. (Cited on pages 11, 23, and 129.)

[BMJ02] Per Bjuréus, Mikael Millberg, and Axel Jantsch. FPGA Resource and Timing Estimation from Matlab Execution Traces. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign. CODES 2002 (IEEE Cat. No. 02TH8627)*. IEEE, 2002. (Cited on page 29.)

[Box58] George EP Box. A note on the generation of random normal deviates. *Ann. Math. Statist.*, 1958. (Cited on page 144.)

[BPSBB21] Andre Bannwart Perina, Arthur Silitonga, Jurgen Becker, and Vanderlei Bonato. Fast Resource and Timing Aware Design Optimisation for High-Level Synthesis. *IEEE Transactions on Computers*, 2021. (Cited on pages 29 and 33.)

[BSZ+21] Chen Bai, Qi Sun, Jianwang Zhai, Yuzhe Ma, Bei Yu, and Martin DF Wong. BOOM-Explorer: RISC-V BOOM Microarchitecture Design Space Exploration Framework. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2021. (Cited on pages 23, 33, 34, and 35.)

[BTBB21] Salvatore Barone, Marcello Traiola, Mario Barbareschi, and Alberto Bosio. Multi-Objective Application-Driven Approximate Design Method. *IEEE Access*, 2021. (Cited on pages 19, 24, 26, 32, 36, 113, and 167.)

[BVR+12] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R Avižienis, J. Wawrzynek, and K. Asanović. Chisel: Constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*, June 2012. (Cited on pages 11, 23, 129, and 163.)

[CCA+11] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H Anderson, Stephen Brown, and Tomasz Czajkowski. LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, 2011. (Cited on pages 21 and 42.)

[CCP+16] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. A Cloud-scale Acceleration Architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016. (Cited on page 139.)

[CGMVSH20] Jorge Castro-Godínez, Julián Mateus-Vargas, Muhammad Shafique, and Jörg Henkel. AxHLS: design space exploration and high-level synthesis of approximate accelerators using approximate functional units and analytical models. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, Virtual Event USA, 2020. IEEE. (Cited on page 28.)

[CPA15] Christopher Celio, David A Patterson, and Krste Asanović. The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-167*, 2015. (Cited on pages 11, 129, and 163.)

[CTL17] Henry Cook, Wesley Terpstra, and Yunsup Lee. Diplomatic Design Patterns: A TileLink Case Study. In *1st Workshop on Computer Architecture Research with RISC-V*, 2017. (Cited on pages 23 and 129.)

[CV10] Horia Calborean and Lucian Vmtan. An Automatic Design Space Exploration Framework for Multicore Architecture Optimizations. In *9th RoEduNet IEEE International Conference*. IEEE, 2010. (Cited on page 19.)

[DLGCJ20] Yann Delomier, Bertrand Le Gal, Jeremie Crenne, and Christophe Jego. Model-based Design of Hardware SC Polar Decoders for FPGAs. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, May 2020. (Cited on page 114.)

[DS16] Dong Liu and Benjamin Carrion Schafer. Efficient and reliable High-Level Synthesis Design Space Explorer for FPGAs. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, Lausanne, Switzerland, August 2016. IEEE. (Cited on pages 34 and 35.)

[DSC08] Lanping Deng, Kanwaldeep Sobti, and Chaitali Chakrabarti. Accurate Models for Estimating Area and Power of FPGA Implementations. In *2008 IEEE International Conference on Acoustics, Speech and Signal Processing*, Las Vegas, NV, USA, March 2008. IEEE. (Cited on pages 27, 30, and 42.)

[DTOBS17] Lorenzo Di Tucci, Kenneth O'Brien, Michaela Blott, and Marco D. Santambrogio. Architectural optimizations for high performance and energy efficient Smith-Waterman implementation on FPGAs using OpenCL. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, Lausanne, Switzerland, March 2017. IEEE. (Cited on page 5.)

[FCB14] Virginie Fresse, Catherine Combes, and Hatem Belhasseb. Mathematical models applied to on-chip network on FPGA for resource estimation. In *2014 IEEE 17th International Conference on Computational Science and Engineering*. IEEE, 2014. (Cited on page 28.)

[FKA+20] Lorenzo Ferretti, Jihye Kwon, Giovanni Ansaloni, Giuseppe Di Guglielmo, Luca P. Carloni, and Laura Pozzi. Leveraging Prior Knowledge for Effective Design-Space Exploration in High-Level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, November 2020. (Cited on pages 15, 34, and 77.)

[FMR20] Bruno Ferres, Olivier Muller, and Frédéric Rousseau. Chisel Usecase: Designing General Matrix Multiply for FPGA. In *International Symposium on Applied Reconfigurable Computing*. Springer, 2020. (Cited on pages 141 and 162.)

[FMR21a] Bruno Ferres, Olivier Muller, and Frédéric Rousseau. Integrating Quick Resource Estimators in Hardware Construction Framework for Design Space Exploration. In *International Workshop on Rapid System Prototyping*, 2021. (Cited on pages 22, 141, and 162.)

[FMR21b] Bruno Ferres, Olivier Muller, and Frédéric Rousseau. A Chisel based Exploration Benchmark. https://gricad-gitlab.univ-grenoble-alpes.fr/tima/sls/projects/qece-benchmark, 2021. [ONLINE] Last accessed on 3rd novembre, 2021. (Cited on pages 85, 114, and 139.)

[FMR21c] Bruno Ferres, Olivier Muller, and Frédéric Rousseau. QECE: Quick Exploration using Chisel Estimations. https://gricad-gitlab.univ-grenoble-alpes.fr/tima/sls/projects/qece, 2021. [ONLINE] Last accessed on 3rd novembre, 2021. (Cited on pages 83, 114, and 167.)

[Ger12] Sabih H. Gerez. Pipeline Implementations of the Fast Fourier Transform (FFT). http://sabihgerez.com/ut/sendfile/sendfile.php/idsp-fft.pdf?sendfile=idsp-fft.pdf, 2012. [ONLINE] Last accessed on 21st september, 2021. (Cited on pages 142 and 168.)

[GMX+21] Hao Geng, Yuzhe Ma, Qi Xu, Jin Miao, Subhendu Roy, and Bei Yu. High-Speed Adder Design Space Exploration via Graph Neural Processes. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021. (Cited on pages 19 and 34.)

[GSN21] Lukas Gressl, Christian Steger, and Ulrich Neffe. Design Space Exploration for Secure IoT Devices and Cyber-Physical Systems. *ACM Transactions on Embedded Computing Systems*, June 2021. (Cited on page 30.)

[GWC16] Xitong Gao, John Wickerson, and George A. Constantinides. Automatically Optimizing the Latency, Area, and Accuracy of C Programs for High-Level Synthesis. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Monterey California USA, February 2016. ACM. (Cited on pages 24 and 30.)

[HMS05] Nicolas Herve, Daniel Menard, and Olivier Sentieys. Data wordlength optimization for FPGA synthesis. In *IEEE Workshop on Signal Processing Systems Design and Implementation, 2005.* IEEE, 2005. (Cited on pages 24 and 29.)

[IKL+17] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, and J. Bachrach. Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2017. (Cited on pages 11 and 43.)

[Int21] Intel. Intel Early Power Estimator. https://www.intel.com/content/www/us/en/support/programmable/support-resources/power/pow-powerplay.html, 2021. [ONLINE] Last accessed on 27th october, 2021. (Cited on page 30.)

[JS15] Keerthan Jaic and Melissa C Smith. Enhancing Hardware Design Flows with MyHDL. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2015. (Cited on page 23.)

[KB16] Nachiket Kapre and Samuel Bayliss. Survey of domain-specific languages for FPGA computing. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, Lausanne, Switzerland, August 2016. IEEE. (Cited on page 21.)

[KBSD19] Martin Kristien, Bruno Bodin, Michel Steuwer, and Christophe Dubach. High-level synthesis of functional patterns with Lift. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming - ARRAY 2019*, Phoenix, AZ, USA, 2019. ACM Press. (Cited on pages 21 and 24.)

[KC20] Jihye Kwon and Luca P. Carloni. Transfer Learning for Design-Space Exploration with High-Level Synthesis. In *Proceedings of the 2020 ACM/IEEE Workshop on Machine Learning for CAD*, Virtual Event Iceland, November 2020. ACM. (Cited on pages 29, 90, and 113.)

[KFP+18] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, et al. Spatial: A Language and Compiler for Application Accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2018. (Cited on pages 21 and 25.)

[LBBH98] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 1998. (Cited on page 145.)

[LC13] Hung-Yi Liu and Luca P. Carloni. On learning-based methods for design-space exploration with high-level synthesis. In *Proceedings of the 50th Annual Design Automation Conference on - DAC '13*, Austin, Texas, 2013. ACM Press. (Cited on page 34.)

[LC16] Charles Lo and Paul Chow. Model-based optimization of High Level Synthesis directives. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, Lausanne, Switzerland, August 2016. IEEE. (Cited on pages 32 and 77.)

[LC18] Charles Lo and Paul Chow. Multi-fidelity Optimization for High-Level Synthesis Directives. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, Dublin, Ireland, August 2018. IEEE. (Cited on pages 29, 32, 41, 54, and 77.)

[LIB16] Patrick S. Li, Adam M. Izraelevitz, and Jonathan Bachrach. Specification for the firrtl language. Technical Report UCB/EECS-2016-9, EECS Department, University of California, Berkeley, Feb 2016. (Cited on page 43.)

[LLS19] Shuangnan Liu, Francis CM Lau, and Benjamin Carrion Schafer. Accelerating fpga prototyping through predictive model-based hls design space exploration. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2019. (Cited on page 19.)

[LT18] Derek Lockhart and Stephen Twigg. Experiences Building Edge TPU with Chisel. https://www.youtube.com/watch?v=x85342Cny8c, 2018. [ONLINE] Last accessed on 12nd october, 2021. (Cited on pages 11 and 129.)

[LZB14] Derek Lockhart, Gary Zibrat, and Christopher Batten. PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, Cambridge, United Kingdom, December 2014. IEEE. (Cited on pages 11, 23, and 129.)

[LZPC15] Peng Li, Peng Zhang, Louis-Noel Pouchet, and Jason Cong. Resource-Aware Throughput Optimization for High-Level Synthesis. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Monterey California USA, February 2015. ACM. (Cited on page 30.)

[LZSZ20] Zhe Lin, Jieru Zhao, Sharad Sinha, and Wei Zhang. HLPow: A Learning-Based Power Modeling Framework for High-Level Synthesis. In *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, Beijing, China, January 2020. IEEE. (Cited on page 30.)

[MAGK16] Pingfan Meng, Alric Althoff, Quentin Gautier, and Ryan Kastner. Adaptive Threshold Non-Pareto Elimination: Rethinking Machine Learning for System Level Design Space Exploration on FPGAs. In *Proceedings of the 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. Research Publishing Services, 2016. (Cited on pages 34 and 35.)

[MGAG16] Mohammad Motamedi, Philipp Gysel, Venkatesh Akella, and Soheil Ghiasi. Design space exploration of FPGA-based Deep Convolutional Neural Networks. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, Macao, Macao, January 2016. IEEE. (Cited on page 34.)

[MKC+20] Manu Manuel, Arne Kreddig, Simon Conrady, Nguyen Anh Vu Doan, and Walter Stechele. Model-Based Design Space Exploration for Approximate Image Processing on FPGA. In *2020 IEEE Nordic Circuits and Systems Conference (NorCAS)*, Oslo, Norway, October 2020. IEEE. (Cited on pages 15, 24, 30, and 32.)

[MMLT17] Asma Mkhinini, Paolo Maistri, Regis Leveugle, and Rached Tourki. HLS design of a hardware accelerator for Homomorphic Encryption. In *2017 IEEE 20th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, 2017. (Cited on page 114.)

[MOG+13] Roel Meeuws, S. Arash Ostadzadeh, Carlo Galuzzi, Vlad Mihai Sima, Razvan Nane, and Koen Bertels. Quipu: A Statistical Model for Predicting Hardware Resources. *ACM Transactions on Reconfigurable Technology and Systems*, May 2013. (Cited on page 28.)

[Moo65] Gordon E Moore. Cramming more components onto integrated circuits. 1965. (Cited on page 6.)

[MS14] Anushree Mahapatra and Benjamin Carrion Schafer. Machine-Learning Based Simulated Annealer Method for High Level Synthesis Design Space Exploration. In *Proceedings of the 2014 Electronic System Level Synthesis Conference (ESLsyn)*, San Francisco, CA, USA, May 2014. IEEE. (Cited on page 34.)

[NHCB02] Anshuman Nayak, Malay Haldar, Alok Choudhary, and Prithviraj Banerjee. Accurate area and delay estimators for FPGAs. In *Proceedings 2002 Design, Automation and Test in Europe Conference and Exhibition*. IEEE, 2002. (Cited on page 27.)

[Nik08] Nikhil, Rishiyur S. What is Bluespec? *ACM SIGDA Newsletter*, 2008. (Cited on page 21.)

[NKO19] Luigi Nardi, David Koeplinger, and Kunle Olukotun. Practical Design Space Exploration. In *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Rennes, FR, October 2019. IEEE. (Cited on pages 15, 34, 42, and 77.)

[NMS+19] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091*, 2019. (Cited on page 146.)

[NVS+17] Eriko Nurvitadhi, Ganesh Venkatesh, Jaewoong Sim, Debbie Marr, Randy Huang, Jason Ong Gee Hock, Yeong Tat Liew, Krishnan Srivatsan, Duncan Moss, Suchit Subhaschandra, et al. 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 - FPGA '17*, 2017. (Cited on page 5.)

[OLT+18] Kenneth O'Neal, Mitch Liu, Hans Tang, Amin Kalantar, Kennen DeRenard, and Philip Brisk. HLSPredict: cross platform performance prediction for FPGA high-level synthesis. In *Proceedings of the International Conference on Computer-Aided Design*, San Diego California, November 2018. ACM. (Cited on pages 28 and 30.)

[Osy85] Andrzej Osyczka. Multicriteria optimization for engineering design. In *Design optimization*. Elsevier, 1985. (Cited on pages 26 and 167.)

[Pap17] C Papon. SpinalHDL: An Alternative Hardware Description Language. *FOSDEM*, 2017. (Cited on page 23.)

[PB14] Adrien Prost-Boucle. *Génération rapide d'accélérateurs matériels par synthèse d'architecture sous contraintes de ressources*. PhD thesis, Université de Grenoble, 2014. (Cited on pages 7 and 9.)

[PBBP+17] Adrien Prost-Boucle, Alban Bourge, Frédéric Pétrot, Hande Alemdar, Nicholas Caldwell, and Vincent Leroy. Scalable high-performance architecture for convolutional ternary neural networks on FPGA. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2017. (Cited on pages 114 and 146.)

[PBMR14] Adrien Prost-Boucle, Olivier Muller, and Frédéric Rousseau. Fast and standalone Design Space Exploration for High-Level Synthesis under resource constraints. *Journal of Systems Architecture*, January 2014. (Cited on pages 24, 28, and 33.)

[PCS21] Daniele Paletti, Davide Conficconi, and Marco D. Santambrogio. Dovado: An Open-Source Design Space Exploration Framework. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, Portland, OR, USA, June 2021. IEEE. (Cited on pages 23, 29, 32, 37, 77, 90, and 113.)

[PE17] Oron Port and Yoav Etsion. DFiant: A dataflow hardware description language. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, Ghent, Belgium, September 2017. IEEE. (Cited on page 21.)

[PPP20] Chan Park, Sungkyung Park, and Chester Sungchung Park. Roofline-Model-Based Design Space Exploration for Dataflow Techniques of CNN Accelerators. *IEEE Access*, 2020. (Cited on pages 23 and 34.)

[REHS+16] Semeen Rehman, Walaa El-Harouni, Muhammad Shafique, Akash Kumar, and Jörg Henkel. Architectural-space exploration of approximate multipliers. In *Proceedings of the 35th International Conference on Computer-Aided Design*, Austin Texas, November 2016. ACM. (Cited on pages 19, 22, and 30.)

[RG75] Lawrence R Rabiner and Bernard Gold. Theory and application of digital signal processing. *Englewood Cliffs: Prentice-Hall*, 1975. (Cited on pages 142 and 168.)

[Ros58] Frank Rosenblatt. The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain. *Psychological review*, (6), 1958. (Cited on pages 145 and 167.)

[San15] Kentaro Sano. Dsl-based design space exploration for temporal and spatial parallelism of custom stream computing. *arXiv preprint arXiv:1509.00040*, 2015. (Cited on pages 21 and 24.)

[SDR+21] Marco Siracusa, Emanuele Delsozzo, Marco Rabozzi, Lorenzo Di Tucci, Samuel Williams, Donatella Sciuto, and Marco Domenico Santambrogio. A Comprehensive Methodology to Optimize FPGA Designs via the Roofline Model. *IEEE Transactions on Computers*, 2021. (Cited on pages 28, 30, and 33.)

[SHM+04] Changchun Shi, James Hwang, Scott McMillan, Ann Root, and Vinay Singh. A System Level Resource Estimation Tool for FPGAs. In *Field Programmable Logic and Application*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. (Cited on pages 27 and 29.)

[SHM13] Ashkan Shokri, Omid Bozorg Haddad, and Miguel A Mariño. Algorithm for increasing the speed of evolutionary optimization and its accuracy in multi-objective problems. *Water resources management*, 2013. (Cited on page 32.)

[SI15] Colin Schmidt and Adam Izraelevitz. A Fast Parameterized SHA3 Accelerator. Technical Report UCB/EECS-2015-204, EECS Department, University of California, Berkeley, Oct 2015. (Cited on page 23.)

[Sin11] Deshanand Singh. Implementing FPGA design with the OpenCL standard. *Altera whitepaper*, 2011. (Cited on page 21.)

[SJ08] Paul Schumacher and Pradip Jha. Fast and accurate resource estimation of RTL-based designs targeting FPGAS. In *2008 International Conference on Field Programmable Logic and Applications*, Heidelberg, Germany, 2008. IEEE. (Cited on pages 27, 42, 43, and 48.)

[SPLDC19] Alessandro Savino, Michele Portolan, Regis Leveugle, and Stefano Di Carlo. Approximate computing design exploration through data lifetime metrics. In *2019 IEEE European Test Symposium (ETS)*, 2019. (Cited on page 30.)

[SR18] R. Shathanaa and N. Ramasubramanian. Design Space Exploration for Architectural Synthesis—A Survey. In *Recent Findings in Intelligent Computing Techniques*. Springer Singapore, Singapore, 2018. (Cited on pages 31 and 36.)

[SW20] Benjamin Carrion Schafer and Zi Wang. High-Level Synthesis Design Space Exploration: Past, Present, and Future. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, October 2020. (Cited on pages 13, 14, 24, 26, 29, 31, 32, 33, and 61.)

[TL12] Tim Todman and Wayne Luk. Reconfigurable Design Automation by High-Level Exploration. In *2012 IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors*, Delft, Netherlands, July 2012. IEEE. (Cited on page 28.)

[VPNH10] Jason Villarreal, Adrian Park, Walid Najjar, and Robert Halstead. Designing Modular Hardware Accelerators in C with ROCCC 2.0. In *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, Charlotte, NC, USA, 2010. IEEE. (Cited on page 21.)

[WAGM+19] Linus Witschen, Muhammad Awais, Hassan Ghasemzadeh Mohammadi, Tobias Wiersema, and Marco Platzner. CIRCA: Towards a modular and extensible framework for approximate circuit generation. *Microelectronics Reliability*, August 2019. (Cited on pages 24, 33, 72, and 77.)

[WIS+18] Edward Wang, Adam Izraelevitz, Colin Schmidt, Borivoje Nikolic, Elad Alon, and Jonathan Bachrach. Hammer: Enabling reusable physical design. In *Workshop on Open-Source EDA Technology (WOSET)*, 2018. (Cited on page 129.)

[WLZ17] Shuo Wang, Yun Liang, and Wei Zhang. FlexCL: An Analytical Performance Model for OpenCL Workloads on Flexible FPGAs. In *Proceedings of the 54th Annual Design Automation Conference 2017*, Austin TX USA, June 2017. ACM. (Cited on page 35.)

[WMH+15] Skyler Windh, Xiaoyin Ma, Robert J. Halstead, Prerna Budhkar, Zabdiel Luna, Omar Hussaini, and Walid A. Najjar. High-Level Language Tools for Reconfigurable Computing. *Proceedings of the IEEE*, March 2015. (Cited on page 21.)

[WSL+20] Deshya Wijesundera, Kushagra Shah, Kisaru Liyanage, Alok Prakash, Thambipillai Srikanthan, and Thilina Perera. Technique for Vendor and Device Agnostic Hardware Area-Time Estimation. In *Applied Reconfigurable Computing. Architectures, Tools, and Applications*. Springer International Publishing, 2020. (Cited on page 28.)

[WWP09] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 2009. (Cited on page 113.)

[Xil16] Xilinx. 7 Series FPGAs Configurable Logic Block. https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf, 2016. [ONLINE] Last accessed on 21st september, 2021. (Cited on page 5.)

[Xil18] Xilinx. 7 Series DSP48E1 Slice User Guide. https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf, 2018. [ONLINE] Last accessed on 21st september, 2021. (Cited on page 50.)

[Xil19] Xilinx. Vivado HLS. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug902-vivado-high-level-synthesis.pdf, 2019. [ONLINE] Last accessed on 3rd november, 2021. (Cited on pages 21 and 42.)

[Xil21a] Xilinx. Vivado Memory Usage. https://www.xilinx.com/products/design-tools/vivado/memory.html, 2021. [ONLINE] Last accessed on 21st september, 2021. (Cited on page 86.)

[Xil21b] Xilinx. Xilinx Power Estimator. https://www.xilinx.com/products/technology/power/xpe, 2021. [ONLINE] Last accessed on 27th october, 2021. (Cited on page 30.)

[XK96] M. Xu and F.J. Kurdahi. Area and timing estimation for lookup table based FPGAs. In *Proceedings ED&TC European Design and Test Conference*, Paris, France, 1996. IEEE Comput. Soc. Press. (Cited on page 28.)

[YHC+21] Hanchen Ye, Cong Hao, Jianyi Cheng, Hyunmin Jeong, Jack Huang, Stephen Neuendorffer, and Deming Chen. ScaleHLS: Scalable High-Level Synthesis through MLIR. *arXiv:2107.11673 [cs]*, August 2021. (Cited on pages 24, 33, 72, and 77.)

[YSR07] Peter Yiannacouras, J. Gregory Steffan, and Jonathan Rose. Exploration and Customization of FPGA-Based Soft Processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, February 2007. (Cited on page 22.)

[ZFS+17] Jieru Zhao, Liang Feng, Sharad Sinha, Wei Zhang, Yun Liang, and Bingsheng He. COMBA: A comprehensive model-based analysis framework for high level synthesis of real applications. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Irvine, CA, November 2017. IEEE. (Cited on page 35.)

[ZFS+20] Jieru Zhao, Liang Feng, Sharad Sinha, Wei Zhang, Yun Liang, and Bingsheng He. Performance Modeling and Directives Optimization for High-Level Synthesis on FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, (7), July 2020. (Cited on pages 28 and 35.)

[Zha08] Zhang, Zhiru and Fan, Yiping and Jiang, Wei and Han, Guoling and Yang, Changqi and Cong, Jason. *AutoPilot: A Platform-Based ESL Synthesis System*. Springer Netherlands, Dordrecht, 2008. (Cited on page 21.)

[ZPL+16] Guanwen Zhong, Alok Prakash, Yun Liang, Tulika Mitra, and Smail Niar. Lin-analyzer: a high-level performance analysis tool for FPGA-based accelerators. In *Proceedings of the 53rd Annual Design Automation Conference*, Austin Texas, June 2016. ACM. (Cited on pages 28, 29, and 35.)

[ZPW+17] Guanwen Zhong, Alok Prakash, Siqi Wang, Yun Liang, Tulika Mitra, and Smail Niar. Design Space exploration of FPGA-based accelerators with multi-level parallelism. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, Lausanne, Switzerland, March 2017. IEEE. (Cited on pages 29 and 33.)

Leveraging Hardware Construction Languages for Flexible Design Space Exploration on FPGA

Abstract — FPGA based accelerators are imposing themselves as energy efficient alternatives to general purpose CPUs. However, the hardware development methodologies are still way behind their software counterparts, and initiatives are to be taken in order to increase the productivity of hardware developers. In this thesis, we explore the possibilities that the emerging Hardware Construction Languages paradigm can bring to the hardware world, notably by leveraging high level features such as functional programming or object oriented development. We start with a comprehensive analysis of the estimation metrics and methodologies in the context of FPGA development, and then put a particular focus on how such paradigm can be used for design space exploration, introducing two complementary methodologies — *meta design* and *meta exploration* — for such usage. A software demonstrator, QECE, has been developed and used to demonstrate the usability of those methodologies in various use cases, thanks to a custom benchmark made of representative applicative kernels.

This thesis is an initiative to enhance hardware developers expressivity, providing them with powerful features such as functional programming and object-oriented development.

Keywords: *FPGA, hardware accelerators, Chisel, design space exploration*

Utilisation de langages de construction matérielle pour une exploration flexible des espaces de conception sur FPGA

Résumé — Les accélérateurs matériels à base de FPGA s'imposent actuellement comme une alternative à haute efficacité énergétique aux processeurs généralistes classiques. Cependant, les méthodologies de développement matériel souffrent d'un grand retard par rapport à leurs pendants logiciels, et des initiatives sont nécessaires afin d'accroire la productivité des concepteurs matériels. Dans cette thèse, nous explorons les possibilités que les nouveaux langages de construction matérielle ouvrent pour le monde de la conception numérique, notamment en permettant l'usage de fonctionnalités de haut niveau telles que la programmation fonctionnelle ou le développement orienté objet. Nous proposons tout d'abord une analyse de différentes métriques et méthodologies d'estimation pour le développement sur FPGA, et nous intéressons ensuite plus particulièrement à ce que ces nouveaux langages peuvent apporter au domaine de l'exploration d'espace de conception, en introduisant deux méthodologies complémentaires: la *méta conception* et la *méta exploration*. Un logiciel démonstrateur, nommé QECE, est développé et utilisé afin de démontrer l'utilisabilité de ces méthodologies sur différents cas d'utilisation, grâce à un ensemble de noyaux applicatifs que nous avons développés.

Cette thèse est une initiative pour améliorer l'expressivité des développeurs matériels, en leur fournissant des fonctionnalités à fort potentiel telles que la programmation fonctionnelle ou le développement orienté objet.

Mots-clés: *FPGA, accélérateurs matériel, Chisel, exploration d'espace de conception*

Thesis prepared at TIMA laboratory
46 Avenue Félix Viallet, 38031, GRENOBLE Cedex, France