Design Space Exploration in High-Level Synthesis

Doctoral Dissertation submitted to the
Faculty of Informatics of the Università della Svizzera Italiana
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Lorenzo Ferretti

under the supervision of
Prof. Laura Pozzi

October 2020

Dissertation Committee

Prof. Nikil Dutt	University of California Irvine, Irvine, United States.
Prof. Paolo Ienne	École polytechnique fédérale de Lausanne, Lausanne, Switzerland.
Prof. Cesare Alippi	Università dalla Svizzera italiana, Lugano, Switzerland.
Prof. Antonio Carzaniga	Università dalla Svizzera italiana, Lugano, Switzerland.

Dissertation accepted on 30 October 2020

Research Advisor	PhD Program Director
Prof. Laura Pozzi	**Prof. Walter Binder, Prof. Silvia Santini**

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Lorenzo Ferretti
Lugano, 30 October 2020

To my beloved

The best way to predict the future
is to invent it.

Alan Kay

Abstract

High Level Synthesis (HLS) is a process which, starting from a high-level description of an application (C/C++), generates the corresponding RTL code describing the hardware implementation of the desired functionality. The HLS process is usually controlled by user-given directives (e.g., directives to set whether or not to unroll a loop) which influence the resulting implementation area and latency. By using HLS, designers are able to rapidly generate different hardware implementations of the same application, without the burden of directly specifying the low level implementation in detail.

Nonetheless, the correlation among directives and resulting performance is often difficult to foresee and to quantify, and the high number of available directives leads to an exponential explosion in the number of possible configurations. In addition, sampling the design space involves a time-consuming hardware synthesis, making a brute-force exploration infeasible beyond very simple cases. However, for a given application, only few directive settings result in Pareto-optimal solutions (with respect to metrics such as area, run-time and power), while most are dominated. The design space exploration problem aims at identifying close to Pareto-optimal implementations while synthesising only a small portion of the possible configurations from the design space.

In this dissertation I present an overview of the HLS design flow, followed by a discussion about existing strategies in literature. Moreover, I present new exploration methodologies able to automatically generate optimised implementations of hardware accelerators. The proposed approaches are able to retrieve a close approximation of the real Pareto solutions while synthesising only a small fraction of the possible design, either by smartly navigating their design space or by leveraging prior knowledge. Herein, I also present a database of design space explorations whose goal is to help researchers to design new strategies by offering a reliable source of knowledge for machine learning based approaches, and standardise the methodologies evaluation. Lastly, the stepping-stones of a new approach relying on deep learning strategies with graph neural networks is presented, and final remarks about future research directions are discussed.

Acknowledgements

Contents

Contents **xi**

1 Introduction **1**

2 Hardware Design Evolution **3**
2.1 RTL-Design Flow 5
2.2 High-Level Synthesis Revolution 6
2.2.1 Evolution of HLS 7
2.2.2 The HLS Process 10
2.2.3 HLS Optimisations 14
2.3 New Challenges 20

3 The Design Space Exploration Problem **23**
3.1 Terminology 23
3.2 Problem Formulation 25
3.3 Metrics 28

4 Related Works **31**
4.1 Model-based Strategies 33
4.2 Black-box-based Strategies 35
4.2.1 Learning-based strategies 36
4.2.2 Refinement-based strategies 37

5 Refinement-Based Strategies **39**
5.1 Cluster-Based Heuristic 39
5.1.1 Exploration Methodology 43
5.1.2 Results 48
5.2 Lattice Search 56
5.2.1 Exploration Methodology 57
5.2.2 Results 64

6 Transfer Learning Driven Design Space Exploration **73**
6.1 Leveraging Prior Knowledge 73
6.1.1 Standard Approach VS Leveraging Prior Knowledge 76
6.1.2 Results 86
6.2 A Database of Design Space Explorations 94
6.2.1 DB4HLS Infrastructure 96
6.2.2 A Domain-Specific Language for DSEs 97
6.2.3 A Framework for Parallelising HLS Runs 99

7 Is Deep Learning a Viable Solution? **101**
7.1 Graph-Based Deep Learning for DSE. 102
7.1.1 Graph Representation of HLS Designs 103
7.1.2 Graph Neural Network for HLS 108
7.1.3 Challenges 110

8 Conclusion **113**
8.1 Contributions during the Ph.D. 114
8.2 What's next? 117

Bibliography **119**

Chapter 1

Introduction

The constant growth in performance and area/energy efficiency requirements of everyday applications has, in the last decades, influenced researchers to design more and more performing and efficient specialised hardware. Hardware accelerators have emerged as a viable solution to satisfy such requirements and address the end of Moore's law [77] and the breakdown of Dennard scaling [35].

However, hardware design is a complex process. It requires the description of millions of transistors that have to work in parallel in order to perform complicated tasks. Traditionally, Integrated Circuit (IC) methodologies have relied on Hardware Description Languages (HDLs) such as VHDL and Verilog in order to describe the logical components and their interaction at a Register Transfer Level (RTL). Nonetheless, the HDL design flow does not scale for large applications. The HDLs main drawback is that they require to concurrently define both functionality and implementation from a low-level view. Therefore, such approaches cannot easily target complex applications or support the generation of circuit variants with different area and latency targets, as often required in case of different performance and cost constraints.

To cope with these shortcomings, High-Level Synthesis (HLS) tools such as LegUp [12], ROCCC [98], SPARK [116], and Xilinx VivadoHLS [126] take a more abstract stance, allowing the design of ICs from high-level specifications. By using HLS tools, designers can guide the design process by applying directives able to steer the resulting RTL implementation according to the desired performance and requirement goals.

While HLS fostered a revolution in hardware design, it opened a new series of challenges. In fact, while HLS allows to easily define vast design spaces for a given hardware specification, determining the performance (latency) and resource requirements (area, power) of each implementation still implies time-

consuming syntheses. Moreover, the number of possible implementations of a design grows exponentially with the number of applied directives, while, in general, only a few of them are Pareto-optimal–i.e., they show the best cost/performance tradeoff–from a performance and resources perspective.

Various HLS-driven Design Space Exploration (DSE) strategies have been proposed to identify (or approximate) the set of Pareto implementations while minimizing the number of synthesis runs. These approaches aim at imitating the behaviour of the HLS tools to pre-estimate the effect of the HLS directives and guide the HLS exploration process. While mandating very few synthesis runs, such strategies struggle when coping with multiple, interdependent optimizations. Hence, they are often limited to capturing the effect of only a few directives.

Herein, I present the methodologies I have devised during my doctoral studies. My contributions are characterized by novel problem formulations that can be exploited to reduce the problem complexity by focusing the exploration only on portion of the design space. The proposed formulations achieve this goal by performing local searches, either dividing the problem in subproblem or exploiting a locality property of the design spaces. I also showcase the possibility of leveraging prior knowledge from past DSEs to effectively infer optimal implementations for new target designs. Then, I have contributed to the field with a database of DSEs whose purpose is to offer designers a reliable source of knowledge for future exploration methodologies relying on machine learning strategies and standardise the evaluation process of the existing and future ones.

Moreover, the possibility of using Deep Learning to address DSEs is discussed. I present a Graph Neural Network model aiming at learning the set of HLS directives resulting into Pareto-optimal implementations from an abstract representation of HLS-application in the form of graphs (i.e., simplified control data flow graphs). Lastly, the results of my doctoral studies are summurised, the final considerations are discussed, and possible future research directions are presented.

This document is structured as follows: Chapter 2 describes the limitations of the traditional hardware design flow, the improvements, but also the the challenges introduced by HLS tools. Chapter 3 formalise the DSE problem and the elements characterising it. Chapter 4 describes the state of art in the domain of DSE problems for hardware design and in particular HLS-driven DSEs. Chapter 5 and 6 present my contribution to this field, and Chapter 7 introduces a new approach involving deep learning to address the DSE task. Lastly, Chapter 8 concludes this document summarising the results obtained and discussing final remarks.

Chapter 2

Hardware Design Evolution

In 1965 Gordon Moore made one of the most famous predictions in technology: the number of transistors in Integrated Circuit (IC) double every two years. His prediction, namely Moore's Law [78], has been used by semiconductor companies to plan technological advancement for years. As a consequence, designers had to devise hardware design techniques able to keep pace with the increasing number of transistors and the available computational resources.

Register-Transfer Level (RTL) became the dominant method to describe ICs [61]. However, the constant growth in the number of processing elements in everyday devices and the increasing complexity of design functionalities have also grown exponentially, causing the design and verification processes of RTL designs to become a bottleneck for productivity [52]. Figure 2.1 shows the predicted trend in the processors count in portable devices made by the International Technology Roadmap for Semiconductors (ITRS). The figure highlights a trend that will to burden the designers' activity in the not distant future.

Moreover, Moore's prediction, that guided the computer industry for over 50 years, appears to not be valid anymore [123][69][57]. The main reason for this can be identified in the breakdown of Dennard's Scaling. In 1974 Dennard observed that as transistors became smaller and smaller, their power density remains constant [28]. Companies reacted to this discover designing faster and faster circuits without significantly affecting the ICs power requirements. However, the breakdown of Dennard's scaling law forced microprocessor companies to identify alternative strategies other than produce faster ICs. In order to satisfy the growing performance and power requirements, designers have identified heterogeneity as a viable alternative to obtain high-performance and energy-efficient hardware. In this context, Application Specific Integrated Circuits (ASICs) able to solve specific tasks have gained a lot of popularity, becoming funda-

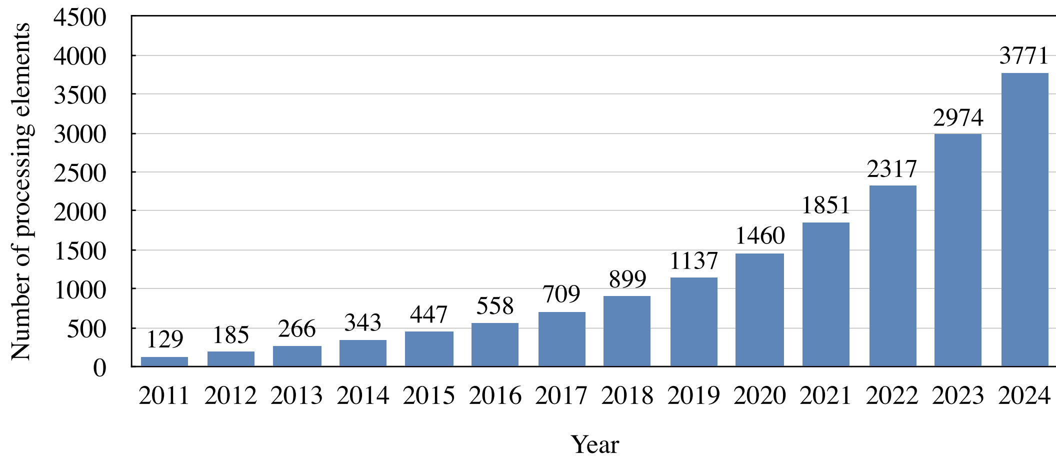


Figure 2.1. ITRS growth prediction of the number of Processing Elements in consumer devices[52].

mental to solve complicated tasks [55][93], and outlining the trend of hardware design [83][8].

While heterogeneous architectures and specialised hardware can boost performance and energy efficiency, the variety of processing elements keeps burdening the design process. While observing Figure 2.1, it is important to notice that not only the number of processing elements per device is estimated to grow, but, due to the heterogeneity, also the variety of processing elements in a device increases. This aspect decreases even more design productivity, since multiple different specialised hardware need to be designed.

To compensate for this drawback, Electronic System-Level (ESL) design automation has been identified as the next step in the hardware design process, with High-Level Synthesis (HLS) tools being a viable solution [72][18][23][66].

HLS allows designers to both speed-up the design phase and raise the abstraction level of it. With HLS designers do not have to provide an RTL description of the design functionality by using Hardware Description Languages (HDL) such us Verilog or VHDL. HLS tools take in input a C/C++ or SystemC specification which is automatically transformed into a cycle-accurate RTL specification implementing the application functionality. Moreover, HLS enables to easily target different technology such as ASICs or Field Programmable Gate Arrays (FPGAs). In addition, HLS allows behavioural verification of the programs at C/C++ level using software verification tools that are faster and simpler than RTL ones, accelerating the this process.

Furthermore, HLS allows the rapid generation of circuits variants. Microar-

chitectural variation can be explored without modifying the original software by using specific HLS directives affecting the resulting implementation. This feature is one of the main advantages of the HLS design flow with respect to the RTL one, allowing the rapid prototyping of hardware accelerators with different costs and performance requirements.

These benefits, by affecting design and verification time, development costs, learning curve of hardware, and time-to-market of the generated hardware designs, cause the hardware acceleration on heterogeneous platform to become a more and more attractive and widely adopted solution [61].

In the next subsections, an overview of the traditional design flows and of the HLS one used to generate hardware accelerators are presented. In Section 2.1 I will briefly discuss the characteristics of the RTL-design flow; then, in Section 2.2 the HLS process and the evolution of the HLS tools from their origin to the present are discussed. Lastly, Section 2.3 presents the challenges introduced by the adoption of the HLS design flow and the domain related research questions.

2.1 RTL-Design Flow

The evolution of design automation and of Computer Aided Design (CAD) tools have boosted the productivity of hardware design processes. Design automation and verification are the keys to the effective use of large scale integrated circuit technology [43]. Design automation tools perform all tasks that were originally performed manually. Simulation and verification of the design functionality and synthesis are few examples. These steps, however, still require a description of the underlying circuit functionality and structure to operate. Designers, by using Hardware Description Languages (HDLs), can define the features, microarchitecture functionalities, and specification requirements of the target hardware implementation.

HDLs enable designers to define the hardware components' functionality as a set of operations on the data. These operations transform the original data, and move them from source storage units (e.g., registers) to destination ones. Thus, these types of specifications are named Register-Transfer-Level (RTL) descriptions. RTL descriptions require designers to specify the logical structure of the circuit using logic and arithmetic operators, to define operations at bit and word-level granularity. Moreover, conditional statements can be used to describe control flow behaviours.

The drawback of RTL-languages is that they do not scale well for large applications. While RTL-languages raised the level of abstraction with respect to

manual circuit design, RTL-descriptions still require to concurrently define both the functionality and the implementations at a low level of details. The definition of arithmetic operations and data transfers at RTL-level implies manual description of the functionality behaviour in a timed-manner. Thus, all performed operations have to be described cycle-by-cycle. This aspect makes RTL languages extremely error-prone, requiring advanced hardware design expertise and imposing a steep learning curve for them. A study [128] showcased how 30K-40K lines of C/C++ code for a 1M-gate design may result in about 300K lines of RTL code to implement the same functionality. This example suggests the necessity to raise the abstraction level of the design process to reduce design errors, development time, cost, and the time-to-market of new designs.

2.2 High-Level Synthesis Revolution

To cope with RTL-design limitations, High-Level Synthesis (HLS) has been introduced. HLS tools, starting from a high-level behavioural description of the software functionality (C/C++ or System C), automatically produce the RTL description of the desired IC. The generated hardware component can then be seamlessly synthesised with ASIC or FPGA toolchains in order to implement the corresponding hardware accelerator.

HLS merges the benefits of software design productivity with the performance and efficiency of hardware. It enables software designers to access hardware performance without actually building hardware design expertise [82]. Similarly, it offers to hardware engineers the design productivity and the level of abstraction typical of software, allowing rapid exploration of micro-architectural variations, an extremely important aspect while targeting complex systems with strict performance cost requirements [66].

The HLS process, relying on high-level programming languages, allows a dramatic improvement in design productivity. Considering the same example from the study discussed in Section 2.1, the lines of code required by the HLS-design flow is 7X-10X less than the one needed by the RTL implementation of the same functionality [128].

Moreover, the definition of functionalities at behavioural-level leads to the diffusion of behavioural intellectual properties (IPs) reuse. The modular nature of the IPs, which, differently from RTL descriptions are not constrained to fixed architectural and interface protocols, allows them to easily be retargeted to different technologies or system requirements .

Moreover, HLS has raised particular interest in the FPGA community. The

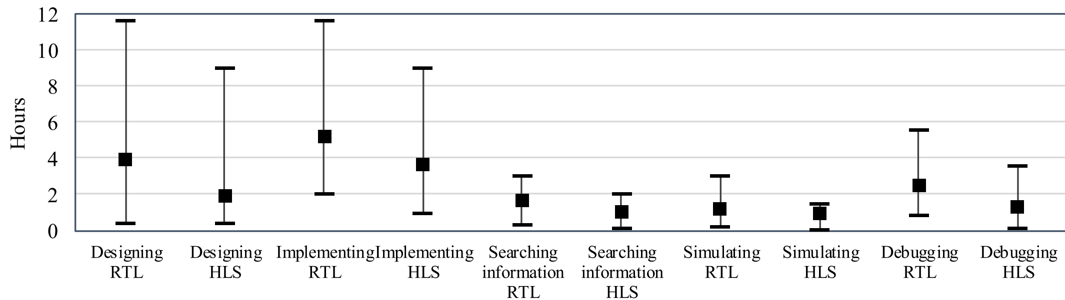


Figure 2.2. Comparison of HLS and RTL design flow productivities across different metrics. The boxplots show the maximum, minmum and average time for the different metrics evaluated. The figure has been published in [61].

reconfigurability of FPGAs well fits with the benefits introduced by HLS tools. In particular, the possibility to rapidly generate different IPs implementing a large variety of functionalities, and the efficient estimate of their performance and costs, make the combination of FPGA and HLS an extremely appealing solution for the design and deployment of complex systems on FPGA architecture [33].

In a recent work, Lahti et al. [61] have analysed different metrics comparing the HLS design flow with respect to the traditional RTL one. They have compared in particular the quality of result and productivity differences between the two. Their findings showcased that while the quality of results of the RTL flow is still better than state-of-the-art HLS tools, the average development time with HLS tools is much faster. The paper shows that a designer can achieve four times more productivity with HLS [61] with respect to the RTL flow. Figure 2.2, from [61] exhibits these differences. The chart shows the time required to implement different designs with the HLS and the RTL design flow. For all the different metrics considered in the study, the HLS flow resulted in higher productivity.

2.2.1 Evolution of HLS

The origin of HLS can be traced to the early 1970s. At that time, Carnegie Mellon researchers built a revolutionary tool (CMU-DA [30][88]) which, using the Instruction Set Processor Specification (ISPS) language [3], allowed the description of a design at a behavioural level. By generating an intermediate representation, common code-transformation techniques were applied, and the concepts of datapath allocation, module selection, and controller generation step of the modern synthesis tools were introduced for the first time. That work, despite little consideration from the industry [72], raised considerable research interest

[18].

The next decade, 1980-1990, has seen the proliferation of a number of different research HLS tools (e.g., ADAM [44], HAL [90], MIMOLA [74], and Hercules [27]), and few industrial ones (Cathedral [26], Yorktown Silicon Compiler [11] and BSSC [139]). These works, similarly to modern tools, decompose the synthesis process in different steps: code transformation, module selection, operation scheduling, datapath allocation, and controller generation.

Code transformation is used to transform the original code into a semantically equivalent version of it, able to better expose the program characteristics and allowing basic optimisations. The module selection phase selects, from a library of different components, the particular functional unit used to implement each operation in the code–e.g., it selects the most appropriate hardware multiplier to implement a multiplication among floats given a certain clock constraint. Operation scheduling assigns each operation into a specific cycle or control state. Then, datapath allocation defines the type and number of hardware resources needed to satisfy the design constraints, and, lastly, controller generation applies all the design decisions to generate an RTL model to be synthesised. All these steps were the cornerstone for the next generations of HLS tool.

A common characteristic of these tools was the adoption of custom languages to define the design specifications. Some of them used extensions of existing languages–i.e., Hercules [27] used HardwareC [60] based on the existing C language –while many others opted for custom languages oriented to domain-specific applications. Despite these tools being the ancestor of the modern generations, only a few of them were widely adopted and obtained large consensus. Among them, the Cathedral project [26] had a long evolution and culminated in its commercialisation and the OptimoDE tool from ARM [17]. However, the poor quality of the results, the domain specialisation of most of the tools, the lack of a comprehensive design language and the creation of many custom ones, combined with the adoption of RTL synthesis tools, spread the idea that behavioural synthesis could not fill the productivity gap[72].

In the next decade, 1990-2000, thanks to the improvement in RTL synthesis tools and the wide adoption of RTL-based designs, the major Electronic Design Automation (EDA) companies have invested in commercial HLS tools. Proprietary tools from Synopsys with Behavioural Compiler [59], Cadence with Visual Architect [49] and Mentor Graphics with Monet [34] entered the market. While these tools were able to raise the interest of industries, they could not replace the RTL design flow. In particular, the necessity of using behavioural description languages as input of the HLS tool, and the steep learning curve of those, limited their diffusion. Moreover, the HLS design flow was still missing important

aspects. Existing tools failed to recognise the difference between data-flow and control flow, often focusing on only one of the two aspects resulting in poor quality of results. Lastly, EDA were not able to correctly foresee potential user of HLS. These tools were thought for current RTL designers, who were skeptical about the newcoming design flow and relied on the quality of results of the traditional RTL-design flow [23]. This aspect made clear that it was necessary to raise the abstraction level of design languages, and to enlarge the pool of users to increase the diffusion of HLS tools.

The rise in abstraction level is the characterising aspect of the current generation of HLS tools. The 21st century has mainly seen the evolution of existing behavioural CAD tools [13][119][24][117]. Despite different opinions [32][99], EDA companies and researchers realised the necessity to adopt a high-level input language, more accessible to algorithm and designers than HDLs. The choice fell on C/C++ and C-like languages–e.g., SystemC. This had a huge impact on the diffusion of HLS tools. It expanded the pool of users for HLS tools and allowed researchers to leverage the most recent compiler technologies and optimisation. This last aspect consequently resulted in an improvement in the quality of result of the HLS tools, making them more reliable and appealing to RTL-designers.

However, despite the large diffusion of C and C++ as dominating input languages, these are used with limitations. Current HLS tools cannot exploit all the features of such languages such as pointers, dynamic memory management, recursion and polymorphism, which can hardly be mapped to a hardware implementation. Moreover, C/C++ lacks some aspects such as definition of variable bit-width, timing, synchronisation, characterising the hardware design. To cope with this, many language extensions and libraries have been proposed [60][42] [120] and restriction to the C input programs have been introduced. Among these strategies, the use of pragmas, directives, and the adoption of a subset of ANSI C/C++ had large diffusion. By using pragmas and directives the HLS process can be guided, and standard compiler techonology can be adopted. This modularity allows to seamlessly move from software to hardware design in the direction of the hardware/software codesign goal.

Another important factor that favoured the diffusion of HLS tools was the growth of the FPGA market and the diffusion of this technology [124]. FPGAs, differently from ASICs, require different implementation programming models and have different design criteria. The limited resources available on a reconfigurable chip force designers to explore different architectural choices able to fit area and performance constraints. The possibility to easily explore these costs and performance tradeoffs with the use of pragmas to govern the HLS process made HLS tool an appealing solution for designers targeting FPGA technology.

Many HLS tools flourished in the 2000 decade belong to this category like SPARK [116], ROCCC [98], Trident [125], Handel-C [46] among many.

A more recent evolution of HLS tools has seen the diffusion of research-oriented projects as Bambu [92], LegUp [12] and AutoPilot [144], with few of them, Autopilot and LegUP, being acquired by EDA companies or commercialised respectively. Among commercial tools, VivadoHLS [126], CatapultC [13], Bluespec [7] and CtoS [117] had large diffusion. Lastly, the 2015 acquisition of Altera from Intel lead to the creation of the recent Intel HLS Compiler [53], and the recent integration of FPGAs on the Amazon Web Services architecture has seen the adoption of the Xilinx design suite SDAccel [109].

2.2.2 The HLS Process

In the previous sections, we have mentioned the benefit of the HLS design flow and the evolution that HLS tools had from the 70s until today. Herein, I will describe the HLS process's details and the steps performed by synthesis tools to automatically generate an RTL implementation starting from a C/C++ design. An experienced reader, who is already aware of the HLS process details, may want to skip this section and move to the next ones.

The HLS process provides many benefits to designers, it allows, starting from a high-level description of an application, the rapid generation of optimized RTL specifications. By focusing only on the behavioural aspect of the functionality to implement in hardware, designers can reason on *what* to implement instead of *how* to achieve that. Moreover, HLS enables rapid exploration of micro-architectural variations through the use of optimisation directives. Thus, multiple variants of the same circuit functionality can be seamlessly implemented satisfying different cost and performance requirements. Besides, the adoption of HLS simplifies the verification process and the portability of the generated IPs.

To generate an RTL implementation, the HLS process requires a high-level specification of the functionality to implement in hardware, an RTL component library, and a set of design constraints–e.g., target architecture, clock period. Given these elements, the HLS process relies on five different steps: compilation, resource allocation, operation scheduling, binding, and control logic generation. All these steps break down the original behavioural representation, extract the operations and variable describing the functionality, and use this information to generate the final RTL implementation.

In the following, the different steps are detailed, and a few practical examples are provided. Figure 2.3 shows an overview of the HLS-design flow, including the required input, the process steps, and the generated output.

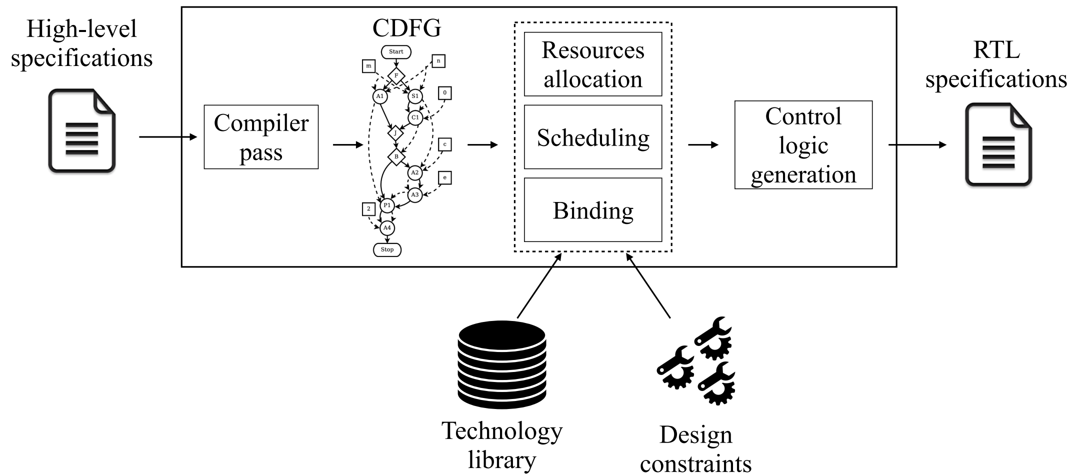


Figure 2.3. Overview of the HLS process.

Compiler pass

The input specifications are processed by a compiler pass which generates and abstract representation of the functionality. The compiler extracts the information needed to identify the resources and the operations to be implemented in hardware. The compiler pass defines "what" will be implemented. In this phase, the compiler may introduce code optimisations, either to simplify the code structure–e.g., dead code elimination, constant propagation, and loop transformation may be applied–or to transform the original code into a functional equivalent version that can be mapped to a more efficient design pattern.

Standard abstract representation of the code are Control Data Flow Graphs (CDFGs)[41][84]. CDFGs encompass both the data and control dependency information between operations. These are built extending the Data Flow Graph (DFG) with control flow dependencies in order to represent loop structures and unbounded iterations. The CDFG nodes and the edges identify the set of operations and variables needed to generate the RTL implementation. These elements are the input of the allocation, scheduling, and binding steps.

Resources Allocation

Once the variables and the operations have been extracted by the compiler, the resources required to map them to hardware are defined. Allocation defines "who" will perform the identified operations. This step, named resource allocation or simply allocation, defines the type and number of functional units, storage components, busses, and other connectivity elements required to generate

the hardware implementation. The available resources are selected from a library of RTL components, usually dependent on target technology and proprietary tools. This library includes all the information required by the synthesis task to estimate area, power, and latency needed in the scheduling and binding phases.

Scheduling

The scheduling phase maps all the identified operations to a specific clock cycle. This step defines the order in which each operation will occur. Scheduling determines "when" an operation will happen. Design constraints and the technology library have a significant impact on the resulting schedule. E.g., by relaxing the clock period constraint more operation may be scheduled into the same clock cycle, according to technology characteristics. Similarly, using a faster technology, more operations may occur during the same cycle.

In addition to technology and design constraints, the code structure also has an impact on scheduling. For example, the concatenation of operation intermediate results may lead to a more efficient scheduling of the operations.

Figure 2.4 shows examples of scheduling given the resource allocations for two different types of technologies and different design constraints–i.e., clock constraint.

Binding

Binding determines which allocated resource will be used for each operation. Binding defines "where" operations are mapped to the allocated resources. Operators are mapped to functional units, and variables are mapped to storage units. In this phase, decisions about the sharing of the allocated resources are made. According to the performance or power/area requirement, the binding phase will decide to allocate more units or share the functional and storage units already allocated among the different operations and variables. These choices may affect the resources required for the connection of the allocated resources. Therefore, some of the choices affecting the connectivity elements performed during the allocation step may be revised at this stage.

Figure 2.4 scheduling d) shows an example of binding with resource sharing. Resources are re-used during the scheduling in order to minimise the area and reduce the number of functional units.

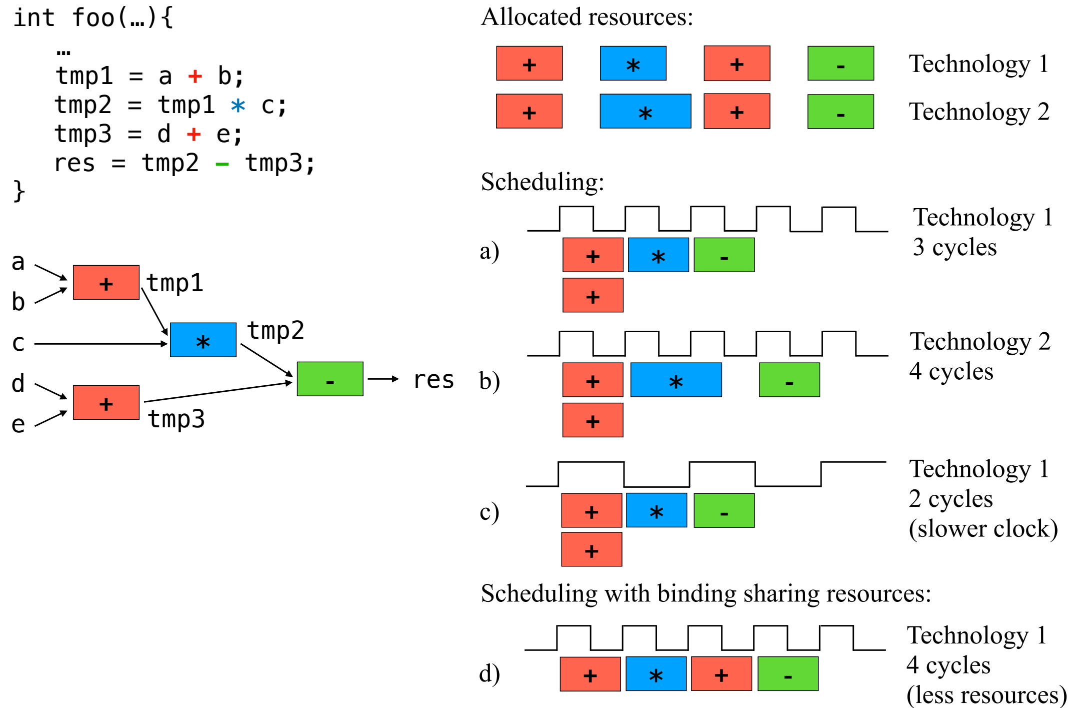


Figure 2.4. Example of resource allocation, scheduling, and binding for the code in the example. Four different scheduling are shown: a) scheduling adopting functional units of technology 1, b) scheduling adopting functional units of technology 2, c) scheduling adopting functional units of technology with a slower clock constraint from the one originally specified by the designer, d) scheduling adopting functional units with binding sharing resource. In the last scheduling example, only one adder is allocated, and the same functional unit is used to compute both `tmp1` and `tmp3` at different clock cycles.

Control Logic Generation

The last step of the HLS process generates the final RTL architecture. The control logic extracted by the compiler pass is used to define the datapath for the functionalities implemented in the previous steps. The datapath includes all the storage elements, functional units, and connection elements defined in the allocation and binding stages according to the defined scheduling. Input, output, and control ports of the design interface are connected to the RTL logic, and a finite state machine implementing the controller unit manages the correct execution of the functionality.

Putting all together

All the above-mentioned steps together generate the resulting RTL. While performing independent tasks, the different phases are affected by the decisions made by each other. In particular, allocation, scheduling, and binding choices are interdependent. For this reason, these three stages are the core of the existing HLS tools and are the ones characterising their quality of results.

The efficiency of these different stages may also impact the target technology of HLS tools. For example, HLS tools targeting FPGAs may be interested in resource-constrained approaches. In this case, the allocation process and a binding sharing the resources will be predominantly imposing more stringent constraints on the scheduling. In other cases, e.g., for time-constrained applications where ASICs are the target technology, more aggressive scheduling will be adopted.

Moreover, these stages are performed in order to satisfy a target objective function, such as the minimisation of area/power or latency. Some HLS tools allow designers to chose the target objective–i.e., Mentor Catapult HLS [13]– while others are optimised for one of the two–i.e., VivadoHLS [126] minimises the latency. However, even for HLS tools where the objective function is predetermined, time and resource requirements can be satisfied by relaxing the design constraints.

While HLS can automatically generate efficient RTL implementation of the given functionality, designers do not directly control its process. To cope with this aspect, HLS tools allow designers to guide the synthesis through HLS directives. These directives, often in the form of compiler's pragmas, directly affect the compiler pass, allocation, scheduling, and binding steps. In the following, a description of the most common optimisations and some practical examples are provided.

2.2.3 HLS Optimisations

HLS tools, by specifying HLS directives, can influence the synthesis process. The directives directly impact the resulting performance and costs, enabling designers to rapidly explore different architectural variations. The number and type of possible optimisations depend on the HLS tool. However, across the different tools available, some macro-categories of optimisations are common. Herein, these will be described, and a few examples of their effect will be discussed. In particular, I will discuss the pragmas available for the VivadoHLS tool [126]. While directives names may differ according to the commercial tool adopted, the

effect they have on the synthesis process is similar.

Spatial Parallelism

One of the most common optimisation is spatial parallelism. Spatial parallelism allows to reduce latency and increase the throughput of an IC. Multiple functional units can be instantiated to concurrently execute the program operations. HLS tools automatically perform instruction-level parallelism during the scheduling and binding phase. However, designers can force the amount of parallelism by explicitly specifying regions of the code to be parallelised. For example, a designer can target a loop with an unroll directive. The directive forces the compiler to unfold the loop body up to a certain factor, or even entirely, imposing to the HLS tool to instantiate the hardware resources required to execute the loop body operations concurrently. Ideally, a loop without loop carried dependencies can be parallelised entirely if enough resources are available. Therefore, the execution time of the entire loop is reduced to the execution time of a single iteration. Usually, spatial parallelism dramatically reduces latency but requires a significant amount of extra resources.

Figure 2.5 shows an example of a loop unrolling directive applied to a loop. The optimisation, specified in the form of pragma in the original code, unrolls the loop by a factor of 2. Therefore, the functional units in the loop's body are doubled, and the total number of iterations required to execute the loop is halved.

Pipelining

Another common optimisation among HLS tools is pipelining. Pipelining can be applied to functions or loops, and it reduces the initiation interval for functions or loops allowing concurrent execution of their operations. In fact, the execution of the next loop or function input can start before the completion of the predecessor's operations. Pipelining requires that data dependency must be respected before moving to the next iteration. Pipelining enables functions and loops to process a new input or loop iteration every N cycles, where N is the Initiation Interval (II). The II specifies the number of clock cycles between successive input processing or loop iterations. The ideal II of 1 specifies that a new input or loop iteration should be processed every cycle. However, the minimum II achievable can be limited by resource constraints or dependencies–i.e., loop carried dependencies. This optimisation, similarly to spatial parallelism, implies the allocation of extra resources required to perform the parallel execution of multiple input/iterations, but it may greatly improve execution time. Compiler

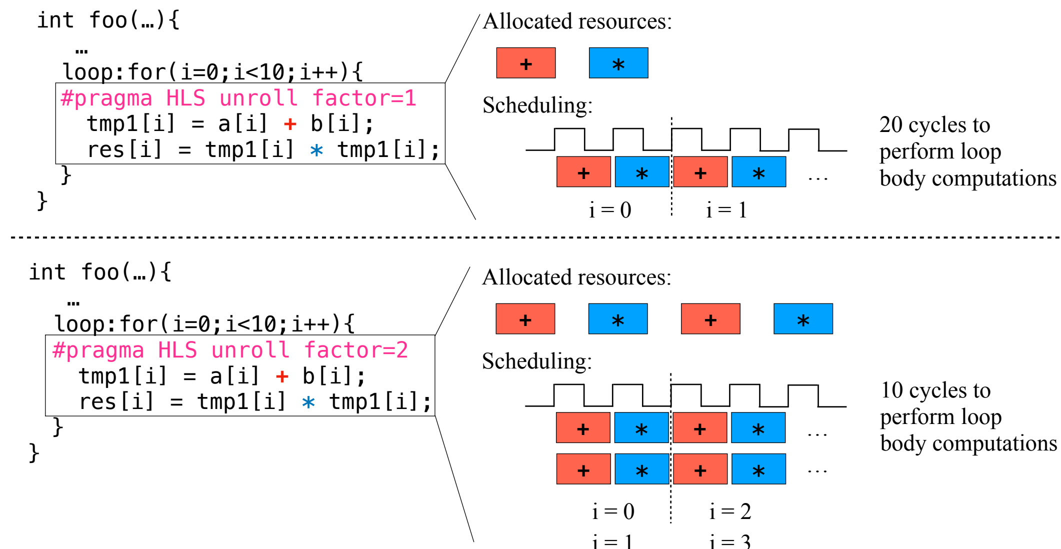


Figure 2.5. Examples of loop unrolling directives applied on a loop. (Top) Allocated resources required by the loop body computation and an example of scheduling for them when no unrolling is applied. (Bottom) Allocated resources and schedule when a loop unrolling factor of 2 is applied. The number of iterations is halved, and the amount of resources is doubled. For simplicity, the computation required to increment the loop iterator variable and check the loop condition are omitted in both examples.

passes can be combined to greatly improve the effectiveness of this optimisation [131][151].

Figure 2.6 shows an example of the pipeline directive applied to a loop. The optimisation, specified in the form of pragma in the original code, defines an initiation interval of 1. The loop body instructions are pipelined allowing the execution of each operation at a new cycle.

Memory Allocation

While implementing a design in hardware, different choices regarding the memory allocation strategy are possible. Private Local Memories (PLM) can be coupled to accelerators to allow direct access to the data needed during computation. In FPGAs, this can be easily achieved by forcing the allocation of multiple memory banks in the form of distributed block RAMs (BRAMs). Using PLM, accelerators can perform multiple memory operations in one cycle according to data distribution over the allocated memories. However, allocation of specific resources is

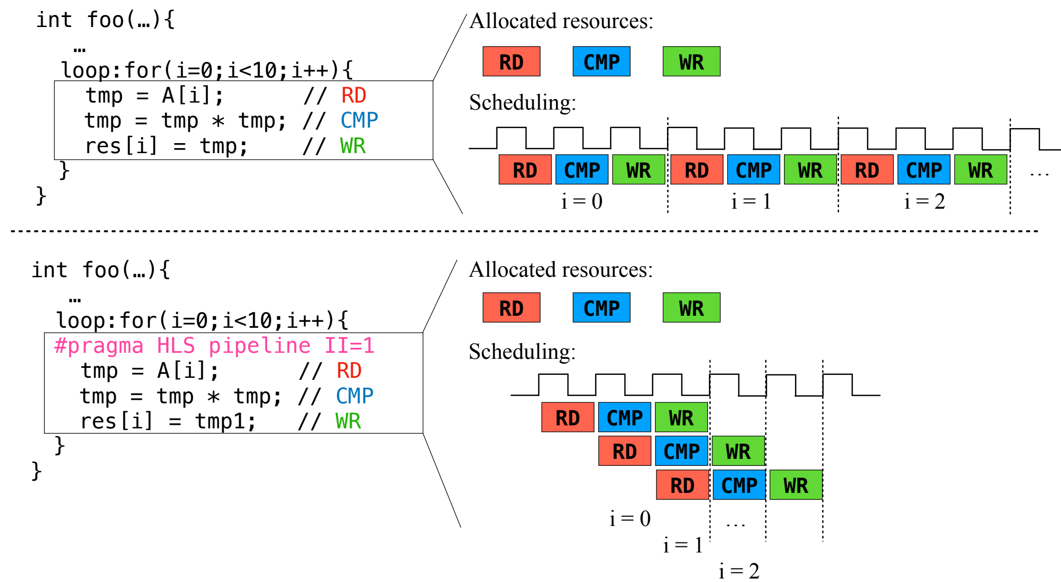


Figure 2.6. Examples of pipelining applied on a loop. (Top) Resource and scheduling required by the loop body without the pipelining directive. Nine clock cycles are required to perform three iterations of the loop. (Bottom) Allocated resources and scheduling with loop pipelining enabled with an initiation interval of one. Five clock cycles are required to perform three iterations of the loop.

required. These resources depend on the available technology library and the target platform. Therefore, available resources may be limited to a certain type and size.

Designers can force the HLS tools to instantiate particular memory elements and allocate design variables to instantiated memories. However, forcing the use of a specific memory element and type can result in suboptimal use of resources if the design variables' size is not aligned to the memory sizes.

Moreover, HLS tools permit to change the physical implementation of the memories by specifying memory partitioning directives. Memories usually have only a limited number of read and write ports, which can limit the throughput of a load/store intensive algorithm. According to the original code's memory access pattern, the bandwidth of an allocated memory element can be increased by partitioning the memories associated with the arrays in the original code. The original array elements can be distributed over multiple smaller memories, effectively increasing the number of load/store ports. While potentially improving the throughput of the design, this optimisation requires more memory instances or

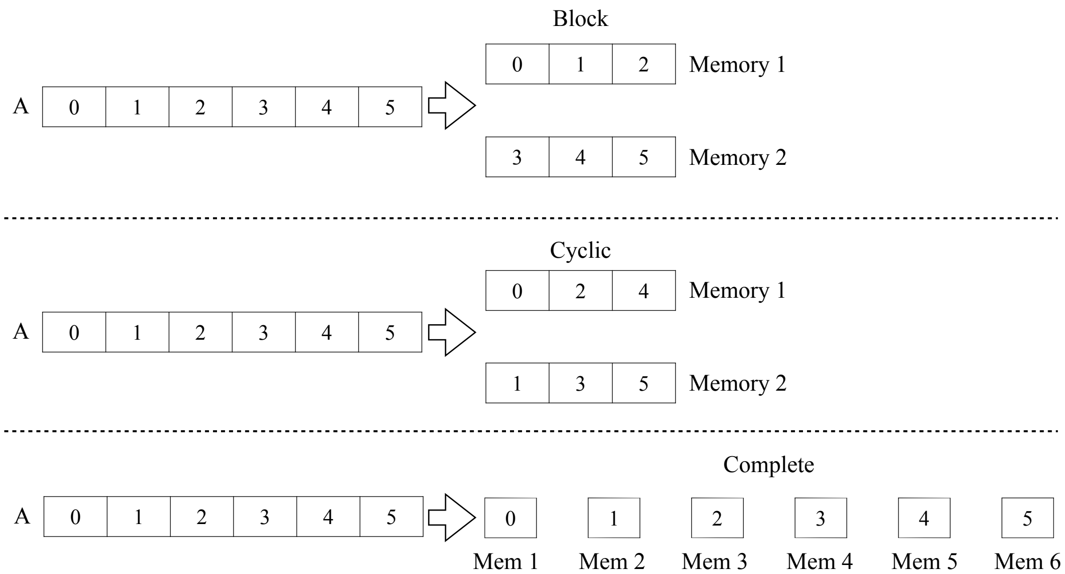


Figure 2.7. Examples of array partitioning directives applied to an array variable. (Top) Block array partitioning with partitioning factor of 2. (Middle) Cyclic array partitioning with partitioning factor of 2. (Bottom) Complete partitioning.

registers and increases the connectivity element among memory and functional units.

Figure 2.7 shows an example of the array partitioning directive applied to three arrays. The figure shows three different types of partitioning strategies–block, cyclic, and complete–and partitioning factors. A block strategy partitions the original memory into equally sized blocks of consecutive elements. A cyclic partition splits the original memory into same size memories interleaving the elements of the associated array. In this case, consecutive array elements are distributed on different memory elements up to a certain partitioning factor. Complete partitioning instead is the most resource expensive approach. It completely decomposes the memory into individuals elements.

Predicated Execution

Software transformations can be applied to allow a parallel schedule of disjoint execution paths. This transformation of the code, often named *if-conversion*, requires the introduction of predicates, or guards, to discriminate at the end of the parallel execution among the concurrent paths' results. The benefits of predicated execution are a higher parallelisation of the code and the elimination of

control dependency in the execution of intensive code structures. This optimisation is extremely effective in the case of balanced branches, while in the case of unbalanced, the execution time could even be slowed down.

Hierarchical-Module Optimisations

By removing or introducing hierarchy among the generated modules, the amount of control logic generated by the synthesis may be affected. For example, enabling function inlining avoids execution and generation of the logic required to manage function invocations. The inlining HLS directive flattens the hierarchy among target function and remaining RTL code. Therefore, the function no longer appears as a separate hierarchy level in the RTL, and a better resource sharing can be achieved. On the other hand, an inlined function cannot be shared, and the reuse of its RTL module is not possible. This optimisation may reduce the overall latency, but more area is usually required to implement the RTL. Similar results can also be obtained for loops by merging the body of the loops.

Bit-width Optimisation

This optimisation specifies the exact amount of bits required by datapath operators and variables. Differently from general-purpose processors, which are designed with a fixed size datapath, HLS tools can generate specialised hardware with custom size operators and registers. This optimisation affects all aspects of the resulting implementations: area, latency, power, and quality of the generated output. It enables to minimise the area of a design while satisfying certain task-related quality of result goals. For example, the implementation of complex machine learning models (i.e., Convolutional Neural Networks) on ASICs and FPGAs relies on the identification of the optimal bit-width for the neural network layers [129]. In this case, reducing the bit-width allows to dramatically diminish the model's size and the memory required to store it. This aspect is extremely beneficial since it permits to deploy complex models on resource-constrained platforms. The main drawback of this approach is related to the effect that bit-width reduction has on the quality of the implemented accelerator's results. In fact, for classification tasks reducing the bit-width significantly affects classification performance. While targeting these applications, designers then have to evaluate the results of the generated hardware to verify that a target classification goal is satisfied.

Resource Library

The allocation step of the HLS process identifies, from a library of RTL components, the resources needed to implement the desired functionality in hardware. Existing libraries are rich and offer several different implementations of the same operation. However, it is hard for designers to foresee the choices made by the HLS tools and estimate the result of the synthesis process. The different components available in these libraries are often vendor-specific or related to the target technology and a characterisation of the library components is not always available.

Through the use of HLS directives, designers can force or prevent the use of specific resources affecting the allocation step, and, consequently, the HLS process results. Enforcing the use of particular resources allows designers to control more tightly the HLS process. Moreover, while designers may have a higher degree of control on the HLS process, the implemented design is generated minimising a target objective during the synthesis. Forcing the use of certain resources may lead to suboptimal choices unless specific resource constraints are required

2.3 New Challenges

The HLS revolution has shaken the belief of hardware designers about the capability of HLS tools. The dramatic gain in the design productivity introduced by the HLS design flow and the quality of results achievable with existing commercial and research tools have consolidated HLS as a viable option for fast prototyping and short-time-to-market [61][18].

However, many of the advantages of HLS are hardly quantifiable with respect to traditional design flows. For example, HLS allowed designers to explore a space of possible optimisations for their design that, before HLS, was hardly treatable. The quality of resulting implementations was dependent on the designer experience, while, nowadays, structured and sound search strategies can be applied to identify effective optimisations.

On the other hand, the possibility of rapidly exploring equivalent versions of the same design has opened a new challenge. Navigating the space of all the possible implementations controlled through HLS directives is a non-trivial task. This problem, namely the Design Space Exploration (DSE) problem, requires the identification of the most effective combination of optimisations for a given design and engineer target requirements. In this dissertation, we focus on this

challenge.

However, DSE is not the only challenge introduced by HLS. Another one is the identification of which portion of the software to accelerate in hardware. Not all software benefits in the same way from hardware acceleration, and complex heterogeneous systems must be properly designed to benefit from heterogeneity and hardware specialisation. To this end, various works have been proposed [143][142], and research in this area is extremely active.

Other challenges involve the improvement of in-system design validation and debugging, better support for domain-specific synthesis, and the possibility of raising the level of abstraction moving to higher level languages (i.e. Python) among others.

Chapter 3

The Design Space Exploration Problem

In this chapter of the dissertation the terminology and a formal definition of the HLS-driven Design Space Exploration (DSE) problem are presented. In addition, a metric, used to evaluate the quality of the DSE and to perform a comparison with respect to the state of the art alternatives is introduced.

Section 3.1 will describe the terminology adopted in this document to discriminate among the different elements of the HLS design flows. In Section 3.2 a formal definition of the DSE problem and the elements characterising it will be presented. Finally, 3.3 will introduce the metric used to evaluate the quality of the DSE methodologies.

3.1 Terminology

An *HLS design* (or *design*) is a functionality to be realized in hardware. For example, a Fast Fourier Transform function and a 2D convolution function from an image processing application are examples of designs that can be separately synthesized with HLS. A *specification* is a high-level description of the design in a programming language such as SystemC or C/C++. The specification, which can be given with an untimed or a loosely-timed model, is the input to the HLS tool. An *implementation* of the design is the output of a run of the HLS tool. This output is typically expressed as an automatically generated RTL code written in Verilog or VHDL. Each implementation is characterized by the values of a performance metric and a cost metric (e.g., latency or throughput as performance metric and, area or energy as costs).

A *synthesis configuration* (or, simply, *configuration*) defines the transformations that a design undergoes through HLS. A designer controls these transformations with constraint and optimization *directives*, such as loop unrolling or

Listing 3.1. Example of specification in C for a toy example design.

```
1 void bar(int input_array[32], int output_array[32]){
2   int i;
3   loop_1:for(i=0; i<32; i++){
4    output_array[i] = input_array[i]*input_array[i];
5   }
6   foo(output_array);
7 }
```

Knob	Location	Type	Sets of values
K_1	input_array	partitioning	{cyclic, block} {1,2,4,8}
K_2	loop_1	unrolling	{1,2,4,8}
K_3	foo()	inlining	{on,off}
…	…	…	…
…	…	…	…
K_n	…	…	$\{V_{n,1}, \ldots, V_{n,m}\}$

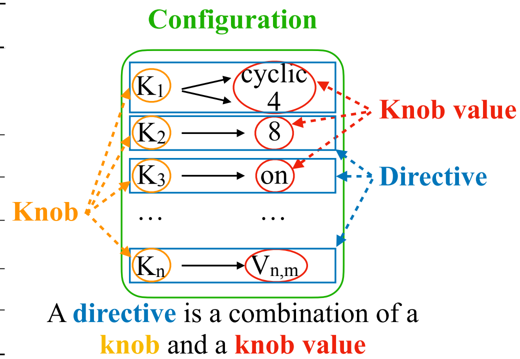


Figure 3.1. Example of terminology associated to the example in Snippet 3.1.

pipelining, array manipulation, and other control and datapath optimizations. A directive is associated with a *location* in the code specification. A location could be either a label in the code or a language construct; for example, a loop or an array declaration. A designer can further customize some directives by specifying the *values* for the directive parameters–for example, the designer can customize the amount of parallelism in the implementation by unrolling a loop a certain number of times or by setting a certain initiation interval for the pipeline implementing the loop.

Each location of a given specification, with an associated directive, encodes a *knob* (a type of directive, the location, and selected parameter values) that the designer considers for the DSE task. For each directive a designer can manually define an admissible set of directive values, and, for directives with multiple parameters (i.e., partitioning allows to specify both the factor and the type), a set of values for each parameter is specified.

Example: Snippet 3.1 and Figure 3.1 show an example of specification for a target design–function `bar` in the snippet–and the associated notion of directive, knob, knob type, location and set of values associated to a directive for possible

targets in the design.

For a design D, let $\mathscr{X}_D$ denote the set of all possible synthesis configurations. In general, $\mathscr{X}_D$ is a very large set, possibly of infinite size. In practice, designers explore a portion of the design space of D by trying a subset $X_D \subset \mathscr{X}_D$, whose elements they choose carefully based on their experience running HLS.

The set of all the possible configurations explored by a designer is the *Configuration Space* (X_D). This is defined as the cartesian product among the sets of directive values for each knob:

$$X_D = K_1 \times K_2 \times \cdots \times K_N \tag{3.1}$$

where N is the number of considered knobs, and K_i is the set of values related to each i knob, i.e. the set of values that the directive associated to knob i can assume. For a directive with multiple parameters–e.g., array partitioning requires to define both the partitioning factor and the type–, K_i is the Cartesian product among each set of values. The size of the configuration space is then given by its cardinality ($|X_D|$).

Lastly, the *design space* (Y_D) is the set of implementations resulting from the synthesis of the configuration is X_D.

3.2 Problem Formulation

The DSE of an HLS design is a multi-objective optimisation problem, with costs and merit as objective functions. In literature different objective functions have been considered for the DSE problem. The survey from Bulnes et al. [97] identifies the following as the most common objectives considered in the state of the art:

- *Latency*. This is the most common unit of measure for the performance. It is usually defined as the number of clock cycles required by a hardware implementation to complete its functionality. Alternatively, the total latency is multiplied with the target clock of the system. In this case the objective is referred as *effective latency*.

- *Throughput.* Ratio among the effective latency and the input size. It measures how efficiently the input is processed with respect to its size.

- *Area*. This is the most common unit of measure for the costs. It can be either the space occupied to implement the IC in hardware, expressed in

μm^2, or the number of components allocated in the target device, i.e. functional units (FU) and registers. While the first is commonly used for ASIC, the second approach is well suited for reconfigurable resources as in the case of FPGA. I this cases common units of measure are the number of Flip-Flops (FF), Look-Up Tables (LUT), Digital Signal Processor (DSP), and Block RAM (BRAM).

- *Resource utilization*. With FPGA, due to the limited number of resources available on a target device, the costs is expressed as percentage of resources required by an implementation. With such formulation, multiple different resources can be aggregated together in a single objective function.

- *Power*. Total power consumption of the IC. This is usually the combination of dynamic and static power consumptions.

- *Wire length* or Data path. Measure of costs of the interconnection and of the connectivity components.

- *Digital noise*. Measure of error due to the combination of computational errors and noise propagation introduced by bit-width reduction and quantisation effects.

- *Reliability*. Measure of probability that an error will occur due to soft errors. This usually depends form the type of FU used in the design.

- *Temperature*. Measure of maximum temperature and temperature variation occurring during the execution of the design functionality. Usually minimised to reduce electronic failures.

- *Robustness*. Protection that a IP offers against attacks, i.e. reverse engineering attacks.

In the context of my works I have considered as measure of cost the area (a), expressed either as the number of *FF*, *LUT*, *DSP*, and *BRAM*, or as aggregated values of those in form of a linear combination of their utilisation for a given technology:

$$a = \frac{FF}{FF_{available}} + \frac{LUT}{LUT_{available}} + \frac{DSP}{DSP_{available}} + \frac{BRAM}{BRAM_{available}} \tag{3.2}$$

Equation 3.2 allows to evaluate the overall utilisation of the resources required by an implementation (*FF*, *LUT*, *DSP*, and *BRAM*) with respect to the ones available on a specific FPGA ($FF_{available}$, $LUT_{available}$, $DSP_{available}$, and $BRAM_{available}$).

As a measure of merits I have considered either latency and effective latency (l). Where the effective latency is measured as the product among the clock cycles and the clock period.

$$l = \#Clock_cycles \times Clock_period \quad (3.3)$$

The resulting area a and latency l obtained for a given configuration through the HLS process define the implementation cost and merit.

Goal of the DSE problem is to identify the Pareto-optimal implementations of a given design while minimising the number of synthesis required to implement them. Given D and X_D, the *design space exploration task* returns a subset of X_D that consists of all Pareto configurations, i.e.

$$P(D, X_D) = \{x | x \in X_D \text{ and } x \text{ is Pareto}\} \quad (3.4)$$

A *Pareto configuration* (p) of a design is a configuration that leads to an implementation that is Pareto-optimal in the bi-objective optimization space defined by the performance and cost metrics.

A (first-rank) Pareto frontier ($P(D, X_D)$, or, simply, P) is the set of points:

$$p \in P \Longleftrightarrow \nexists\ q \in X_D, q \neq p \mid A(q) \leq A(p) \wedge L(q) \leq L(p).$$

Where $A(\cdot)$ and $L(\cdot)$ are the area and latency values associated to a configuration.

In other words, iff $p \in P$, then no other solutions exist in the design space having simultaneously less area and less latency than p.

A (first-rank) Pareto frontier is the set of Pareto-optimal points. Finally, an *i-th rank Pareto frontier* (for $i > 1$) is defined as the Pareto frontier obtained after removing the lower rank frontiers from the design space.

Figure 3.2 shows examples of different ranked Pareto frontiers.

DSE strategies aim at finding an approximate Pareto frontier $\widehat{P}(D, X_D)$ of the best performing implementations, as close as possible to the one deriving from exhaustive search $P(D, X_D)$, while minimizing the number of synthesis runs.

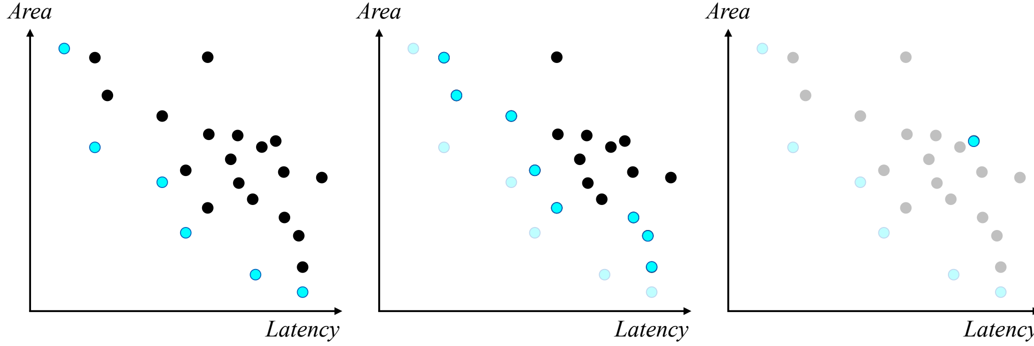


Figure 3.2. Example of three different rank Pareto frontier. (Left) 1st-rank Pareto frontier. (Center) 2nd-rank Pareto frontier. (Right) last rank Pareto frontier.

3.3 Metrics

Quality evaluation of DSE is a challenging aspect. In literature a variety of metrics have been adopted to measure the results of the DSE process. Different works [91] [22] [21] measure the improvement obtained by an optimised implementation with respect to a standard one–e.g., the one generated by the HLS tool without applying directives. For a given implementation the performance and area improvement and/or reduction are measured and compared with the ones obtained with different methodologies. While this approach offers an immediate view on the effectiveness of a given implementation it doesn't highlight the ability of a methodology to explore concurrently the design space objectives.

To deal with this problem, Zitler et al. [149] suggested the use of the following metrics:

- *Average Distance from Reference Set (ADRS)*. The ADRS metric expresses the distance between a reference curve P (the Pareto frontier from ground truth data), and an approximated curve $\widehat{P}$. The ADRS for two objective functions is defined as:

$$ADRS(\widehat{P}, P) = \left[\frac{1}{|P|} \sum_{p \in P} \min_{\widehat{p} \in \widehat{P}} (d(\widehat{p}, p)) \right] \tag{3.5}$$

  $\widehat{P}$ and P are the set of points defining the approximated Pareto frontier and the reference one, respectively. $|P|$ defines the cardinality of the reference set P and $d(\widehat{p}, p)$ is the distance among a reference point and an approximated one defined as:

$$d(\widehat{p},p) = \max\left\{0, \frac{A_{\widehat{p}} - A_p}{A_p}, \frac{L_{\widehat{p}} - L_p}{L_p}\right\} \tag{3.6}$$

Given this formulation, low ADRS values are better, because they imply proximity between P and $\widehat{P}$.

- *Hypervolume* or *S-metric* or *Lebesgue measure*. This metric is used to measure the difference among hypervolumes (HV) between the approximated Pareto curve $\widehat{P}$ and the reference Pareto-set P. The HV of a set of solutions measures the size of the portion of the objective space that is dominated by those solutions collectively [134]. It requires the definition of a bounding point to calculate the volume of design space. Given a 2D design space, the HV measures the difference in area among the region of design space comprised between the bounding point and the Pareto-front P with respect to the area of the bounding point and the approximated Pareto curve $\widehat{P}$. A low difference among HVs implies a good approximation of the Pareto frontier.

- *Pareto Dominance*. This metric measures the relation among the number Pareto-optimal design discovered $\widehat{P}$, and the one in the reference Pareto-set P. This relation is defined as:

$$Dominance = \frac{|\widehat{P} \bigcap P|}{|P|} \tag{3.7}$$

  A high dominance score implies a good exploration result.

- *Cardinality*. This metric lists the Pareto solution discovered. High cardinality implies a larger variety of solution discovered. While this metric does not require a reference set to be calculated, it does not guarantee a good exploration quality.

Among the above mentioned metrics, with the exception of the *cardinality* metric, a ground-truth is required to compute the metric score. However, retrieving the reference Pareto-set P requires the exhaustive exploration of the design space and such process can be extremely time consuming if not infeasible. This aspect has often limited the use of such metrics as measure of the quality of results.

In my works, similarly to [64],[71],[86],[114],[138],[146], I have evaluated my methodologies by adopting the ADRS metric, measuring the distance among the Pareto-frontier retrieved by a given strategies with respect to a ground-truth. This approach, while requiring the exhaustive synthesis of the configuration space

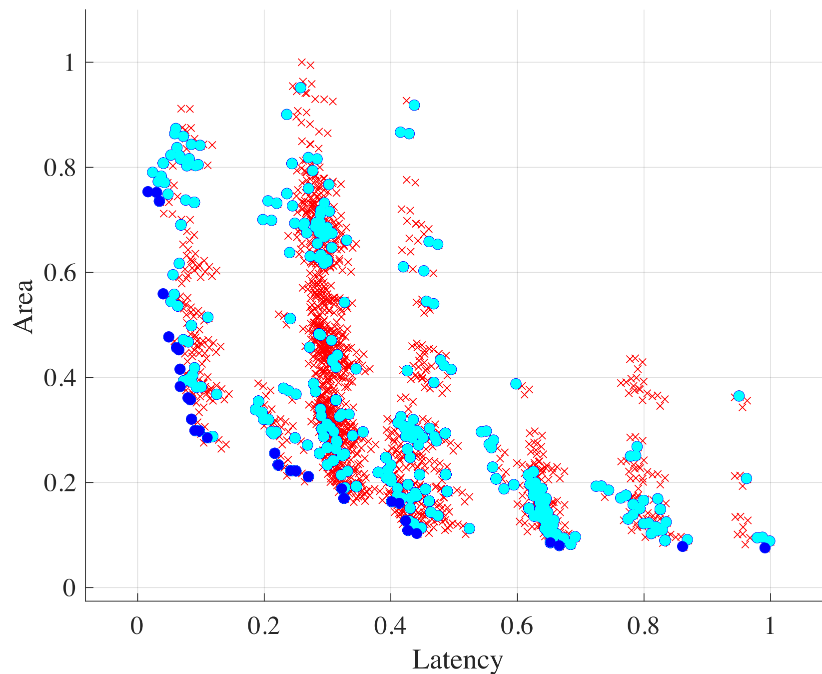


Figure 3.3. Example of an exhaustive exploration compared with an exploration heuristic. The x are the result of exhaustive exploration, while the dots (dark and light) are the design points explored by our heuristic (13% of the total). Dark dots represent the retrieved Pareto solution.

defined by the designer, allows a comparison with existing methodology which is both qualitatively with respect to the state of the art alternatives and with respect to the design space of the target design.

Figure 3.3, shows an example of DSE performed for `decode` benchmark from CHStone [47]. The synthesised configurations are marked with filled dots, non synthesised ones with crosses, and Pareto-optimal solutions of the design space with darker dots. The two objective functions considered in this DSE are the area, as a number of FF, and the latency, in number of clock cycles. An ADRS of 0.0128 was achieved with only 234 synthesis runs (over 1728 possible configuration).

Chapter 4

Related Works

A number of recent works have analysed the state of the art of HLS-driven DSE. The surveys from Schafer et al. [108], Bulnes et al. [97] and Shathanaa [112] offer an overview of the trends in the design of DSE strategies. These surveys propose their own classification of the methodologies in literature. However, none of them agree on a unique categorisation of the existing approaches.

In the survey from Schafer et al. [108] DSE techniques are classified in two categories: synthesis-based and model-based. According to the proposed classification, the synthesis-based methodologies generate a new configuration and invoke the HLS tool to evaluate the resulting implementation. Model-based ones instead avoid the invocation of the synthesis process to estimate the resulting cost and performance. The proposed classification however became shadowy when supervised learning methodology are discussed. According to the survey, these methodologies belongs to both the above mentioned categories. In fact, in order to build their knowledge, and in some cases refine it, the synthesis of many different configurations is required. Only once the methodology has learned a model of the HLS tool behaviour they use the acquired knowledge to estimate the effect of the HLS directives.

Alternatively, Bulnes et al. [97] propose a more fine-grained classification of the multi-objective methods for DSEs. According to the proposed taxonomy, two macro-categories can be identified: exact methods and approximate ones. While exact methods [80][81][79], based on variation of the branch and bound algorithm, had a scarce appealing in the scientific community, approximate ones had a large popularity among researchers and have steered the research direction. However, the survey focused only on heuristic algorithms and meta-heauristic approaches almost ignoring the portion of state of the art considering analytical methods.

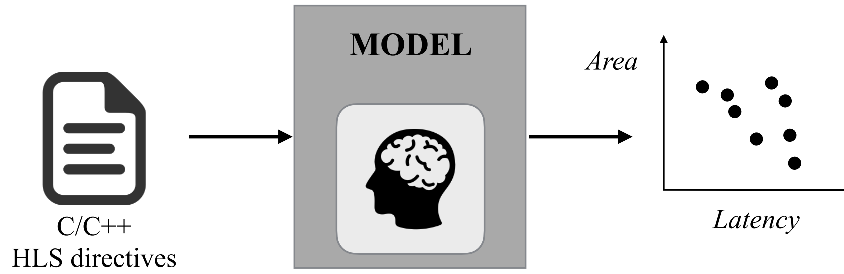


Figure 4.1. Model-based approach where the quality of the estimations relies on the knowledge embedded in a model.

Lastly, Shathanaa et al. [112] divide the state of the art into learning-based methodologies, exploration-based types, evolutionary algorithm, and population-based stochastic optimisation methods. This classification partially overlaps with the one proposed by Schaefer et al. [108], but as for the work from Bulnes et al.[97], analytical models are not discussed.

Herein, in order to discuss the state of the art I will introduce a slightly different classification inspired to the one presented by Schaefer et al. [108]. The classification I propose look at the DSE problems from a higher level of abstraction and broadly divides the literature into two categories: *model-based* methodologies and *black-box-based* methodologies. *Model-based* strategies (Figure 4.1) aim at estimating the effect of a directive given *a priori* knowledge of the HLS tool employed and of the application structure. *Black-box-based* methodologies, instead, *infer* the behaviour of the synthesis tool and the application properties. These methodologies can be further subdivided in learning-based methodologies and refinement-based ones. Approaches in the former category exploit an initial training phase to learn the HLS process behaviour and then perform the exploration (Figure 4.2); the latter strategies use an iterative refinement approach to gradually estimate the model and guide the DSE selecting, at run time, the most promising configurations (Figure 4.3).

In the following sections I will discuss the major contribution proposed in the past years in the context of DSEs. In Section 4.1, the strategies belonging to the *model-based* category are discussed, while in Section 4.2 are presented and discussed the methodologies belonging to the *black-box-based* group.

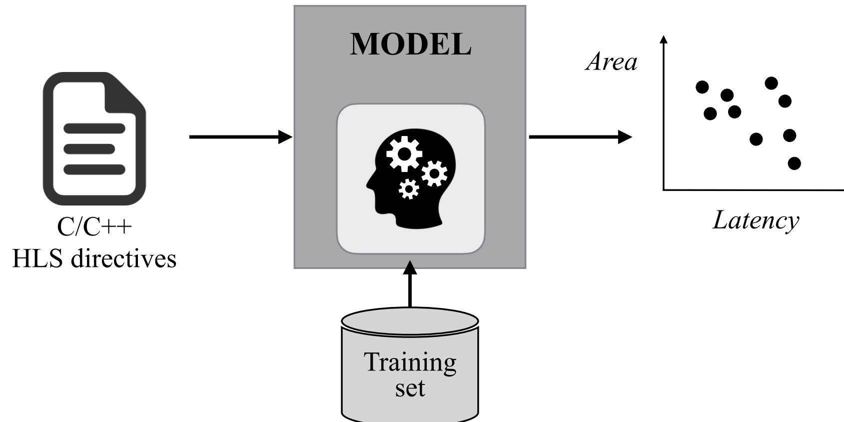


Figure 4.2. Learning-based methodologies. These methodologies rely on a training phase to infer a model and generate the estimated performance and costs.

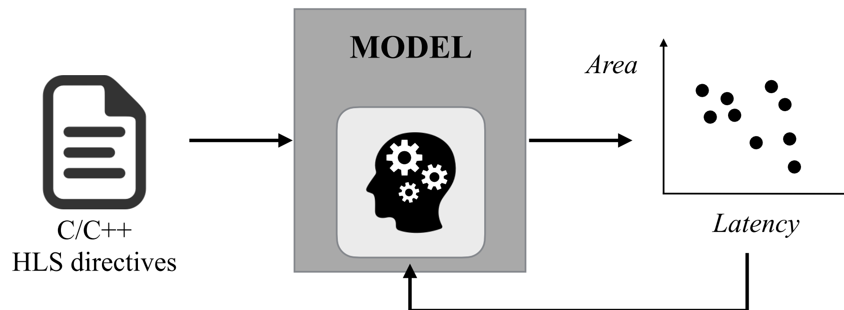


Figure 4.3. Refinement-based methodologies. These methodologies learn runtime the behaviour of the HLS tools updating their model at each new iteration.

4.1 Model-based Strategies

Model-based approaches aims at estimating the behaviour of the HLS process, for a given design, without directly invoking the synthesis process. Such techniques perform an *a priori* estimation of performance and costs of the resulting implementations. While such approaches are often accurate in estimating the implementation characteristics, these are often limited in the number of HLS directives considered. In order to pre-estimate the area, power, and latency, the designers build a model–usually an analytical one–that analyses the behavioural description of the design and predicts performance and required resources.

These approaches require the characterisation of both HLS directives and of

the specific HLS tool modelled. Such aspects, are the main drawback of *model-based* strategies. Estimating the effect of multiple directives together is an extremely complicated task due to the impact that each of them has on the allocation, scheduling, and binding operations of the synthesis process. Thus, because of this complexity, existing models are often limited to only few directives. Nonetheless, such models are tailored for specific HLS tools, causing the methodology to be constrained to it.

Example of these approaches are the works proposed by Choi et al. [16], Chi et al. [14], and Cong et al. [19], [20]. The work from Choi et al.[16] considers only array partitioning, loop unrolling and pipelining as HLS directive, and an analytical model is built in order to estimate resource and performance of the resulting implementation. Similarly, in the works from Chi et al. [14], and Cong et al. [19], [20], analytical models quantify the performance and resource consumption. In these paper, to reduce the complexity of the model, the strategies focus on specific class of applications–i.e. stencils in the work from Chi et al. [14]–, architectures–i.e. systolic arrays in the work from Cong et al. [19]–, or by proposing design templates [20] to reduce the configuration space size.

On a similar stance, Tan et al. [121] and Liu et al. [63] propose the use of architectural templates to constrain the design space and effectively lowering the complexity of the problem before building an analytical model of it.

In addition, a number of recent works from Zhong et al. [147], Wang et al. [130], and Zhao et al. [145] have proposed methodologies based on static graph analysis techniques. In these works compiler techniques are used to generate a graph representation of the design starting from the behavioural representation of it. The graph representations are used to perform analysis of the processing elements and their communication. These information are then used to build a model of the computational elements and quantify the communication cost.

From a different perspective, Shao et al. [111] have proposed Aladdin, a pre-RTL, power-performance simulator for fixed-function accelerators. The framework accurately estimates performance, power, and area of accelerators starting from their C/C++ implementation and a set of associated directives. Similarly, Choi et al. [15] have presented a HLS simulation flow able to extract scheduling information from the HLS tool and automatically construct an equivalent cycle-accurate simulation model while preserving C semantics. These approaches while offering an accurate prediction of performance and costs by avoiding the synthesis process, do not help addressing the problem related to the high dimensionality of the design space.

The works from Balivaran et al. [6], and So et al.[115] exploit early estimation techniques, based on pre-characterisations of specific directives for a

given HLS tool, to decide the configurations to synthesise during the exploration. Lastly, Haubelt et al. [48] propose a set of rules that enable designers to estimate the effect obtained by combining different design implementations by modelling the directives behaviour.

All these approaches have been shown to effectively address the exploration problem by reducing the complexity of the design space. However, these results are achieved either by constraining the size of it, or by focusing on specific class of applications. Therefore, while being able to obtain high quality results these methodologies lacks in generalisation.

Moreover, analytical models are fine tuned for specific HLS tools, binding the methodology to the tool analysed to build the model. Adapting the existing strategies to a different tool is not always a straightforward process since it requires the characterisation of both directives and behaviours of the alternative tool. In addition, commercial tools change rapidly to address the market requirements (e.g. a new version of Vivado HLS [126] is released every year, and minor updates are are released every 4 months), causing model based approaches to potentially become obsolete in a short time. Because of these aspects, model-based approaches do not guarantee the portability of the devised models both across different tools and among different releases of it.

On the other hand, *Black-box-based* strategies address these limitations by adopting an agnostic approach both to the number of directives considered, and to the HLS tool chosen by the designer. The next section offers an overview of the state of the art methodologies belonging to this group.

4.2 Black-box-based Strategies

Differently from the *model-based* approaches discussed in Section 4.1, *black-box-based* strategies do not assume an initial knowledge of the problem. These approaches neither assume prior knowledge of the HLS tools or a characterisation of the HLS directives considered. As a drawback, these approaches need to build their knowledge either offline or online during the exploration. This step has a cost–not present in the *model-based* strategies–which often requires the *black-box-based* methodologies to perform a higher number of synthesis to retrieve high quality results.

Black-box-based methodologies, can be divided in two different groups. On one hand, there are *learning-based* approaches which rely on an initial phase to learn a model of the synthesis process and, according to the gained knowledge, guide the selection of the configurations to be explored. These strategies usually

build their knowledge in a one time process, then, they use such knowledge to estimate performance and costs or to guide the exploration. On the other hand, *refinement-based* approaches may rely or not on an initial training phase to build the initial knowledge base, but subsequently, during the ongoing exploration, such knowledge is refined with the newly acquired data.

Given this classification, different strategies have been proposed. According to the taxonomy from [108] and [97] surveys, we can identify three main group of strategies: meta-heuristics, dedicated heuristics and supervised learning approaches. While meta-heuristics and dedicated heuristics fall under the *refinement-based* umbrella, supervised learning approaches can be either in the the *learning-based* group or in the *refinement-based* one, depending on whether the model is updated or not during the exploration.

4.2.1 Learning-based strategies

Ozisikyilmaz et al. [85] proposed an approach using statistical inference to create a predictive model relying on neural network and linear regression to guide the DSE of a computer architecture. The proposed approach predicts performance and cost given processor, memory, and bus parameters of a new architecture. While this approach highlighted the possibility to use neural network to guide DSE, to the best of my knowledge, there are no works in literature that adopted such an approach for HLS-driven DSE.

Alternative approaches have relied on different learning models in order to retrieve good quality results with a low budget of synthesis. To this end, the work from Zuluaga [150] proposes a regression model which relies on Gaussian process to predict area and latency given an initial training set. Schafer et al. use learning-based methods to implement local search techniques, using pattern matching techniques [106]. Meng et al. [76] proposed a machine learning approach using adaptive threshold non-Pareto elimination which, instead of focusing on improving the conventional accuracy of the learner, focuses on understanding and estimating the risk of losing good designs due to learning inaccuracy. Zacharopoulos et al. [141] combines a compiler pass analysis and Random Forest classifier to predict the optimal unrolling factor of loops.

All of the above mentioned methodologies rely on an initial phase to learn a model of the synthesis process and, according to the gained knowledge, they perform the selection of the configurations to explore.

4.2.2 Refinement-based strategies

These methodologies, differently from the learning-based ones, are able to refine their knowledge during exploration. In this way, the search for Pareto solutions can focus on promising regions of the design space discovered online, excluding the sub-optimal ones by adapting the internal model. During the search for Pareto optimal solutions, feedbacks resulting from synthesis are collected and are used to refine the model, aiming at expanding its knowledge while identifying the most promising configurations.

To this end, meta-heuristic approaches and dedicated heuristics have been shown to obtain good results when dealing with multi-objective optimisation problems. In particular population based heuristics, and response surface models had large popularity in the research community.

Schafer et al. use an adaptive simulated annealing approach [105] to generates a series of designs exploring efficiently the design space. The same author proposed a probabilistic model to predict Pareto-dominant solution according to the choice of the HLS pragmas [104], and a divide and conquer approach which first explores the loops separately and at the end merges the exploration results [107].

Liu et al. [64] uses the Random Forest algorithm to infer a model of the HLS tool and refine the model at each new synthesis. Similarly, Fornaciari et al. [40] propose a sensitivity-based methodology which evaluates the influence of architectural parameters, selecting and guiding the exploration according to the most influential ones.

Different works explored population based techniques [1], [87] and [51] investigating the use of Genetic Algorithms to guide the DSE problem, addressing different subsets of directives. Other works ([138], [71], [113] and [86]) use a Response Surface Model to refine the simulation-based exploration and generate a model of the synthesis engine.

With a different approaches, Beltrame [5] has instead developed a methodology which uses the domain knowledge derived from the platform architecture to guide the exploration using a Markov decision process. Wu et al. [135] instead, presented a strategy that considers datapath and dynamic FU allocation to explore area/power trade-off. Piccolboni et al. [91] instead designed an automatic methodology for the design-space exploration that concurrently coordinates both HLS directives, and memory optimization.

In a recent editor's note, Doppa et al. [31] highlighted the importance of leveraging prior knowledge to effectively reduce the complexity of DSE problems. A small number of works take this stance in the context of hardware design.

However, they do so from a different and somehow limited perspective.

To this end, Dai et al. aim at improving the accuracy of HLS estimations using post-synthesis data [25], while the goal of Liu et al. is to estimate the performance on FPGA from an ASIC synthesis report [67]. Deshwa et al. leverage prior knowledge in the context of network-on-chip DSEs, to identify promising starting point for their exploration methodology [29]. More recently, Wang et al. [132] proposed a method to accelerate the process of HLS-driven DSE by pre-characterizing micro-kernels offline and creating predictive models of these.

Finally, Martins et al. [73] also present a strategy to harness prior knowledge based on a similarity metric, but their framework is geared towards the selection of *compiler* optimizations, as opposed to targeting the hardware domain of HLS.

My research focused mainly on the identification of DSE strategies categorised under the *black-box-based* class. In particular, works detailed in Chapter 5 belong to the *refinement-based* category while the one described in Chapter 6 falls in the *learning* one. They differ from the literature for the introduction of new formulations able to outperform state of the art methodologies. Moreover, to the best of my knowledge, the *learning-based* methodology detailed in Chapter 6 is the first attempt to apply transfer learning to DSE problems for automatic optimisation of hardware accelerators.

Chapter 5

Refinement-Based Strategies

In this chapter two DSE methodologies, that I have devised during the Ph.D., are discussed. Both the methodologies presented here belongs to the *refinement-based* categorization introduced in Chapter 4. *Refinement-based strategies* aim at learning the behaviour of an HLS tool according to a black-box approach. By observing the results of the HLS process for a given set of inputs, a *refinement-based* strategy aims at estimating the resulting performance and costs of the HLS process given the current knowledge of the problem. Each time a new implementation is generated, the new acquired knowledge is used to adapt the inference process and select a new configuration to synthesise, aiming at improving the Pareto-frontier.

Section 5.1 presents a cluster-based approach that identifies patterns among similar implementations in the design space and decomposes the DSE problem in many smaller subproblems, effectively lowering its complexity and addressing each subproblem independently. Section 5.2 describes a local-search based strategy that reshapes the DSE problems into a lattice and navigates it based on the observation that Pareto-implementation are neighbours in the lattice representation of the design space.

5.1 Cluster-Based Heuristic

By analysing the result of a DSE, the presence of certain patterns among the data can be observed. Figure 5.1 shows an example of DSE for the *ChenIDCt* function, from the CHStone benchmark suite [47]. The synthesised implementations are marked with red crosses, while Pareto-optimal solutions of the design space are marked with darker dots. From the figure, can be easily identified groups of points clustered together and comma-shaped patterns. However, while these

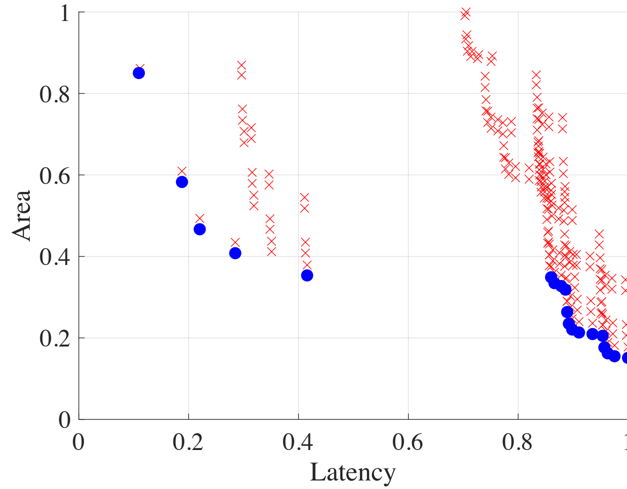


Figure 5.1. DSE of `ChenIDCt` from CHStone [47] in area-latency space.

patterns may be easily identifiable in simple design spaces, they may not be easy to spot in more complex ones.

The exploration strategy presented in this section is motivated by the observation that some combinations of values, when assigned to directives, result in high-quality implementations, while others are sub-optimal, leading to designs with high cost and low performance. For example, the unrolling factors of two different loops may be inter-dependent because of a producer-consumer relation. Configurations which do not abide to this relation likely result in sub-optimal designs. Starting from a small initial set of area/latency points in a design space, I therefore explore it by clustering solutions with a high degree of similarity, and discarding clusters which are distant from the Pareto curve. Pareto-optimal implementations of the clusters are then combined to generate new configurations estimated to improve the Pareto frontier.

Clustering allows to decompose the DSE problem in many smaller subproblems, effectively lowering its complexity. Solutions are clustered considering their similarity both according to the synthesis output and to the input directive values.

An implementation of a design D is defined as a vector in the space of area (A), latency (L), and configuration space. The space defined by the area and latency is named S while the space of all the possible configuration values is named X_D (or, simply, X). Each element of X is a configuration vector $\vec{x}$. A configuration vector is created by the concatenation of the knob values defining a configuration. The set of all the configuration vectors defines the configuration

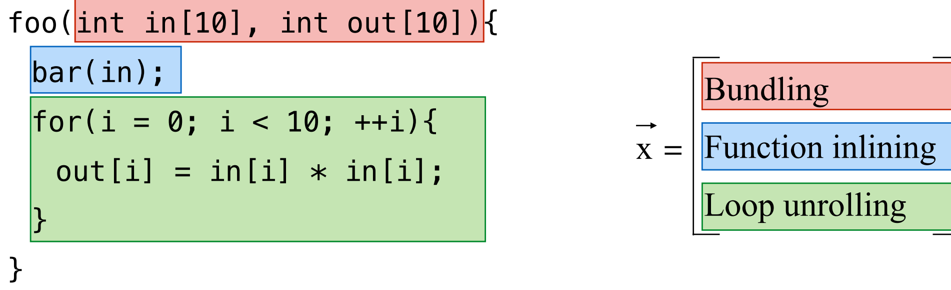


Figure 5.2. Example of a C code and the regions that may be addressed with HLS directives.

space. Each component of the configuration vector belongs to the set of directive values associated to a specific knob. Given N knobs, a configuration vector $\vec{x}$ is defined as:

$$\vec{x} = [k_1, k_2, \cdots, k_m], \quad k_1 \in K_1, k_2 \in K_2, \ldots k_n \in K_m \tag{5.1}$$

with K_i being the set of directive values associated to the knob i. To obtain an equally distributed representation of the directive values in X, we represent these in a discretised form in which each element k_i has the same distance from its previous and following elements in K_i with $K_i \in [0, 1]$.

Example. Figure 5.2 shows a simple HLS code snippet and three possible HLS directives that can be applied: function inlining, input bundling and loop unrolling. Inlining can either be performed or not, so the inline directive has two legal values. Similarly, the two arrays can either be bundled on the same port or each assigned to a different one. Finally, for this example we assume that the legal loop unrolling factors are $[1, 2, 5, 10]$, corresponding to the discretized values of $[0, 1/3, 2/3, 1]$. A possible discretized configuration vector for this application is $\vec{k} = [0, 1, 1/3]$, which corresponds to an implementation with no port bundling, inlining of function `bar`, and the for loop unrolled by 2.

The retrieved area (a) and latency (l) numbers obtained synthesizing a configuration, defined by a vector $\vec{x}$ from X, are concatenated to generate a vector $\vec{s}$ belonging to S as follows:

$$\vec{s} = [a, l], \quad a \in A, l \in L \tag{5.2}$$

Area and latency values are normalized in order to represent them with the same range of values of the directive sets. In this case, the normalization is derived by dividing each element in S by the maximum area and latency values among the explored configurations.

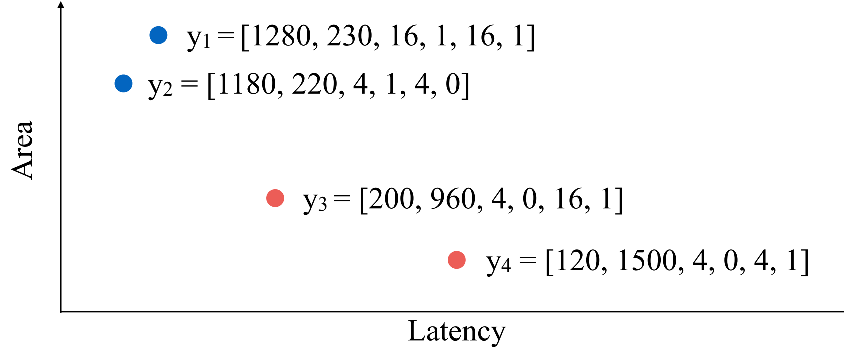


Figure 5.3. Explored points in the design space are characterized by their area, latency (first two values) and directives (last four) parameters. In the example, y_1 and y_2 may be grouped in a single cluster because they have a similar performance. y_3 and y_4 are also grouped together, because they adopt similar directives values.

An implementation in the design space is completely characterized by its values in S and X. The concatenation of these two spaces is named the clustering space Y, and its elements $\vec{y}$ are defined as:

$$\vec{y} \in Y = A \times L \times K_1 \times K_2 \times \cdots \times K_m \tag{5.3}$$

$$\vec{y} = [a, l, k_1, k_2, \cdots, k_m] \tag{5.4}$$

Two points (e.g. y_3 and y_4 in Figure 5.3) may be assigned to the same cluster if they present similar design parameters, even if they have quite different area and delay. The grouping of points into clusters is performed each time a new solution is synthesized, letting the correlation between values of different directives to naturally emerge. Therefore, the proposed strategy waives the need for a model of the effect of each directive on a target design.

An approach only relying on intra-cluster exploration may never reach DSE regions which do not include any point in the initial set. To avoid this pitfall, clusters are combined to generate new ones. This inter-cluster step enables the exploration of points whose characteristics are in-between two previously considered design space regions. Since it searches for intermediate solutions, such strategy is most effective when points with extreme directive values (high or low) are included in the DSE as part of the initial set. To ensure this condition, the directive values in the initial sampling set are generated according to a U-shaped probabilistic distribution. A detailed description of the DSE framework steps is provided in the following section.

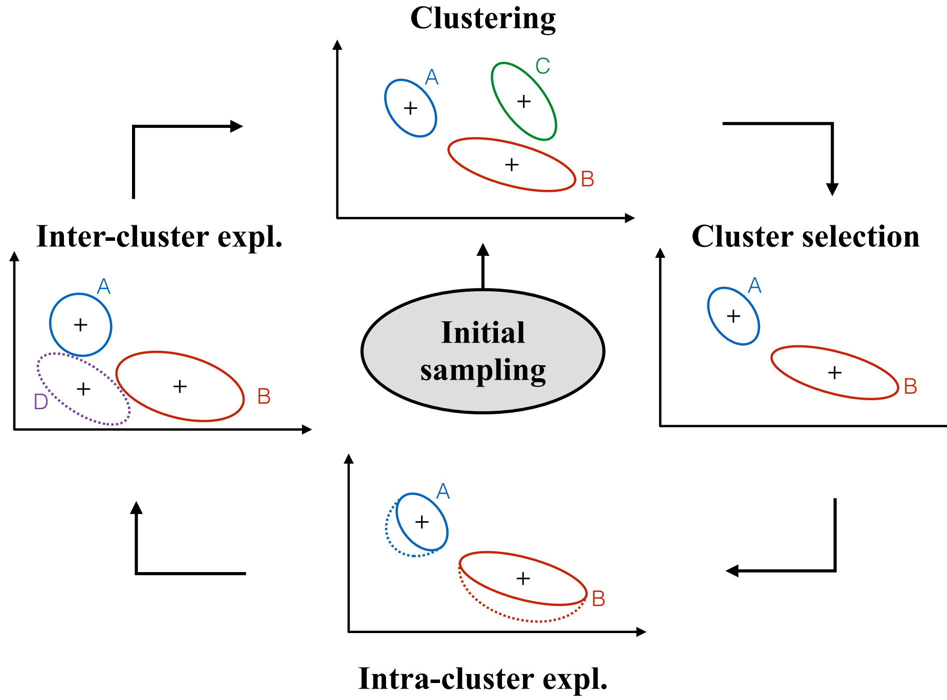


Figure 5.4. Overview of the Clustering-Based DSE framework.

5.1.1 Exploration Methodology

Figure 5.4 summarises the clustering-based heuristic. The clustering-based exploration process is divided in five steps: a) *Initial sampling*, b) *Clustering*, c) *Cluster selection*, d) *Intra-cluster exploration*, e) *Inter-cluster exploration*. Starting from the initial sampling of the design space, the Clustering, Cluster selection, Intra-cluster exploration (expansion of each cluster) and Inter-cluster exploration steps (generation of new clusters) are iteratively performed until either no new solution is found, or a user-defined budget of synthesis runs expires.

Initial sampling

This step generates the initial set of configurations $\bar{X}$, and derives the first approximation of the design space Pareto curve $\bar{P}$. For an initial sampling size n, the $\bar{X}$ space is composed of n unique configuration vectors $\vec{x}$.

The elements of each $\vec{x}$ originate from the probabilistic sampling of a symmetric Beta distribution, which is characterized by a density function, defined

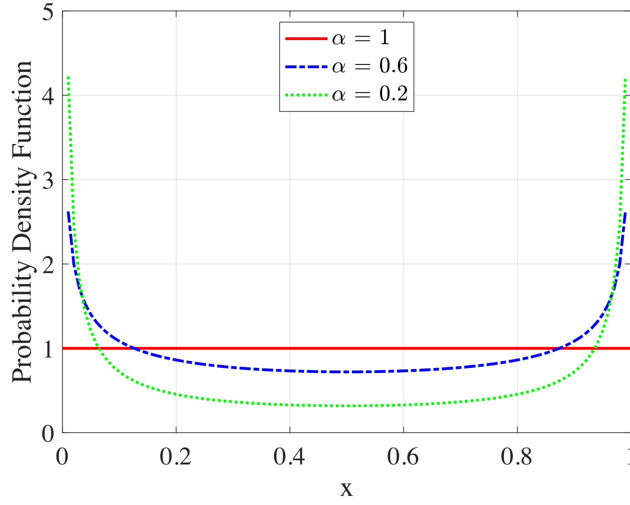


Figure 5.5. Beta distributions for different values of α.

over an interval $0 \leq x \leq 1$, as:

$$f(x) = \frac{x^{\alpha-1}(1-x)^{\alpha-1}}{\mathrm{B}(\alpha)}$$

Where B is the Beta function:

$$\mathrm{B}(\alpha) = \int_0^1 x^{\alpha-1}(1-x)^{\alpha-1} dx$$

As shown in Figure 5.5, this U-shaped distribution has, with a value of α lower than 1, a higher probability associated to the boundary values of x. By adopting it, the initial explored set of directive values $\bar{X}$ will contain elements whose values have, with high probability, extreme values from the respective set of directive values. This is a desired property in the proposed framework, so that Pareto solutions with in-between directive values will be explored during the refinement steps.

The set of area and latency vectors, resulting from the synthesis of the initially sampled configurations, defines first instance of $\bar{S}$. Each vector of $\bar{S}$ and the corresponding element of $\bar{X}$ are concatenated to generate the set $\bar{Y}$.

Clustering

Once the initial set of $\bar{Y}$ is generated, its elements (y) having common characteristics are aggregated into clusters. To this end, I relied on the Hierarchical

Clustering algorithm [133] [75]. As a similarity metric, I considered the squared Euclidean distance among the points in the $\bar{Y}$ space.

To ensure a good balance between intra- and inter-cluster exploration, multiple clusters should be present, while most of them should aggregate multiple points. This trade-off is governed via a clustering factor, which sets the number of clusters to a percentage of the number of the explored designed points. An exploration of different settings for this parameter is presented in Section 5.1.2.

The clustering operation partitions the $\bar{Y}$ space into multiple clusters C_i, and defines $\bar{C}$ as:

$$\bar{C} = \bigcup C_i \tag{5.5}$$

Each cluster is characterised by its *centroid* $\vec{c}$, which is, for each $(a, l, k_1, k_2, ...)$ component, the average value among the points belonging to the clusters. Moreover, clusters possess a *boundary* in the S space, corresponding to a 4 elements tuple containing the maximum and minimum values of area and latency of the cluster points.

Example: a cluster $C_1 = \{\vec{y}_1, \vec{y}_2, \vec{y}_3\}$ with $\vec{y}_1 = [0.1, 0.8, 0.1, 0.9, 1]$, $\vec{y}_2 = [0.15, 0.7, 0.2, 0.8, 0]$ and $\vec{y}_3 = [0.5, 0.5, 0.4, 0.4, 1]$ has the boundary: $b_{c_1} = (0.1, 0.5, 0.5, 0.8)$.

Cluster selection

This step selects which clusters will be considered for the generation of new points in the Synthesis space.

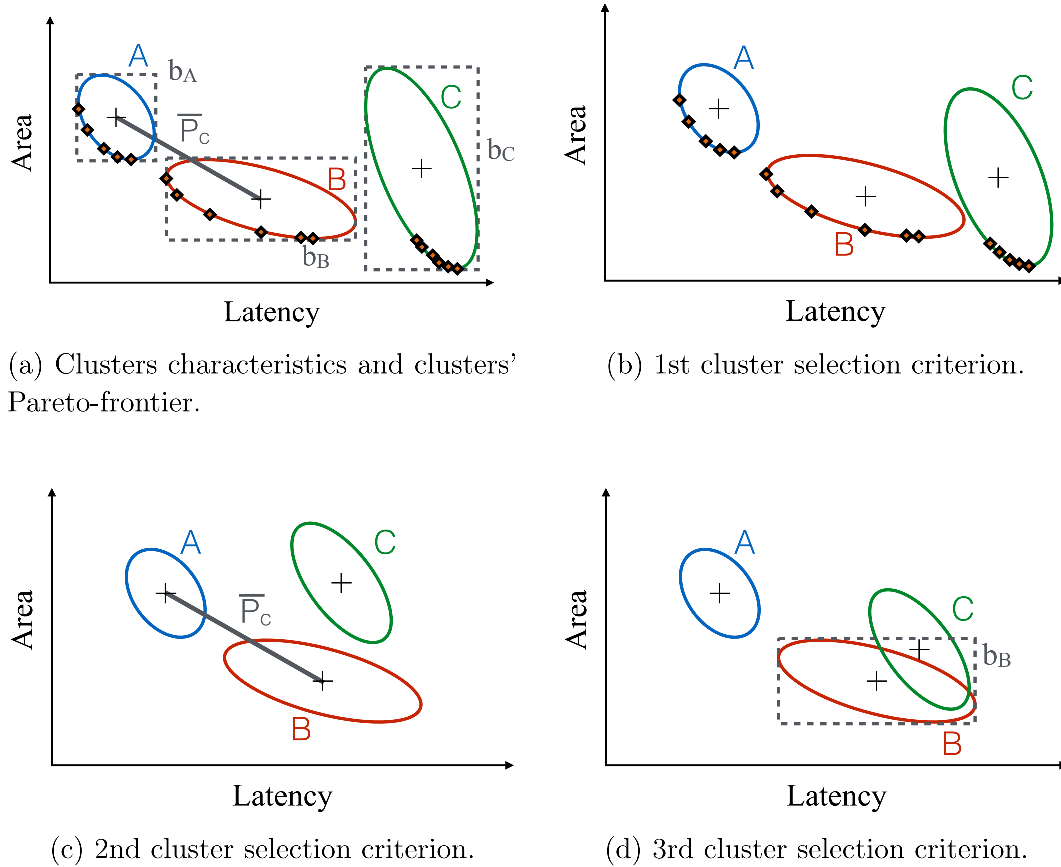
To perform it, as shown in Figure 5.6a, I consider: the Pareto frontier of the explored design space $\bar{P}$ (pictured as diamonds), the Pareto frontier of the centroids of the clusters $\bar{P}_C$ (pictured as a line among centroids) and the cluster boundaries (pictured as dashed rectangles). This data is employed to select candidate clusters corresponding to design space regions which contain promising solutions. Only the points belonging to these clusters are considered in the following intra- and inter-cluster exploration steps, while the rest are discarded.

Candidate clusters are selected according to three criteria:

1. if $\vec{y} \in C_i \ \wedge \ \vec{y} \in \bar{P}$, then C_i is a candidate cluster.

2. if a cluster C_i belongs to the Pareto frontier of centroids $\bar{P}_C$, then C_i is a candidate cluster.

3. for each $\vec{c} \in \bar{C} \backslash \bar{P}_C$, if $A(\vec{c})$ and $L(\vec{c})$ are inside the boundaries of an element of $\bar{P}_C$, then C_i is a candidate cluster.



(a) Clusters characteristics and clusters' Pareto-frontier.

(b) 1st cluster selection criterion.

(c) 2nd cluster selection criterion.

(d) 3rd cluster selection criterion.

Figure 5.6. Cluster characterization and application of the cluster selection criteria described in Section 5.1.1. (a) shows the elements which characterize a cluster and the design space. In (b) and (d) all clusters are selected for further exploration, while in (c) cluster C is pruned.

Example: Figure 5.6 exemplifies the application of the cluster selection criteria. In Figure 5.6b, A, B and C are all selected because each of them contains elements which belong to $\bar{P}$. Figure 5.6c shows an example where the application of the third criterion leads to the selection of clusters A and B. Finally, Figure 5.6d shows an example where the third criterion is applied, since the centroid of cluster C is within the boundary of cluster B.

Intra-cluster exploration

Intra-cluster exploration identifies new solutions by examining candidate clusters individually. For each cluster, the algorithm considers the points belonging to its

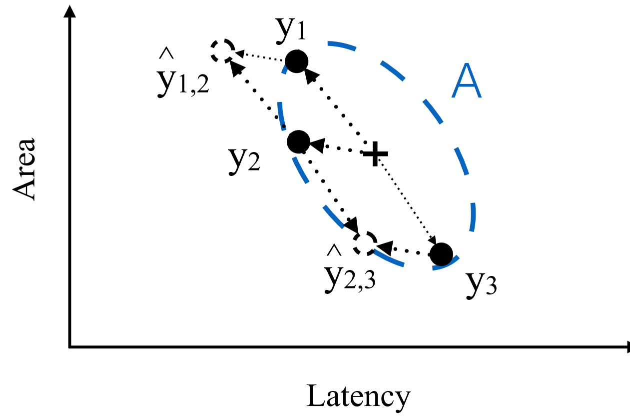


Figure 5.7. Example of intra-cluster exploration, where the estimated solutions (dashed empty dots) are generated combining Pareto-optimal ones (dark filled dots).

local Pareto frontier P_{C_i}. These points are pair-wise combined in the Y space by performing a vector addition relative to the cluster centroid, generating new *estimated* solutions $\widehat{y}$ (Figure 5.7). Combinations which do not improve the global Pareto frontier are discarded without performing a synthesis run.

Estimated $\widehat{y}$ elements may have directive values which do not correspond to valid settings for the HLS directives (e.g.: they are not integer numbers). Up to three valid configurations are therefore derived according to the following rules:

1. each component is casted to the closest directive value.

2. all components are upcasted to a valid directive value.

3. all components are downcasted to a valid value.

After this pass, estimated points which have already been synthesized are also discarded. The resulting set of $\widehat{x}$ configurations are then used to invoke the HLS synthesis tool to retrieve the corresponding (non-estimated) area and latency. Finally, the new obtained y are added to the $\bar{Y}$ space and an updated Pareto frontier is retrieved.

Example: Figure 5.7 shows an example of intra-cluster exploration. The dark filled dots are the points of the cluster before the exploration, while the dashed empty dots are the estimated Pareto configurations. In this example only the combination of $y_1 + y_2$ and $y_2 + y_3$ generate new estimated design points, since $y_1 + y_3$ does not improve the current Pareto curve.

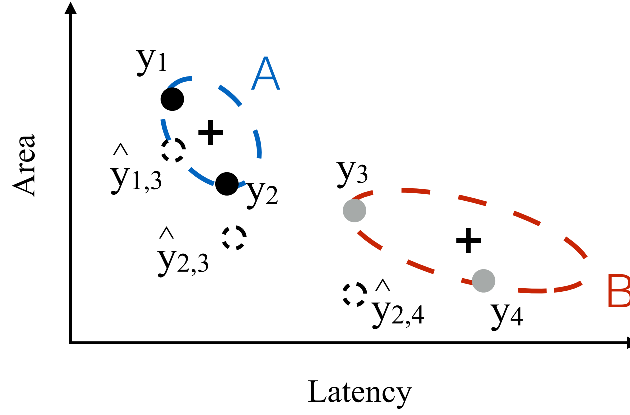


Figure 5.8. Example of inter-cluster exploration. The Pareto points of cluster A (dark filled dots) and B (light filled dots) are combined generating new candidates solutions (dashed empty dots).

Inter-cluster exploration

In order to discover unexplored regions of the design space, this stage combines design points belonging to different clusters. This step considers all pairings of the clusters in $\widehat{P}_C$, merging them and calculating their common centroid.

Vector addition, relative to the centroid, is applied to the Pareto points of the merged cluster. The resulting set of estimated $\widehat{y}$ vectors are casted to valid configuration as in the intra-clustering stage. Finally, configurations which are not yet part of $\widehat{Y}$ are synthesized, and the Pareto frontier is updated.

Example: Figure 5.8 shows an example of an inter-cluster exploration. The Pareto points of cluster A, y_1 and y_2 (dark filled dots), are pairwise summed with the ones of cluster B, y_3 and y_4 (light filled dots). The results of their vector sum generate three estimated design points $\widehat{y}_{1,3}$, $\widehat{y}_{2,3}$ and $\widehat{y}_{2,4}$ (dashed empty dots). The vector sum between y_1 and y_4 is estimated not to improve the Pareto frontier and is therefore discarded.

5.1.2 Results

Herein the implementation details and the results obtained by the proposed cluster-based heuristic are discussed. First we describe implementational details and the benchmark considered. Then, the exploration of the heuristic's hyper-parameters is discussed, and, lastly, a comparison with the state of the art alternative is shown.

Experimental setup

The HLS exploration framework presented in the previous sections has been implemented in Matlab, which, as part of its Statistics and Machine Learning Toolbox [75], provides an implementation of the Hierarchical-Clustering algorithm employed in the clustering stage. The implementations have been generated using VivadoHLS from Xilinx [126] as a high-level synthesis tool, using the Kintex7 FPGA as the target architecture, and a clock constraint of 10ns.

HLS benchmarks are derived from the CHStone suite [47]. Table 5.1 reports them, as well as the directives employed in each case and their considered values, derived by manual inspection of the benchmarks. The table also indicates the configuration space size.

To assess the performance of the proposed methodology, I have compared the result with the heuristic proposed by Liu et al. [64], which has been re-implemented. It is based on the Random Forest (RF) algorithm [9], which refines an initial design space sampled with the Transductive Experimental Design (TED [140]) method. Such combination has been shown in [64] to outperform other strategies based on different machine learning algorithms [150], [138], [86] and [71]. As a further baseline, I have also considered the cluster-based implementation and that of Liu et al. when a random initial sampling is adopted.

For all experiments, the Pareto frontier P is derived from an exhaustive search of all possible directive configurations. Such brute-force exploration required multiple days of computation on each of the considered benchmarks, highlighting the importance of exploration heuristics targeting HLS. Since both the proposed methodology and that of Liu et al. have a probabilistic component governed by a seed value, for each experimental setting I run the algorithms 100 times, averaging the results.

As a quality metric, I employed the ADRS to measures the difference between two Pareto curves: the ground-truth one P and the approximate one retrieved by the heuristic $\bar{P}$. A low value of ADRS indicates that $\bar{P}$ well approximates P, while a high one reports a low-quality approximation.

Parameter tuning

In this section are evaluated the effect of varying the parameters required by the cluster-based framework: the clustering factor (governing the size and number of clusters generated at run-time) and the number of points evaluated in the initial sampling phase.

Figure 5.9 shows the impact of adopting different clustering factors, for the

Table 5.1. Pragmas applied to the different CHStone function explored. ($|D|$ = size of the design space)

Benchmark	Function	Pragma	Values
jpeg	ChenIDct $\|D\| = 224$	unroll loop1	0,2,4,8
		unroll loop2	0,2,4,8
		unroll loop3	0,2,4,8,16,32,64
		bundle	0,1
adpcm	encode $\|D\| = 640$	unroll loop1	0,5,10,25,50
		unroll loop2	0,2,5,10
		unroll loop3	0,2,11,22
		unroll loop4	0,6
		inline all	0,1
		bundle	0,1
	decode $\|D\| = 640$	unroll loop1	0,5,10,25,50
		unroll loop2	0,2,5,10
		unroll loop3	0,2,5,10
		unroll loop4	0,6
		inline all	0,1
		bundle	0,1
gsm	Autocorr $\|D\| = 1728$	unroll loop1	0,4,16,40,80,160
		unroll loop2	0,4,16,40,80,160
		unroll loop3	0,9
		unroll loop4	0,4,19,38,76,152
		unroll loop5	0,9
		bundle	0,1
	Reflection $\|X\| = 4608$	unroll loop1	0,3,9
		unroll loop2	0,7
		unroll loop3	0,3,9
		unroll loop4	0,2,4,8
		inline func1	0,1
		inline func2	0,1
		inline func3	0,1
		inline func4	0,1
		inline func5	0,1
		bundle	0,1

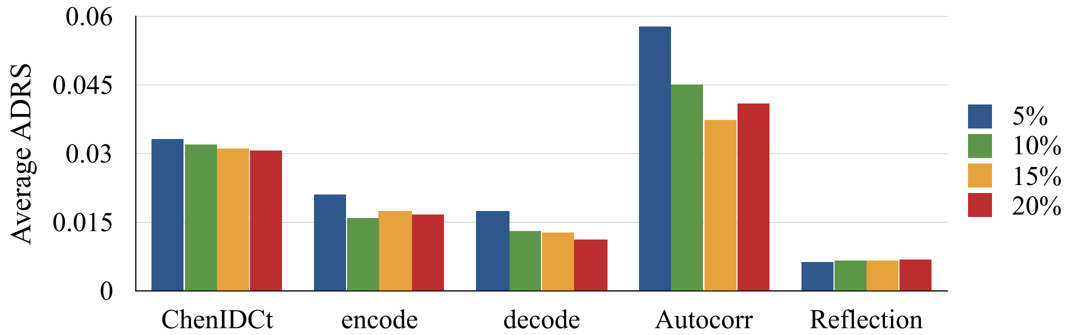


Figure 5.9. Effect of the clustering tradeoff factor for the different benchmarks, with a size of the Initial sampling set equal to 10% (lower values are better).

considered benchmarks, on the achieved ADRS. The data reported in the figure considers a number of clusters equal to 5%, 10%, 15% and 20% of the number of explored design points. It highlights that this parameter has a small impact on the quality of the results, with a value of 15% leading, on average, to marginally better results.

The size of the initial sampled set plays instead a more important role, and this is seen in Figure 5.10. The figure shows the performance of the proposed heuristic, in terms of mean ADRS achieved, for different initial sampling sizes (5%, 10%, 15% and 20% of the design space) for the `decode` function from the `gsm` benchmark. It can be observed that, the higher the initial sampling size, the better approximation the algorithm finally converges to–i.e., it converges to a lower ADRS value. On the other hand, if we consider an a-priori limited number of synthesis, then lower initial sampling sizes can outperform higher values. For example, for a budget of 100 synthesis, an initial sampling of 10% reaches a lower ADRS than an initial sampling of 15%. Indeed, if the number of synthesis is limited, there is a tradeoff between how many synthesis should be spent initially, and how many are then left for exploration. The results shown in the state of the art comparison are collected adopting an initial sampling size of 10% and a clustering factor of 15%. Consistent results have been obtained in further experiments with all initial sampling sizes and clustering factors. Given these considerations, a clustering factor of 15% and an initial sampling size of 10% have been chosen the following experiments.

State of the art comparisons

Herein is discussed a comparative evaluation of the proposed methodology with respect to the one proposed by Liu et al. of [64]. The comparison is illustrated

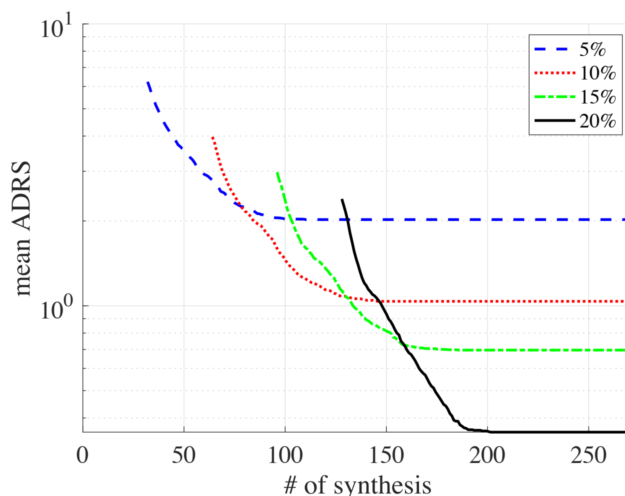


Figure 5.10. Comparison among different initial sampling sizes for the `decode` function from CHStone [47].

in Figures 5.11 through 5.15. These figures report the ADRS achieved by the proposed framework (Clust-Beta), with a maximum budget of synthesis equal to 40% of the total design space and an initial sampling budget equal to 10%. Results are compared with five other combinations of initial sampling and refinement exploration strategies: the intra- and inter-cluster exploration combined with random or TED initial sampling, and Random Forest (RF) exploration of a Beta, random or TED initial sampling. Across all benchmarks, Clust-Beta consistently outperforms alternative methodologies, both when a low or a high number of synthesis are considered, with the only exception of `Autocorr` for a high synthesis budget. The *competitive advantage* of Clust-Beta qualitatively lays in our design space decomposition, together with the intra- and inter-cluster exploration and the use of a Beta-distribution for the initial sampling. The combination of these factors enables to focus the exploration only on the most promising regions of the design space. Further experiments with *all* initial sampling sizes reported in Figure 5.10, sweeping the synthesis budget up until both our method and the best performing alternative converge or reach 40% of the design space size. Results are consistent with the one shown in Figures 5.11—5.15: Clust-Beta outperforms the other considered methodologies most (87%) of the times, resulting in smaller ADRS.

Lastly, I have evaluated the algorithm run-time (without considering the time required for the synthesis) of Clust-Beta with respect to RF-TED. For the run-time comparison I have considered the execution time of Clust-Beta and RF-TED run-

Table 5.2. Run-time comparisons: Clust-Beta vs RF-TED

	ChenIDCt	encode	decode	Autocorr	Reflection
Run-time comparisons	1.9x	1.76x	1.81x	1.31x	20.17x

ning both algorithm until no new synthesizable configuration are generated, or 40% of the design space is explored. The run-time is then divided, in both cases, by the number of synthesis effectively run, so that a measure of the time spent in the exploration engine itself can be obtained. Results are reported in Table 5.2. Note that, besides reaching a better approximation of the Pareto curves, Clust-Beta is also quicker compared to the one proposed by Liu et al. of [64]. The highest speedup is achieved in the `Reflection` case, which is the largest among the considered applications, hinting at a better scalability of the devised methodology.

This work has been presented at the IEEE International Conference in Computer Design (ICCD) 2017, and the paper has been selected for a journal publication in IEEE Transaction on Emerging Topics in Computing (TETC) 2018 [36].

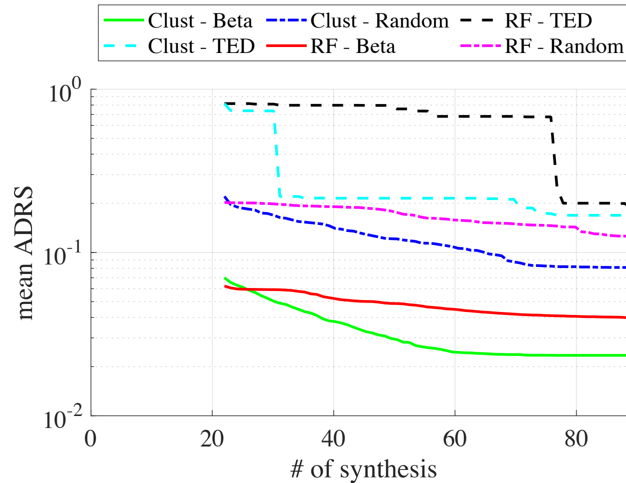


Figure 5.11. ADRS curves comparisons for the `ChenIDCt` benchmark.

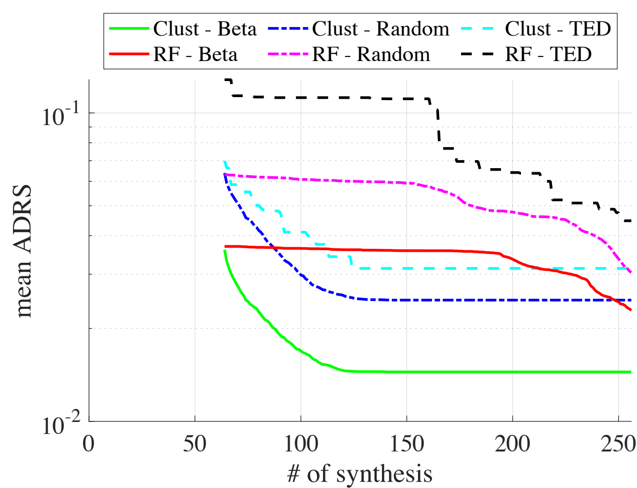


Figure 5.12. ADRS curves comparisons for the `Encode` benchmark.

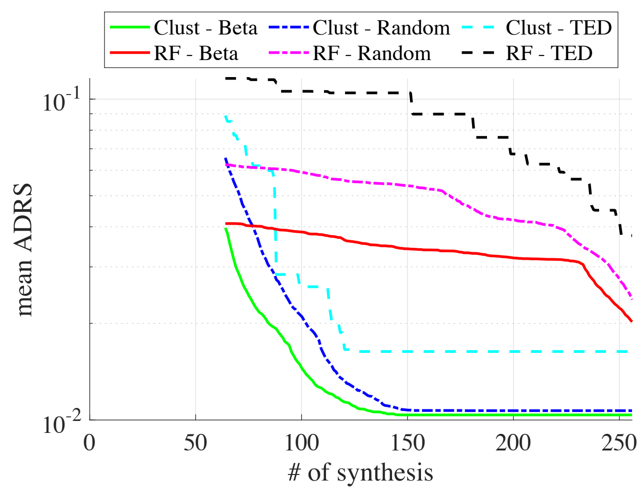


Figure 5.13. ADRS curves comparisons for the `Decode` benchmark.

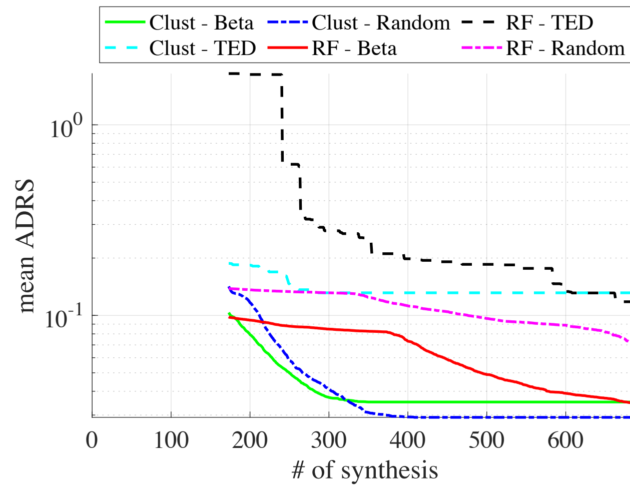


Figure 5.14. ADRS curves comparisons for the `Autocorr` benchmark.

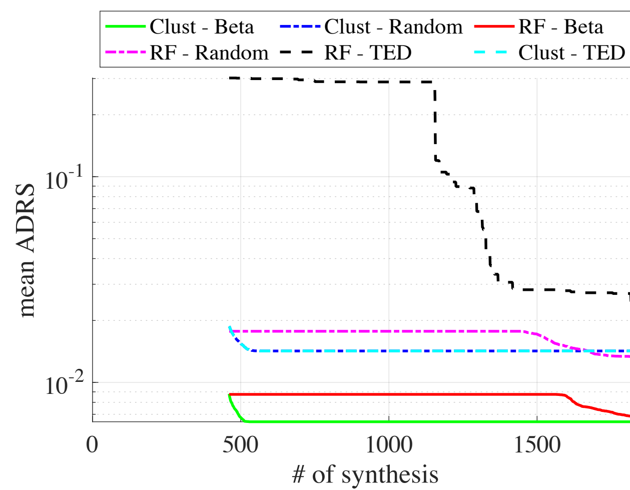


Figure 5.15. ADRS curves comparisons for the `Reflection` benchmark.

5.2 Lattice Search

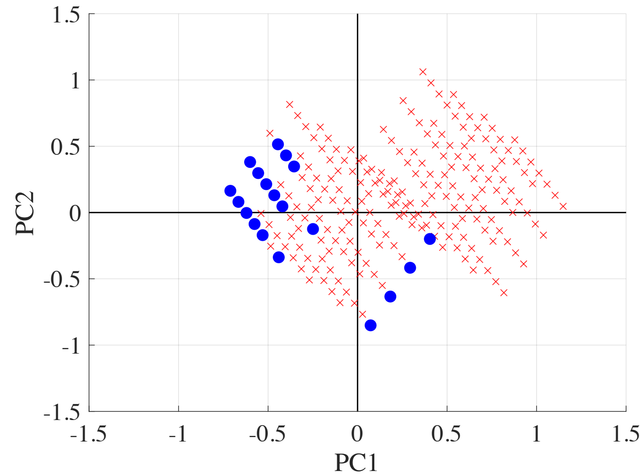


Figure 5.16. DSE of ChenIDCt CHStone [47] in the PCA directives space.

Figure 5.1 shown an example of non-exhaustive exploration of the *ChenIDCt* function, from the CHStone suite [47], where all Pareto-solutions have been found. The figure shows the configurations plotted on a 2 dimensional space of area and latency. Synthesised configurations are marked with filled dots, non synthesised ones with crosses, and Pareto-optimal solutions of the design space with darker dots.

By analysing the exploration results in the area-latency space, it is hard to identify correlations among the different configurations. However, the same dataset can be observed–applying Principal Component Analysis [54]–as a function of the variance of the directives. Figure 5.16 represents the same configurations of Figure 5.1 in the directives space, after performing Principal Component Analysis (PCA) on the normalised directive values, thus reshaping them according to their variance. The figure plots the data only as function of first and second PCA components. Crucially, in this representation Pareto-optimal solutions tend to cluster together, or to be distributed on a common plane.

As the configuration space X is the Cartesian product among the different directive sets K_n, the variance of the design space depends on the variance of each directive set K_n, which in turn depends only on the sets cardinality. By shaping the design space as a unitary N-dimensional lattice (where each dimension represents a directive associated to a directive set K_n) I therefore obtain a representation of the HLS problem in which, as in PCA representation, Pareto-points are closely clustered, a key characteristics exploited by the exploration

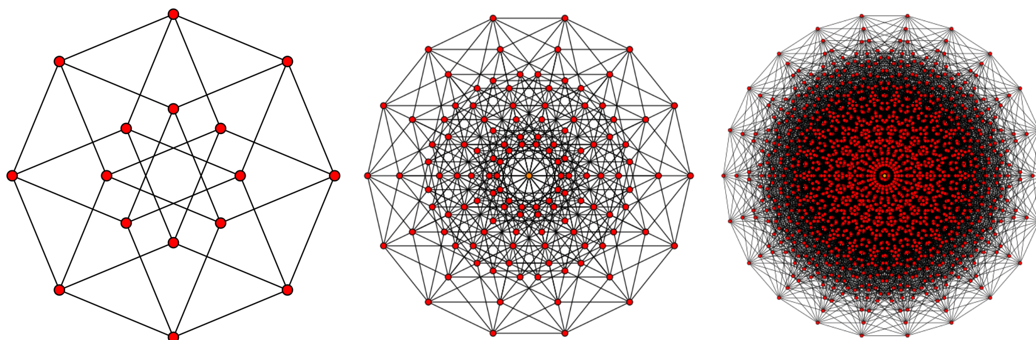


Figure 5.17. Petrie projections of N-dimensional lattices. (Left) 4-dimensional lattice. (Center) 7-dimensional lattice. (Right) 11-dimensional lattice. For simplicity, only the vertices of the N-dimensional lattices are shown.

framework illustrated in the following sections.

Figure 5.17 shows three examples of lattice. The figure shows the petrie polygon orthographic projections of a 4-dimensional lattice, 7-dimensional lattice, and 11-dimensional lattice. For simplicity, only the vertices of the N-dimensional lattices are shown.

To strengthen this observation I have analysed, again for the *ChenIDCt* benchmark, the distribution of the euclidean distances in the lattice space among Pareto-optimal solutions, with respect to the ones considering all other configurations. Results, shown in Figure 5.18, highlight that the average distance among Pareto-optimal solutions is significantly smaller than the average distance among non Pareto-optimal ones. Indeed, the Mann-Whitney-Wilcoxon Test (or U-test, [70]) applied to the two distance sets results in a small p-value of $1.8843e^{-08}$, which indicates their statistical difference and the possibility to distinguish between them. Similar outcomes were retrieved for the other benchmarks evaluated (described in Section 5.2.2), and summarised in Table 5.3.

5.2.1 Exploration Methodology

The proposed lattice-traversal DSE strategy, as shown in Figure 5.19, is divided in 3 phases: A) *Lattice creation & initial Sampling*, B) *Selection of lattice Pareto-neighbours*, B) *Synthesis & lattice labelling*. After lattice creation and initial sampling, steps b) and c) are repeated until either a user-defined budget of synthesis runs is reached or the neighbourhood search does not return any more feasible candidate.

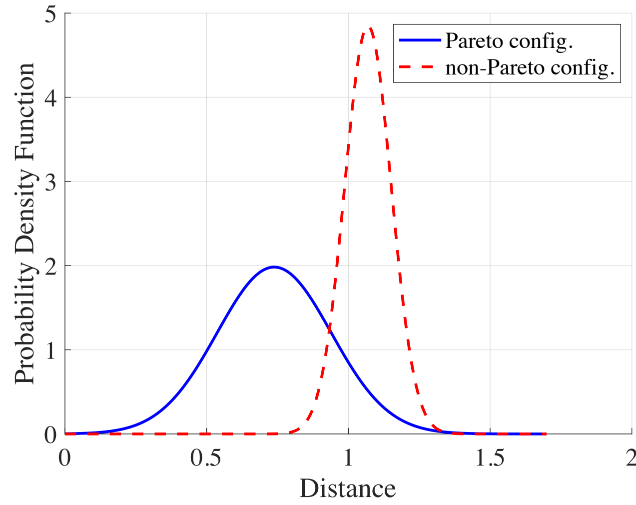


Figure 5.18. Distance distribution among Pareto configurations and among non-Pareto ones.

Table 5.3. Distribution of the distances among Pareto-configurations and non-Pareto ones, for different CHStone benchmark functions[47].

Function	**non-Pareto**		**Pareto**		**U test**
	μ	σ	μ	σ	p-value
ChenIDCt	1.0689	0.0823	0.7382	0.2013	1.8843e-08
Encode	1.4719	0.0634	1.0323	0.1440	4.5067e-13
Decode	1.4729	0.0638	1.2319	0.1778	4.4463e-09
Reflection	2.0762	0.0481	1.5082	0.1317	2.1293e-11
Autocorr	1.4363	0.0607	1.2803	0.1158	3.9376e-18

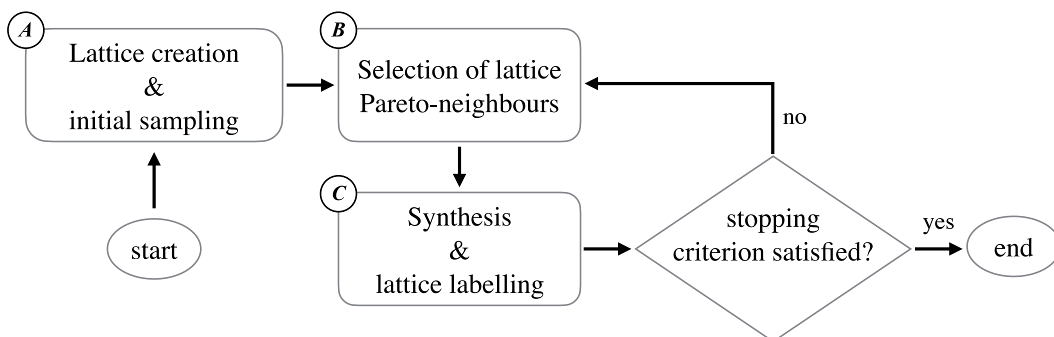


Figure 5.19. Overview of the lattice exploration algorithm.

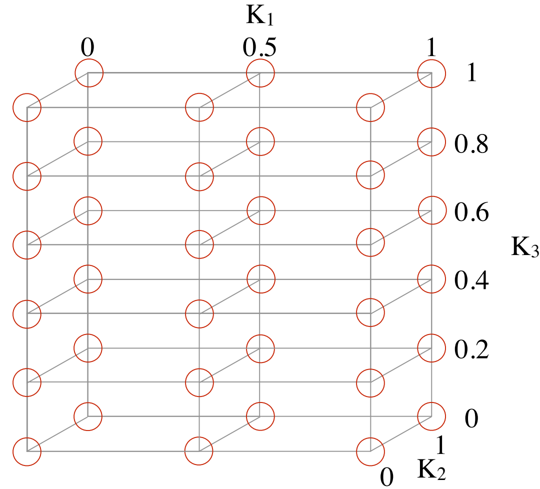


Figure 5.20. Example of 3-dimensional lattice for 3 directive sets with cardinalities: $|K_1| = 3$, $|K_2| = 2$ and $|K_3| = 6$.

Lattice representation and initial sampling.

Given the design space X, composed of N different sets of directive values K_N, the different configurations x are represented as an N-dimensional lattice structure where each dimension is a directive set. Each node in the lattice is a configuration x and each dimension goes from 0 to 1 with M admissible values, with M being the number of directive values in K_i. The lattice is ordered according to the K_N directive values. Figure 5.20 shows an example of 3-dimensional lattice for 3 directive sets with cardinalities: $|K_1| = 3$, $|K_2| = 2$ and $|K_3| = 6$.

After lattice creation, an initial sampling of the design space generates the first set of configurations $\bar{X}$ from which the first approximation of the real Pareto frontier is obtained. $\bar{X}$ is composed of a set of n unique configuration directives x. We adopt the same initial sampling strategy proposed in [36], which uses a U-shaped Beta distribution to sample n directive values from the corresponding admissible set K_i, generating n different configurations.

Then, area and latency information is retrieved by synthesising the configurations in $\bar{X}$ to obtain the corresponding values $\bar{S}$. From $\bar{S}$, the Pareto-solutions are identified and a first approximation $\bar{P}$ of P is obtained. Figure 5.21 shows an example of lattice, discriminating the nodes between synthesised solutions (belonging to $\bar{X}$), non-synthesised ones ($\widehat{X} = D \backslash \bar{X}$) and, among the nodes in $\bar{X}$, the Pareto configurations $\bar{P}$.

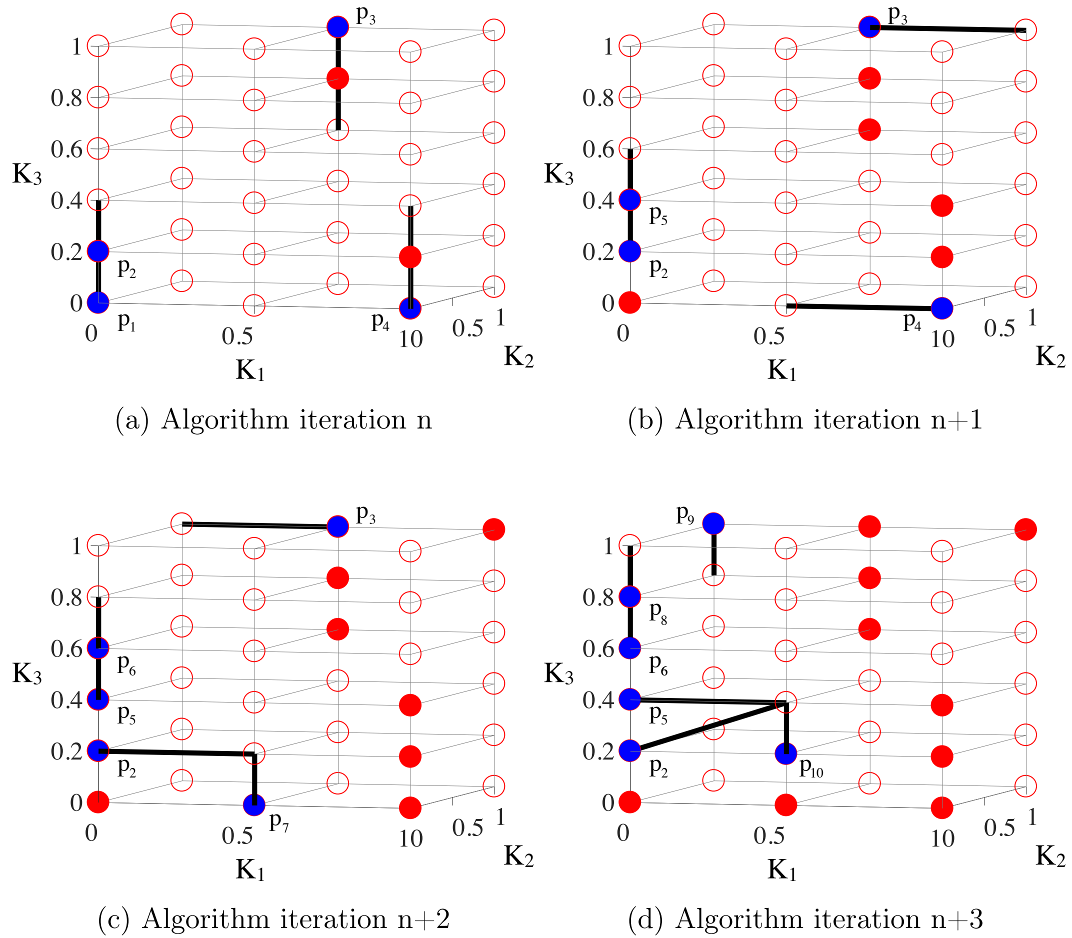
Selection of lattice Pareto-Neighbours

Once the N-dimensional lattice is created and the initial points are sampled, few further design points in $\widehat{X}$ are selected as candidates for synthesis. To this end, for each element in $\bar{P}$ the algorithm considers the euclidean distance among the elements in $\widehat{X}$ and selects the closest ones, building a new node set $\widehat{X}^*$. $\bar{X}$ is then updated with the new selected elements, removing these from $\widehat{X}$.

Synthesis & lattice labelling

After the selection of the Pareto-Neighbours $\widehat{X}^*$, these elements are synthesised and the corresponding area and latency results are retrieved. Then, the new Pareto-frontier is computed and the set of $\bar{P}$ nodes is updated. The algorithm continues to explore the design space selecting at each iteration a new set of $\widehat{X}^*$ until the design space is exhaustively explored or a stopping criterion–such as a budget of synthesis–is satisfied and the exploration ends.

Example: Figure 5.21a shows an example of exploration with six synthesised configurations and a set of Pareto-solutions $\bar{P}$. In the figure, the synthesised solutions $\bar{X}$ are the filled dots, while the empty ones are the elements of $\widehat{X}$. The filled dots can be further divided into Pareto-solutions (darker, blue dots) and dominated solutions (lighter, red ones). Among the elements of $\bar{X}$, four are Pareto-solutions: p_1, p_2, p_3 and p_4. The algorithm searches for Pareto-solution neighbours selecting first the configurations closer to a point p_i. The selection of the new configurations moves in the direction of the dimension with the maximum variance–in this case the directive set K_3. The thicker lines show which configurations are selected as neighbours. At the following iteration, Figure 5.21b, a new Pareto-element has been discovered and sets $\bar{P}$ and $\bar{X}$ have been updated. At this point, the exploration from p_1 stops–since p_1 is now dominated–while exploration for the new set of $\bar{P}$ proceeds as before. For Pareto-solutions p_3 and p_4, the nearest neighbours in $\widehat{X}$ are along lattice dimension K_1. While for p_4 there is only one close element, p_3 has two neighbours at the same distance. In this case p_3 explores randomly one of the two solutions. In the next iteration (Figure 5.21c) $\bar{P}$ and $\bar{X}$ are updated; p_3, which has not found a Pareto-solution in the neighbour which was previously selected among two, now tries the other one; the other points continue moving towards closest non-synthesised neighbours. At this iteration, the exploration leads p_2 and the newly discovered p_7 to select the same neighbour, and the corresponding configuration will be synthesised only one time. Finally, Figure 5.21d shows a further iteration of the exploration algorithm. In this case, 3 different paths reach the same configuration, which is



(a) Algorithm iteration n (b) Algorithm iteration n+1

(c) Algorithm iteration n+2 (d) Algorithm iteration n+3

Figure 5.21. Example of lattice exploration. Dots represent the X directive configurations. Configurations belonging to $\widehat{X}$ have not yet been synthesised, and are marked as empty dots. Synthesised configurations (hence belonging to the $\bar{X}$ set) are indicated with filled dots. Among these, darker filled dots are Pareto-solutions $\bar{P}$. Thick lines connect Pareto-solutions to their closest configurations, which will be synthesised in the next iteration of the exploration algorithm.

identified as the nearest neighbour in $\widehat{X}$ of p_2, p_5 and p_{10}.

Local searches

The naïve algorithm implementation described in the previous section requires to store in memory, for the duration of the exploration, the entire multi-dimensional lattice. Such requirement could hamper the scalability of the lattice-based ap-

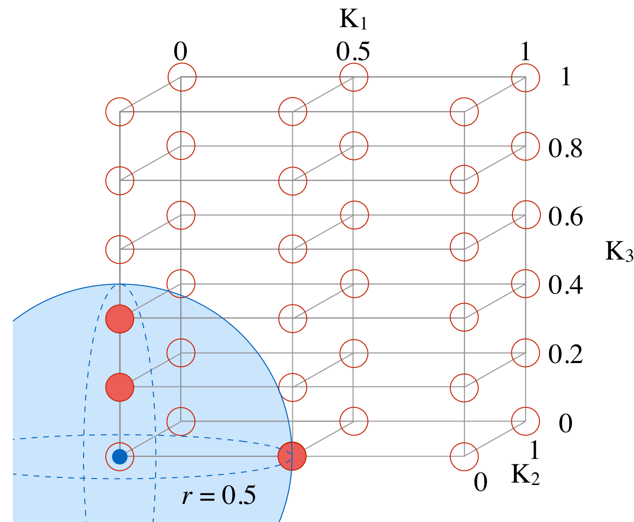


Figure 5.22. Example of local search with an exploration sphere of radius $r = 0.5$. The dots are configurations in the lattice. The filled dots are the possible candidate configurations included in the Exploration Sphere.

proach, since lattice creation may even be impossible for large designs, due to the exponential relation between the number of directives and the number of lattice points.

To address this problem, I devised an optimisation that only performs local searches in the neighbourhood of the $\bar{P}$ points, and therefore does not require representation of the entire lattice. The identification of Pareto-Neighbours is then performed *only within* an exploration sphere of short and tuneable radius (Figure 5.22), by iteratively generating the configurations included in the sphere centred in a $p \in \bar{P}$ point.

To perform local searches, information of the already-synthesised configurations must still be preserved, to avoid multiple synthesis of the same design point and multiple selection of the same point as a closest neighbour. To this end I use a tree structure (Synthesised Configuration Tree, SCT). The tree has as many levels as the design space directives (plus the tree root), with each level having at most as many branches as the admissible directive values. At the start of the exploration, the SCT is empty, except for the root node. Then, each time a synthesis is performed, a new leaf is added, as well as the proper path connecting it to the root. The tree would eventually become completely populated in the case of an exhaustive exploration (in which all configurations are visited), but remains sparse when only a small fraction of the design points are synthesised.

A second tree structure (Visited Configuration Tree, VCT) is dynamically con-

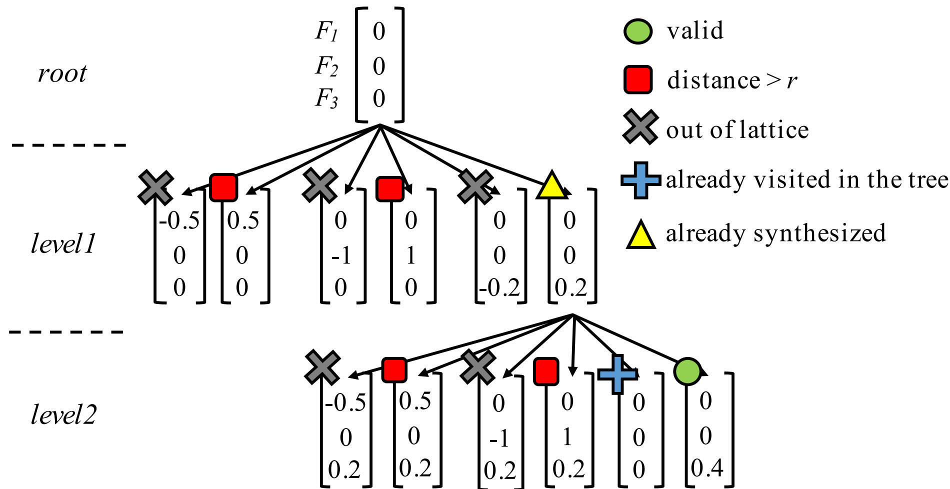


Figure 5.23. VCT structure generated by a neighbourhood search of radius $r = 0.4$, considering the lattice structure is of Figure 5.21 and starting on the [0,0,0] point. At the first tree level, three of the resulting configurations are out of the lattice bounds and two exceed the radius constraint. If the node [0,0,0.2] has already been synthesised, the search proceeds to a second level, where only [0,0,0.4] is a valid neighbour point.

structed at each neighbourhood search. As shown in Figure 5.23, the root node of this tree contains the directive values of the point at the centre of the exploration sphere. Then at each level of the tree, one directive value is incremented or decremented by the directive minimum distance. A test is then performed to check if any of the configurations at each VCT node is not encoded in the SCT (and has not, therefore, been synthesised). If this is the case for exactly one VCT node, its configuration is the desired p neighbour. If instead multiple candidates are retrieved, the one at the minimum distance is selected. Finally, if no solutions are found, the radius constraint is iteratively increased to allow exploration of further configurations. VCT branches are pruned if a) any directive value is < 0 or > 1 , indicating that the related configuration is outside of the lattice bounds, b) the configuration vector of a node has already been visited elsewhere in the tree, and c) the distance between the configuration values of a VCT node and the ones of p is $> r$. Hence, VCTs contain at most as many nodes as the number of lattice nodes belonging to the exploration sphere.

Besides recasting the nearest-neighbour search as a local problem, this formulation has the benefit of providing a tuning parameter in the form of the maximum allowable radius r_{max} to be explored. By changing it, it is possible to tune

the extent of local searches and in effect limit the maximum depth of VCTs. The effect of different r_{max} settings are studied in Section 5.2.2.

5.2.2 Results

Experimental setup

I have implemented the lattice-based HLS exploration methodology in the Python language. All configurations for the considered benchmark functions were synthesised using VivadoHLS from Xilinx [126], targeting a Kintex7 FPGA, with a clock constraint of $10ns$. The same benchmarks adopted for the cluster-based heuristics detailed in Table 5.1 have been considered. The explorations have been evaluate comparing the results obtained by the proposed methodology with the ground-truth obtained through an exhaustive exploration of the design space. Note that the both the cluster-based approach and the lattice-traversing exploration framework can be interfaced to any HLS synthesis tool and target any synthesis directive, as long as a notion of performance and cost can be derived from the outputs, and inputs can be expressed with directive sets having a non-infinite cardinality.

As done by other related works (e.g. [64][36] [71][86][114][138][36]), I have evaluated the quality of the performed explorations in terms ADRS.

Parameter tuning

In Section 5.2.1, I introduced a tuneable parameter r_{max}, that constrains Pareto neighbour searches. Figure 5.24 reports the effect of varying its value on the achievable ADRS, for the *Decode* function. By setting it to the minimum lattice distance ($r_{max} = 0.25$), only a very small lattice neighbourhood is considered. This choice entails that only Pareto neighbours along the directive values having the highest cardinality are explored. In addition, I have considered r_{max} values of 0.5, 1, and ∞. In the latter case, the search is only bounded by the lattice size, and hence proceeds until any non-synthesised point is found, if one exists in the design space. Results show that a small radius suffices to perform high-quality explorations: while a value of 0.25 results in a relatively poor value of final ADRS, setting the radius to 0.5 already enables to retrieve a very good approximation of the real Pareto frontier while still exploring only a small portion of the design space.

In turn, reducing the size of the Exploration Sphere implies that few neighbours must be evaluated during local searches, minimising the memory footprint

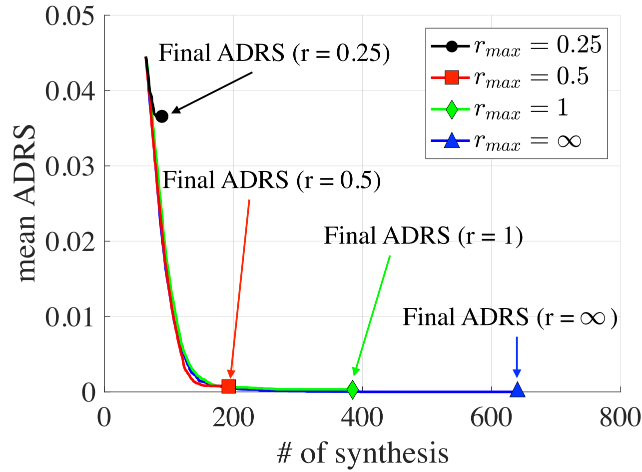


Figure 5.24. ADRS obtained with lattice explorations for the Decode function, with different r_{max} constraints.

of the lattice-based algorithm, while also reducing its run-time. Indeed, Figure 5.25 shows an exponential relationship between the maximum radius and the size of the VCT search tree described in Section 5.2.1, as well as the time required for its traversal. Similar results were obtained for the other considered benchmarks. They are reported in Table 5.4, showing the VCT size for different r_{max} values, and the run-time required for a single neighbourhood search. In all cases, a constrained radius $r_{max} = 0.5$ offers a good tradeoff between the quality of the obtained results and the portion of design space evaluated at each iteration.

The fast and effective explorations deriving from local searches result in much lower workloads compared with state of the art alternatives, as the comparative evaluation illustrated in Figure 5.26 showcases. The data in Figure 5.26 disregards the time required for synthesis, being independent from the exploration strategy. This characteristic enables the tackling of very large design spaces, such as the one illustrated later in this section.

In an additional round of experiments, I have investigated the robustness of *Lattice-expl* when the initial sapling size parameter is varied. In the previous experiments an initial sampling size equal to 10% of the design space has been considered. In this regard, Table 5.5 shows the number of synthesis required, for each functions, to reach an ADRS of 0.01, when adopting initial samplings equal to 1%, 2%, 5% and 10% of the design space size. The results highlight that our methodology is able to smartly navigate the design even when starting from few initial configurations.

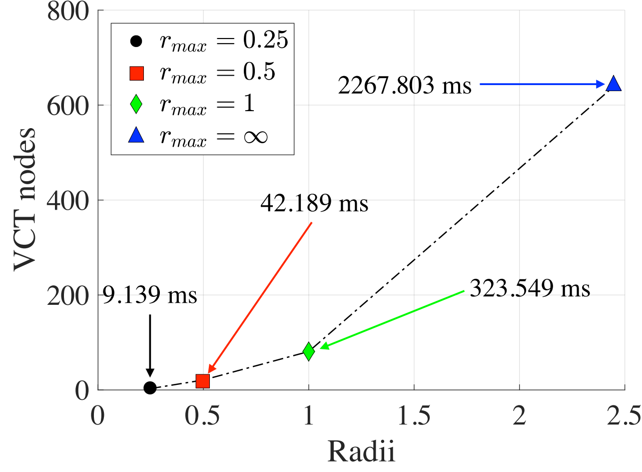


Figure 5.25. VCT tree size and average time required for a lattice neighbourhood exploration with different r_{max} constraints, targeting the Decode function.

Table 5.4. Memory and run-time evaluation, varying the radius of the Exploration Sphere.

		Function				
		ChenIDCt	Encode	Decode	Autocorr	Reflection
$r_{max} = 0.25$	Time $[ms]$	7.545	6.418	9.139	23.899	25.718
	ADRS	0.0618	0.0273	0.0367	0.0086	0.0083
	VCT nodes	3	3	3	7	1
$r_{max} = 0.5$	Time $[ms]$	35.700	38.313	42.189	202.509	79.376
	ADRS	0.0021	0.0039	0.0007	0.0017	0.0083
	VCT nodes	39	21	21	81	7
$r_{max} = 1$	Time $[ms]$	153.297	309.019	323.549	1392.983	149.943
	ADRS	0	0	0	0	0
	VCT nodes	111	81	81	218	43

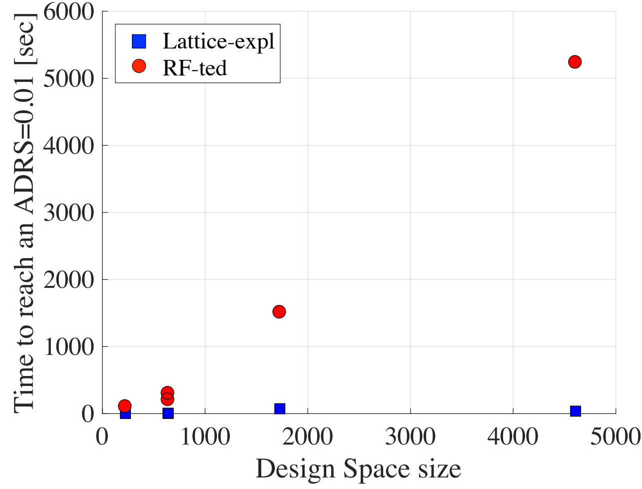


Figure 5.26. Run-times required by the RF-Ted[64] and Lattice-expl ($r_{max} = 0.5$) algorithms to reach an ADRS of 0.01, considering the benchmarks in Table 5.1.

Table 5.5. Number of synthesis (in percentage with respect to the design space) required to obtain an ADRS = 0.01, with $r_{max} = 0.5$, varying the size of the initial sampling.

	Initial sampling			
Function	*1%*	*2%*	*5%*	*10%*
ChenIDCt	60 (27%)	67 (30%)	61 (27%)	52 (23%)
Encode	180 (28%)	159 (25%)	111 (17%)	102 (16%)
Decode	201 (31%)	117 (18%)	86 (13%)	103 (16%)
Autocorr	838 (48%)	680 (39%)	589 (34%)	363 (21%)
Reflection	246 (5%)	240 (5%)	332 (7%)	460 (10%)

State of the art comparisons

In Figures 5.27-5.31 the performance of the proposed methodology (*Lattice-expl*) is compared with two state of the art algorithms: the heuristic proposed by Liu et al. (*RF-ted*, [64]), re-implemented for comparison, and the *Clust-beta* [36], presented in 5.1. Since both the *Lattice-expl* methodology and the *Clust-beta* have a probabilistic component due to the initial sampling, for those cases I ran the explorations 100 times, averaging the results. An initial sampling size equal to 10% of the design space has been been used as a sampling size. The work from [64] is based on Random Forest, and has been shown to outperform other

strategies based on different machine learning algorithms [150], [138], [86] and [71].

The comparisons are illustrated in Figures 5.27 to 5.31. The figures report the ADRSs achieved by the three methodologies, starting after the initial sampling phase. Across all benchmark functions, *Lattice-expl* reaches a tangibly better (lower) ADRS with a smaller number of synthesis at the end of the exploration. However, in few case in the early stage of the exploration, the *Clust-beta* methodology is able to obtain slightly better results than *Lattice-expl*, due to the greedy nature of *Clust-beta*.

In addition to the comparison with the state of the art, to showcase the high degree of scalability of the lattice-based exploration methodology, I have performed an exploration of the *Autocorr* benchmark along 9 different directives (including clock frequency, unrolling factors of every loops, function inlining and bundling options). As in this case the resulting design space size exceeds *1.5 million* points, we could not derive the ground truth of the real Pareto implementations, required to compute the quality of solutions in terms of ADRS. Figure 5.32 instead plots the Pareto frontier retrieved by our framework after a number of different synthesis budgets, highlighting that the lattice-based approach *can be employed even in such extremely large design spaces*, and showing an effective iterative improvement of the Pareto frontier.

This work has been published in the Proceeding of IEEE International Conference in Computer Design (ICCD) 2018 [37].

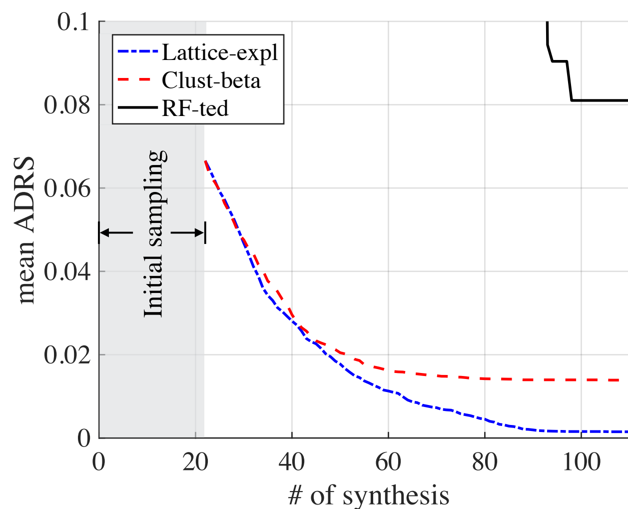


Figure 5.27. ADRS curves comparisons for the `ChenIDCt` benchmark.

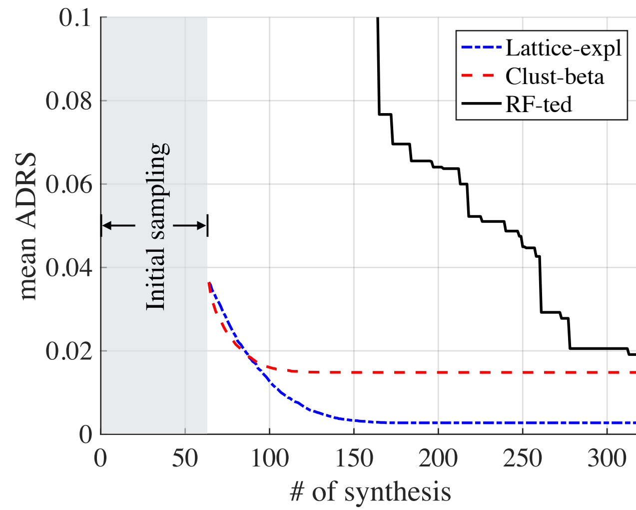


Figure 5.28. ADRS curves comparisons for the `Encode` benchmark.

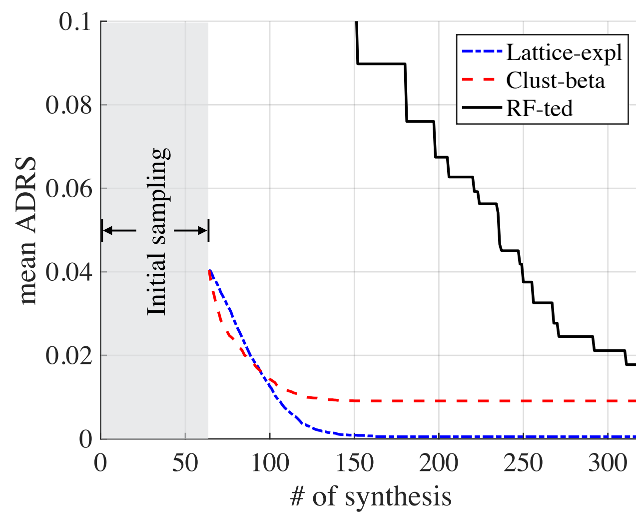


Figure 5.29. ADRS curves comparisons for the `Decode` benchmark.

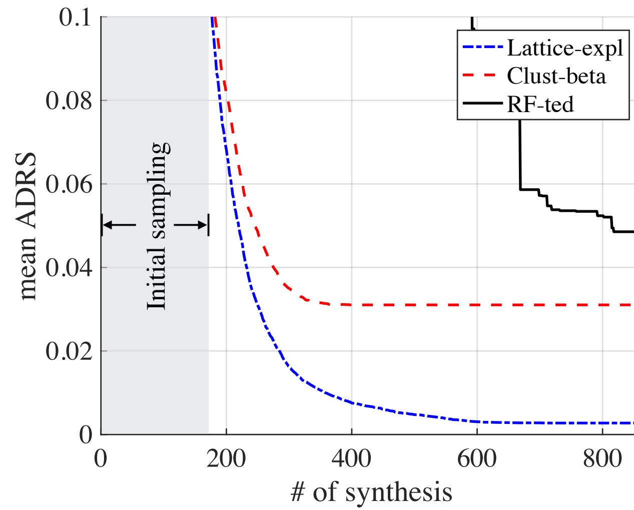


Figure 5.30. ADRS curves comparisons for the `Autrocorr` benchmark.

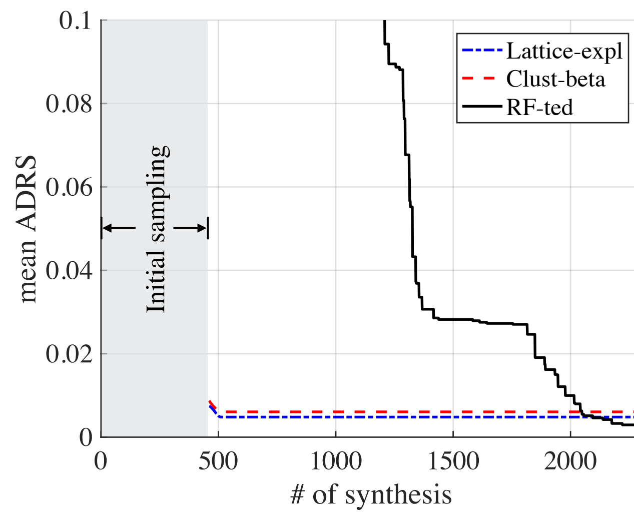


Figure 5.31. ADRS curves comparisons for the `Reflection` benchmark.

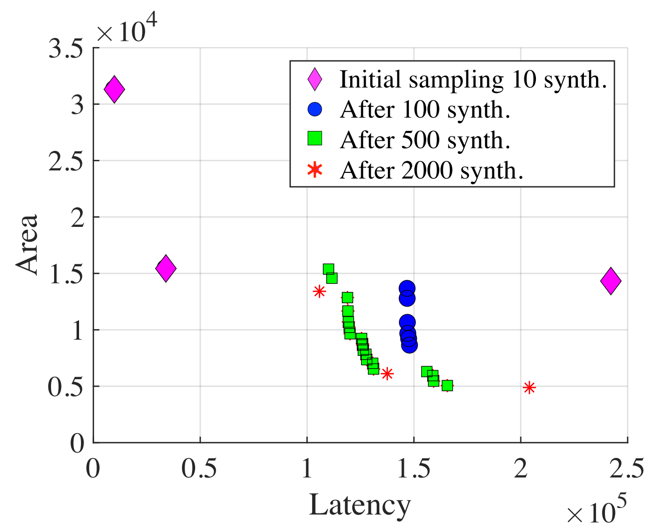


Figure 5.32. Pareto frontier evolution for the `Autrocorr` benchmark with more than 1.5 million possible configurations.

Chapter 6

Transfer Learning Driven Design Space Exploration

In Chapter 5 two different Refinement-based methodologies have been presented. The two approaches start the DSE relying on an initial set of synthesis retrieved trough the use of an initial sampling approach. After this initial step, both the methodology start navigating the design space aiming at leveraging the knowledge of the synthesised configurations in order to discover new Pareto solutions. While these strategies are effective in discovering an approximation of the Pareto frontiers of the target designs, it is clear that the necessity to build the initial set of knowledge from scratch every time is a limitation of the two approaches.

To this end, I have investigated the the feasibility of effectively *harnessing the knowledge of past synthesis outcomes to guide the optimization of new designs*. This new approach, differently from the previous two, does not belong to the Refinement-based category defined in Chapter 4, but falls in the Learning-based classification. In the next sections the details of the proposed strategy leveraging prior knowledge from past DSEs (Section 6.1) will be described. Then, I present a database of DSEs aiming at helping researchers developing new strategy and allowing an easier comparison among existing and future DSE methodologies (Section 6.2).

6.1 Leveraging Prior Knowledge

When optimizing a design with HLS, an expert designer starts by identifying which directives are applicable. For example, given the code in Snippet 6.5, the designer may be interested in exploring unrolling factors for loops, combined

Listing 6.1. `last_step_scan` (target).

```
1 void last_step_scan(int bucket[SIZE], int sum[RADIX]){
2   int i, j, k;
3   loop_1:for(i = 0; i < RADIX;i++){
4     loop_2:for(j = 0; j < BLOCK; j++) {
5       k = (i * BLOCK) + j;
6       bucket[k] = bucket[k] + sum[i];
7     }
8   }
9 }
```

Listing 6.2. `get_delta_matrix_weights2` (source).

```
1 void get_delta_matrix_weights2(double delta_weights2[N_NODES
    *N_NODES], double output_difference[N_NODES], double
    last_activations[N_NODES]) {
2   int i, j;
3   loop_1:for(i = 0; i < N_NODES; i++) {
4     loop_2:for(j = 0; j < N_NODES; j++) {
5       delta_weights2[i * N_NODES + j] = last_activations[i]
          * output_difference[j];
6     }
7   }
8 }
```

with different degrees of partitioning for the input/output arrays.

Furthermore, the designer may recall to have already optimized in the past a design with a similar code structure, such as the one reported in Snippet 6.2. Indeed, even if they are not identical (e.g., the loop boundaries and the memory access patterns differ), the two code snippets have some structural similarities: they both iterate over two nested loops and process data provided in input to the function through pointers. These similarities may be sufficient to suggest adapting those directives that lead to optimal implementations for Snippet 6.2 to the case of Snippet 6.5, instead of starting the DSE by trying anew many combinations of directives.

The designer's empirical strategy to tackle the DSE task hence consists of three main steps: a) identify the main structural characteristics in the code of the target design, b) pinpoint a similar already-explored design, and finally c)

transfer the knowledge from the source design to the new target design.

The proposed methodology performs these steps, but, differently from the above-described scenario, operates in a systematic and automated way. Section 6.1.2 shows that explorations, guided by prior knowledge, yield close approximations of the Pareto-optimal results from an exhaustive approach, while requiring very few synthesis runs. The methodology discussed in this section answers the following three research questions.

R.Q.1: From an HLS perspective, how can similarities among designs be quantified?

In general, code written in a high-level programming language such as C/C++ or SystemC is ill-suited for the automatic identification of structural similarities. Therefore, an abstract representation *that only retains the characteristics of interest for HLS optimizations*, e.g., the structure of loops and that of memory access patterns, is proposed. Such a representation (termed *specification encoding*) is automatically generated with a custom compiler pass. Since the representation is in the form of a string of symbols, a *string-similarity algorithm* is used to quantify the similarity in terms of computational patterns that exist between a source design (from a library capturing prior knowledge) and the target design.

R.Q.2: How can the similarity between directive choices for different designs be assessed?

Besides the specification code, the other aspect affecting the HLS results is the choice of HLS directives. Indeed, a proper source of previous knowledge should have a choice of directive values similar to the one of the target. As an example, if a loop can be unrolled by only a small degree in a source, little information can be leveraged to optimize a loop in the target for very high unrolling factors. A domain-specific language is introduced to describe succinctly the set of directives associated with a design, as well as a metric to measure the similarity between sets of directives associated with the source and target designs. Then, in a *source selection strategy* step, design and directive similarities are combined to identify the most promising source for the given target design.

R.Q.3: How to infer from prior knowledge HLS directives that give optimal results?

A strategy that transforms the HLS directives for the source design into HLS directives of the target design, as shown in the lower part of Figure **??**, addresses this last question.

In the next subsection, the answers to these three research questions are discussed in details.

6.1.1 Standard Approach VS Leveraging Prior Knowledge

Recalling the definition from Chapter 3, for a design T, $\mathcal{X}_T$ denote the set of all possible synthesis configurations, which is in general an impractically large set. In practice, designers focus the exploration on a smaller portion of the design space of T by trying a subset $X_T \subset \mathcal{X}_T$. The DSE task returns a set of Pareto configurations, $P(T,X_T)$ which is a subset of X_T. This subset is obtained by first (1) performing $|X_T|$ HLS runs on T, one run for each $x \in X_T$, and then (2) by selecting only those configurations that turn out to be Pareto configurations.

Now, assume that before performing the DSE task for T (the *target* design), the designer has performed the DSE task for another design S (the *source* design), thereby obtaining $P(S,X_S)$ for a given subset X_S of the configuration set $\mathcal{X}_S$. Furthermore, assume that a function $g : X_S \rightarrow X_T$ exists that transforms a configuration for the source design into one for the target design, i.e.

$$g(x_s) = x_t \tag{6.1}$$

with $x_s \in X_S$ and $x_t \in X_T$. With the help of function g, the designer can leverage *prior knowledge* on the source design, in order to perform a DSE for the target design with a potentially much smaller number of HLS runs.

Let X_T^S be the set of all configurations for the target design T that are obtained by transforming the Pareto configurations (up to a certain Pareto frontier rank) of the source design, i.e.:

$$X_T^S = \{g(x_s) | x_s \in P(S,X_S)\} \tag{6.2}$$

By synthesizing the target design T with the configurations in X_T^S, the set $P(T,X_T^S)$ can be obtained, as an approximation $\widehat{P}(T,X_T)$ of the set of Pareto configurations $P(T,X_T)$.

Figure 6.1 showcases the difference between a standard approach and one leveraging previous knowledge.

Note that this approximation requires $|X_T^S|$ HLS runs, while the derivation of the actual set of Pareto configurations would require $|X_T|$ HLS runs. Tuning the maximum Pareto frontier rank whose configurations are transformed from source to target, the synthesis effort and the approximation of $P(T,X_T^S)$ by $\widehat{P}(T,X_T)$ can be traded-off. I have explored the effect of varying this parameter in Section 6.1.2. Since for a given design T the number of Pareto configurations $|P(T,X_T)|$ is, in general, much smaller than the number of configurations $|X_T|$, if sets $P(T,X_T)$ and $P(S,X_S)$ are of comparable sizes then leveraging prior knowledge allows a major reduction in the number of time-consuming HLS runs while deriving the approximated set $\widehat{P}(T,X_T)$.

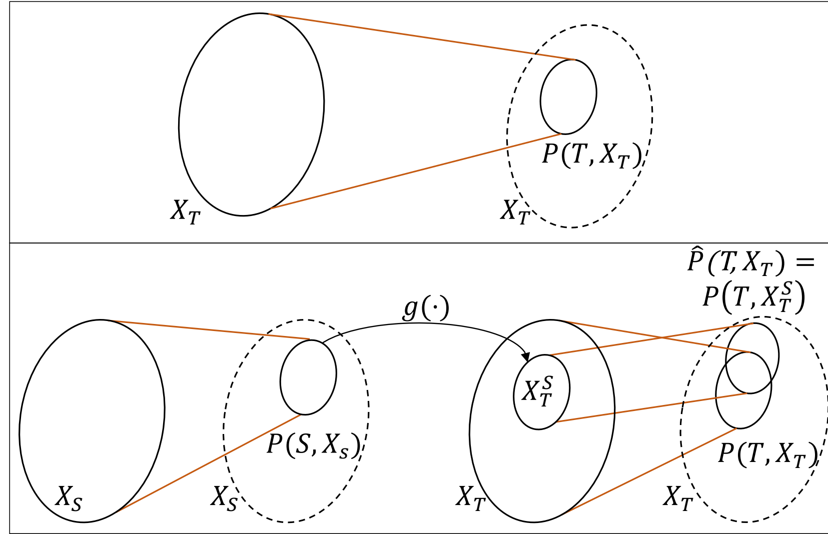


Figure 6.1. (Top) Standard approach: the designer defines a set of configurations to be explored, X_T, given a target design T. Only after synthesizing all the X_T configurations, Pareto optimal ones $P(T,X_T)$ are identified.
(Bottom) Approach leveraging prior knowledge instead: the configurations to be synthesized X_T^S are inferred from the $P(S,X_S)$ of a similar design S. By synthesizing T with $X_T^S \ll X_T$ configurations, a close approximation $\widehat{P}(T,X_T)$ of the Pareto frontier is obtained.

Moreover, the degree to which $P(T,X_T)$ is approximated by $\widehat{P}(T,X_T)$ depends on the choice of a proper source design S to derive the prior knowledge for the given target T. To this end, I introduce a novel and concise representation to encode the specification and a configuration space of each design via an abstract characterization called *signature*. Then, a similarity metric between the signatures is defined. If the signatures of two designs (source and target) have high similarity, then Pareto configurations for the source desing–when transformed to configurations for the other–may approximate well the actual Pareto configurations for the target design. Moreover, signatures are also employed to automate the transformation of Pareto configurations between source and target spaces, thereby realizing function $g(\cdot)$ of Equation 6.1.

Figure 6.2 illustrates the overall flow of the methodology. Given a target design's specification and configuration space as input, the proposed strategy (A) derives the signature of the design, and (B) employs a similarity metric over such signature to search, in a database of already performed DSEs (the sources), the most similar one. Once a source is selected, the Pareto configurations *for that*

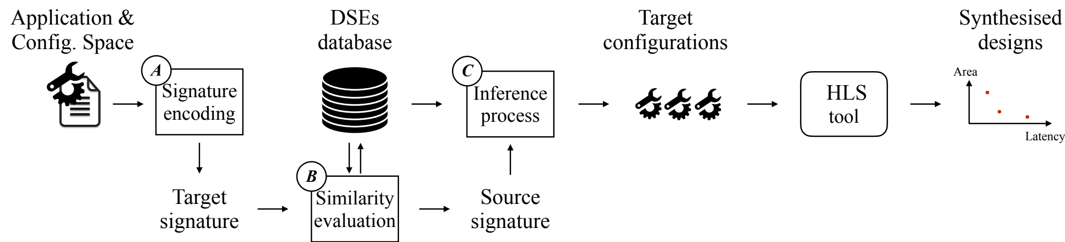


Figure 6.2. Methodology flow. The target design and its configurations space are encoded into a signature, which is compared to the ones of existing DSEs. The source having the most similar signature is selected to drive the inference process and generate the target configurations.

source are extracted, and (C) they are transformed by an inference process into valid configurations for the target.

Signature Encoding

This step aims at characterizing a DSE with a compact representation that abstracts the specification (code) and the associated configurations (set of applied directives). The proposed *specification encoding* (SE) and *configuration space descriptor* (CSD) capture these two aspects. The combination of SE and CSD uniquely define a *signature encoding*.

Specification Encoding. A specification encoding describes those aspects of an HLS specification that can be targeted by HLS directives, such as the presence of loops and read/write operations, while disregarding anything that is not interesting from a HLS-driven DSE perspective.

The encoding process generates a string representation of the specification that highlights the source code structure.

Table 6.1 shows the encoding scheme adopted and the correspondence between the string symbols and the code constructs. The SE is derived from the C/C++ specification through an LLVM [62] pass. I have extended the compiler to parse the abstract syntax tree and produce the SE string. The last column of Table 6.1 also shows the HLS directives that can be associated with each code construct. Curly braces are used to identify the scope associated with symbols, thus allowing hierarchical representations (e.g., a function containing multiple nested loops).

Running Example: Given the function `last_step_scan` from Snippet 6.5 and the encoding in Table 6.1, the proposed SE is `F{PP}L{L{RRW}}`. The encoding states that the function (`F`) receives two parameters by reference (`PP`), it has two

Table 6.1. Specification encoding of design source code.

Symbols	Code constructs	HLS directives
F	Function definition	None
V	Function parameter passed by value	None
P	Function parameter passed by reference	Partitioning and resource
A	Arrays definition or declaration	Partitioning and resource
S	Struct definition or declaration	Partitioning and resource
L	Loops	Unrolling
R	Read operations	None
W	Write operations	None
C_{id}	Function call	Inlining
{ .. }	Scope	None

nested loops and the innermost loop performs two reads and one write operations (`L{L{RRW}}`). Likewise, the SE for Snippet 6.2 is `F{PPP}L{L{RRW}}`, showcasing a similar, but not identical, structure.

Configuration Space Descriptor. A domain-specific language to concisely describe a user-defined configuration space has been defined. For source designs, CSDs describe which configurations are available in its design space, while for a target a CSD indicates the set of configurations that a designer wishes to explore. Each line of the descriptor encodes a *knob* (a type of directives, the location, and the selected parameter values) that the designer considers for the DSE task. For a directive with multiple parameters, a set of values for each parameter is specified.

Listing 6.3. Configuration Space Descriptor of `last_step_scan`.

```
1 resource;last_step_scan;bucket;{RAM_2P_BRAM}
2 resource;last_step_scan;sum;{RAM_2P_BRAM}
3 array_partition;last_step_scan;bucket;1;{cyclic,block};
    {1->512,pow_2}
4 array_partition;last_step_scan;sum;1;{cyclic,block};
    {1->128,pow_2}
5 unroll;last_step_scan;last_1;{1->128,pow_2}
6 unroll;last_step_scan;last_2;{1,2,4,8,16}
7 clock;{10}
```

Listing 6.4. Configuration Space Descriptor of `get_delta_matrix_weights2`.

```
1 array_partition;get_delta_matrix_weights2;delta_weights2;
     1;{cyclic ,block};{1−>256,pow_2}
2 array_partition;get_delta_matrix_weights2;output_difference;
     1;{cyclic ,block};{1−>64,pow_2}
3 array_partition;get_delta_matrix_weights2;last_activations;
     1;{cyclic ,block};{1−>64,pow_2}
4 unroll;get_delta_matrix_weights2;loop_1;{1−>64,pow_2}
5 unroll;get_delta_matrix_weights2;loop_2;{1−>64,pow_2}
6 clock;{10}
```

Running Example: Given the function `last_step_scan` in Snippet 6.5, the associated CSD is shown in Snippet 6.3. The descriptor defines seven different knobs that can be associated with the function `last_step_scan`. Line 1 of Snippet 6.3 shows a knob with a single value: it associates a dual-ported Block RAM (BRAM) to the array `bucket` that is the input of the function. Line 3 instead defines a knob governing the array partitioning directive defined by all the pairs having one of two partitioning strategies (`cyclic` and `block`) as first component, and the ten possible partitioning factors (all the powers of two from 1 up to 512) as the second one.

The CSD is parsed and the resulting set of configurations of the design space is generated as follows:

$$X = K_1 \times K_2 \times \cdots \times K_N \tag{6.3}$$

where N is the number of considered knobs, and K_i is the set of values related to each i knob, i.e. the set of values that the directive associated to the knob i can assume. For a directive with multiple parameters, K_i is the Cartesian product among each set of values. The size of the configuration space is then given by its cardinality ($|X|$).

Similarity evaluation

To choose a candidate source design for a target design, a similarity metric between their signature encodings is computed. This similarity is computed given the similarities of the design SEs and CSDs:

$$Sim = \alpha Sim_{SE} + (1-\alpha) Sim_{CSD} \quad \alpha \in [0,1] \tag{6.4}$$

The parameter α in Equation 6.4 weights the contributions of the SE and CSD similarities. The best source candidate, selected according to the similarity metric, is used to transfer knowledge from source to target design during the inference step.

Figure 6.3-left shows the similarity matrix for the 39 functions in the MachSuite Benchmarks. Each row shows the similarity between a target design and all the candidate source designs; the diagonal elements show the similarity of the design to itself. The figure highlights a high similarity variance that discriminates well between similar and dissimilar sources. In Section 6.1.2, I show that the chosen similarity metric leads to an effective selection of the source for the given target.

SE similarity. Since the specification encoding is expressed as a string, a string-based algorithm can be used to assess the similarity between SEs. In this approach, I adopt the *Longest Common Subsequence (LCS)* metric [89]. This metric returns a score $Sim_{SE} \in [0, 1]$, whose value is closer to 1 the more two strings are alike. Given two application signatures of length n and m, the running time of a dynamic programming approach is $O(n \times m)$[127]. However, it is important to observe that the signature encoding reduces the size of the input codes to strings with a small number of characters; therefore the time required for the similarity evaluation is negligible with respect to the synthesis time.

The LCS metric is defined as follow:

$$Sim_{AS}(A_i, B_j) = \begin{cases} 0 & if \quad i = 0 \quad or \quad j = 0 \\ Sim_{AS}(A_{i-1}, B_{j-1}) + 1 & if \quad i, j > 0 \quad and \quad a_i = b_j \\ max\{Sim_{AS}(A_i, B_{j-1}), & if \quad i, j > 0 \quad and \quad a_i \neq b_j \\ \quad Sim_{AS}(A_{i-1}, B_j)\} & \end{cases} \tag{6.5}$$

Where a_i and b_j are the ith and jth element of the strings A and B respectively. Figure 6.3-center expresses the Sim_{SE} matrix for the same 39 functions, where each row shows the similarity between a target design and all candidate sources.

Running Example: Given the specification encoding of the target design `F{PP}L{L{RRW}}` and the source design `F{PPP}L{L{RRW}}`, the resulting SE similarity score is 0.93.

CSD Similarity. The similarity between two CSDs is assessed by comparing the knobs K_i for a target configuration space X_T (for design T) to the knobs K_j for a source configuration space X_S (for design S) using a mapping function $M_{T,S}$, which relates each knob of the target CSD to a specific knob of the source CSD:

$$M_{T,S}(K_i) = K_j \tag{6.6}$$

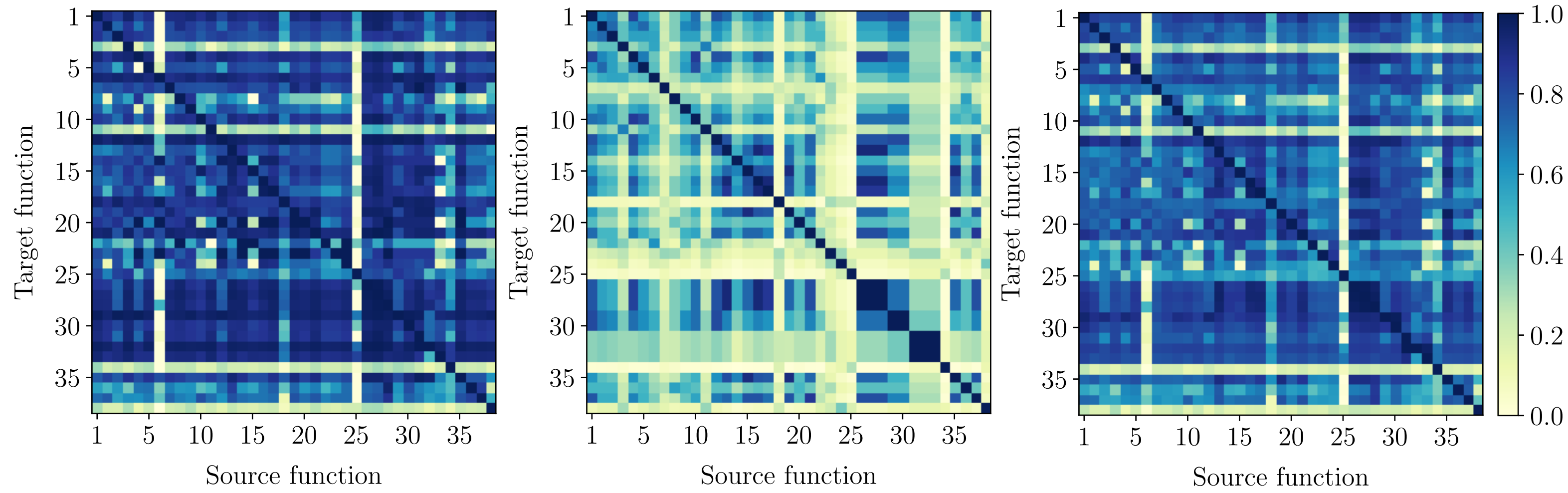


Figure 6.3. (Left) Signature similarity matrix obtained with $\alpha = 0.2$. (Center) Specification Encoding similarity matrix. (Right) Configuration Space Descriptor similarity matrix. Darker color expresses a higher similarity. Each row of the matrix shows the similarity between a target design and the source ones. The indices on the axes corresponds to the function IDs in Table 6.2.

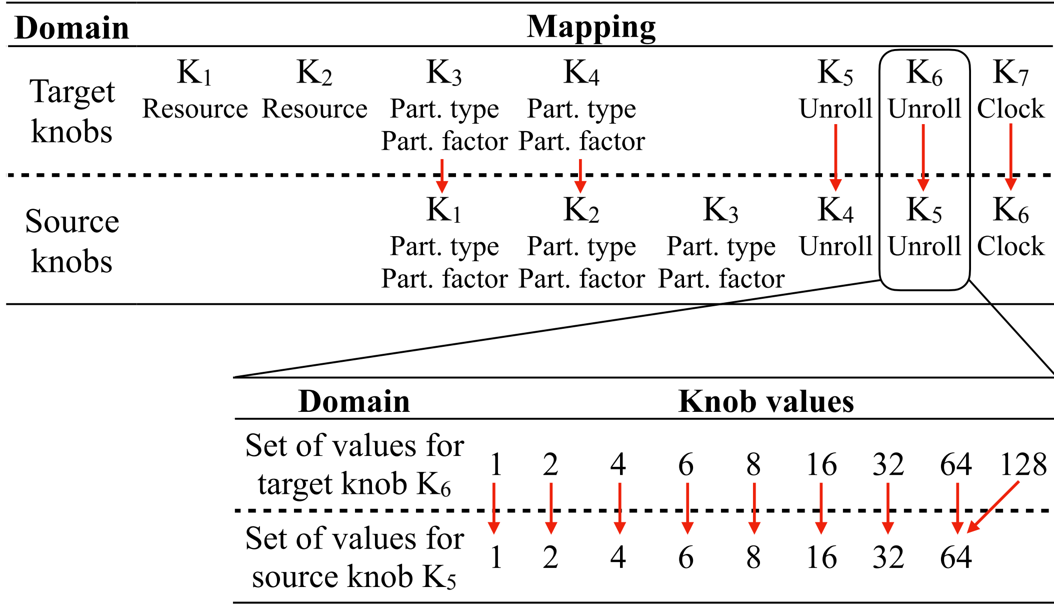


Figure 6.4. Top: mapping between the knobs of source and target CSD shown as example (Snippet 6.5 and 6.2). Bottom: correspondence between the knob value sets of knobs K_6 and K_5 in target and source, leading to a distance Δ of 1.

The function $M_{T,S}$ is determined through an alignment procedure. By iterating over the knobs of the target CSD from top to bottom, each knob is mapped to the first non-mapped knob of the same type belonging to the source CSD. Eventually, if no more knobs are available in the source, some target knobs may be left unmapped.

Running Example: Let us consider the target and source designs and their CSDs in Snippet 6.3 and Snippet 6.4, respectively. The CSD of function `last_step_scan` has seven different knobs–each knob is one line of the descriptor–while the CSD of function `get_delta_matrix_weights2` has only six knobs. Figure 6.4 shows the mapping $M_{T,S}$ between the two CSDs. Five out of seven knobs of function `last_step_scan` can be mapped by using `get_delta_matrix_weights2` as a source design; while knobs K_1 and K_2 of `last_ step_scan` are un-mapped.

Once a mapping $M_{T,S}$ is defined between a target configuration space X_T, having I different knobs, and a source configuration space X_S, having with J different knobs, their similarity is computed as follows:

$$Sim_{CSD} = 1 - \left[\frac{1}{I} \sum_{i=1}^{I} \Delta(K_i, M_{T,S}(K_i))/D_{MAX} \right] \tag{6.7}$$

where D_{MAX} is a normalization factor–constant across all the source candidates for a given target–such that $Sim_{CSD} \in [0,1]$. Then, $\Delta(\cdot)$ is a function measuring the minimum distance between a source knob and a target knob:

$$\Delta(K_i, K_j) = \sqrt{\sum_{n=1}^{|K_i|} (\min_{m=1}^{|K_j|} |\delta(k_n, k_m)|)^2} \quad k_n \in K_i, k_m \in K_j \tag{6.8}$$

The above equation sums up the distance between each target knob value k_n and the one that is closest to it among all source knob values k_m. The function $\delta(k_n, k_m)$ computes the distance between two knob values of the same directive type that has Z parameters, e.g., $k_n = (k_{n,1}, \cdots, k_{n,Z})$:

$$\delta(k_n, k_m) = \sqrt{\sum_{z=1}^{Z} |k_{n,z}, k_{m,z}|^2} \tag{6.9}$$

where numerical parameter values are casted to their respective log_2 value, and categorical parameter values are represented with one-hot encoding.

Since for unmapped knobs there is no correspondence between source and target, the distance $\Delta(\cdot)$ in Equation 6.8 is computed between the values of the target knob and the default value of the directive.

Figure 6.3-right shows the resulting Sim_{CSD} matrix for the functions in the MachSuite Benchmark Suite considered in this work.

Running Example: Given the mapping between the functions in the running example, for each target knob the distance with respect to the source one is measured. Figure 6.4 (bottom) shows the computation of the distance for the target knob K_6 mapped to the source node K_5, each having a single value set of possible unrolling factors. K_6 specifies 9 factors (from 1 to 128, all of them powers-of-two), while K_5 comprises 8 values (from 1 to 64). Since the δ is calculated among numerical values, the directive values are casted to their respective $log2$; therefore, the knobs discrepancy leads to a Δ equal to $(log_2 128 - log_2 64) = 1$. When accounting for all target knobs, the CSD similarity between the source and target is 0.97.

Inference

After a source design is identified for a target design, the inference process transfers knowledge from the source to the target configuration space, hence implementing Equation 6.1. In the first step of such process, the configurations belonging to the Pareto frontier in the source configuration space are extracted from a

Domain	Inference						
Source knobs			K_1 cyclic 256	K_2 cyclic 8	K_4 32	K_5 64	K_6 10
Target knobs	K_1 2P_BRAM	K_2 2P_BRAM	K_3 cyclic, block 1,…,256,512	K_4 cyclic, block 1,..,8,..,128	K_5 1,..,32,..,128	K_6 1,2,4,8,16	K_7 10

Figure 6.5. Inference from source to target design spaces from the running example. The inferred values of the HLS directive knobs are underlined in the bottom part of the figure.

library of prior knowledge. These are peeled from the source design space, allowing the identification of second-rank Pareto configurations. Then, it proceeds iteratively to extract higher ranked Pareto frontiers, until a certain number of these have been retrieved from the source design space.

Each selected configuration is transformed into a valid one in the target CSD. To this end, first, knobs in the source and target spaces are mapped according to the mapping function $M_{T,S}$ described in the previous section. If a target knob K_i is not be mapped to a source knob, the value of x_T^i value is fixed to the directive default, since no previous knowledge related to that knob can be leveraged. The values of all other knobs are instead inferred from the source design space.

Then, given a source configuration $x_S = [x_S^1, \dots, x_S^J] \in X_S$, with x_S^j being the value for knob j, the corresponding target configuration is set as $x_T = [x_T^1, \dots, x_T^I]$ $\in X_T$, where each component x_T^i are knob values associated to the knob i.

For each configuration component, the inference function ($g : X_S \rightarrow X_T$, introduced in Equation 6.1) is defined as follows:

$$x_T^i = \arg\min_n \{\delta(k_n, x_S^j)\} \tag{6.10}$$

where $\delta(\cdot)$ is the distance function defined in Equation 6.9, x_S^j is the value assigned to the $j-th$ knob of the source, and $k_n \in K_i$ is the set of all possible sets of values that the knob K_i of the target design can assume, as specified by its configuration space descriptor. Therefore, each target directive value x_T^i is assigned to the closest value to x_S^j among those specified in the target knob set for knob i.

Running Example: Let us assume that, among the many Pareto configurations of the source design in Sippet 6.2, one configuration has the directive values shown in Figure 6.5. Given the mapping function from Figure 6.4 and Equation 6.10, the source Pareto configuration are transformed into a valid target configurations. The partitioning factors–256 and 8 for K_1 and K_2 of the source–are

mapped to the closest partitioning factor values of the target knobs–respectively 256 and 8 for K_1 and K_2 of the target. Similarly, the same partitioning type–cyclic–is inferred from the source Pareto configuration for the target ones. Finally, the source unrolling factors and the clock, 32, 64 and 10 for K_4, K_5 and K_6 are mapped to 32, 16 and 10 for K_5, K_6 and K_7 in the target, respectively.

6.1.2 Results

Experimental setup

I have implemented the similarity evaluation and inference algorithms in Python, while the SE encoding was implemented in C++ as a custom compiler pass within the LLVM infrastructure, as described in Section 6.1.1.

The experiments targeted all of the functions in the MachSuite benchmarks collection [94], except those that expose very small design spaces, and those having a variable latency for different invocations during benchmark execution due to input-dependent control flows. In total, 11 designs were discarded. The resulting suite comprises 39 functions, which have on average 40 lines of code and 308 in the biggest case.

For each design, I have performed an extensive DSE across their configuration spaces up to tens of thousands of design points. Vivado HLS [126] was used to run synthesis with a target clock period of 10*ns* and targeting a ZynqMP Ultrascale+ (`xczu9eg`) FPGA chip. I have collected the design configurations and synthesis results in a MySQL database.

In order to control the configuration space size[1], I only employed power-of-two values for directives having a numerical range (e.g., loop-unrolling and array-partitioning factors), and, in some cases, I forced related knobs to have the same value (e.g., the partitioning factor of an array and the unrolling of a loop accessing it once every iteration). Such decision corresponds to the intuitive choice of constraints that a designer would impose when tasked with the exploration of the design space.

The extensive DSEs are used in two ways. On the one hand, the results are used as ground truth to assess the performance of the approach. On the other, they are used as a source of previous knowledge. In the latter case, a leave-one-out cross-validation is adopted, considering each design as a target using all others as candidate knowledge sources.

[1]Even for the simple case in Snippet 6.2, considering all loop unrolling factors, two types of resources, two types of partitioning and all partitioning factors would result in more than 10^8 configurations.

Lastly, similarly to [65, 36, 37, 146], the Average Distance from Reference Set (ADRS) metric is used.

Results

Outcome of Explorations. Table 6.2 summarizes the results of the explorations performed with the proposed methodology. It reports the target function *IDs* (used as indexes in Figure 6.3), their benchmarks and the function names. Moreover, for each case, it provides the function *IDs* and the function names of the source having the highest similarity score, the obtained ADRS values in the target space, the number of synthesized configurations derived from that source, and the size of the related configuration space ($|CS|$).

For the vast majority of targets, the proposed approach requires very few syntheses to reach low ADRS scores. As an example, when targeting `aes_addRoundKey` (row index: 20) while leveraging the knowledge of the `add_bias_to_activations` source, only 13 out of 500 possible synthesis rounds are performed, still resulting in a perfect identification of the Pareto frontier of best performing implementations (ADRS = 0). Only 35 out of thousands of configurations are synthesized for the `gemm` target (row index: 5) while reaching a very close Pareto frontier approximation (ADRS = 0.012).

The results of Table 6.2 were obtained by inferring up to the 10th-ranked Pareto frontier (as defined in Section 3) and by fixing the trade-off between SE and CSD similarity (introduced as α in Section 6.1.1) to 0.2. Both settings are explored in the rest of this section.

Tuning of the similarity function. Figure 6.6 shows the ADRS achieved when selecting the most similar candidates according to the similarity metric in Equation 6.4 while varying the parameter α, i.e., the relative weight of specification encoding and configuration space similarity. Data is shown on a logarithmic scale and in an aggregated form across all targets. Boxes encompass the first and third quartile of the ADRS values obtained by the DSEs of all targets, while the lines inside them indicate the median case. The skewers above and below each box are the upper and lower 1.5 interquartile. As before, I infer configurations up until the 10th-ranked Pareto frontier in the source design space.

Figure 6.6 shows that both SE and CSD have an impact on the quality of results and that CSD similarity generally has a more significant impact than the SE one. An α value of 0.2 both minimizes the interquartile range and the median ADRS.

Effectiveness of the similarity metric. Figure 6.7 highlights the importance of a proper source of previous knowledge in order to achieve effective explor-

Table 6.2. List of functions explored from MachSuite[94] (grouped by benchmark). The table reports: target function *IDs*, it benchmark name, target function name, source function *IDs*, source function names, ADRS value, number of synthesized configurations, and size of the configuration space ($|CS|$).

ID	Benchmark	Target function	Source ID	Source function	ADRS	# Synth.	|CS|
1	spmv ellpack	ellpack	28	get_delta_matrix_weights2	0.034	65	1600
2	bfs bulk	bulk	28	get_delta_matrix_weights2	0.010	39	2352
3	md knn	md_kernel	30	get_oracle_activations1	0.006	25	1600
4	viterbi	viterbi	28	get_delta_matrix_weights2	$8.7e^{-4}$	12	1152
5	gemm ncubed	gemm	28	get_delta_matrix_weights2	0.012	35	2744
6	gemm blocked	bbgemm	28	get_delta_matrix_weights2	1.689	35	1600
7	fft strided	fft	30	get_oracle_activations1	0.0007	15	1600
8	fft transpose	twiddles8	32	product_with_bias_input_layer	0.0002	24	64
9	sort merge	ms_mergesort	27	get_delta_matrix_weights1	0.322	31	1024
10		merge	30	get_oracle_activations1	0.262	19	4096
11	stencil stencil2d	stencil	28	get_delta_matrix_weights2	0.015	46	1344
12	stencil stencil3d	stencil3d	28	get_delta_matrix_weights2	1.88	16	1536
13		update	30	get_oracle_activations1	0.009	28	2400
14		hist	28	get_delta_matrix_weights2	0.007	46	4704
15		init	18	local_scan	0.078	68	484
16	radix sort	sum_scan	36	add_bias_to_activations	0.136	25	1280
17		last_step_scan	28	get_delta_matrix_weights2	0.004	90	800
18		local_scan	17	last_step_scan	0.005	71	704
19		ss_sort	32	product_with_bias_input_layer	0.0005	21	1792
20		aes_addRoundKey	36	add_bias_to_activations	0	13	500
21		aes_subBytes	29	get_delta_matrix_weights3	0	8	50
22		aes_addRoundKey_cpy	28	get_delta_matrix_weights2	0	71	625
23	aes	aes_shiftRows	16	sum_scan	0.013	8	20
24		aes_mixColumns	25	aes_expandEncKey	0	15	18
25		aes_expandEncKey	13	update	0.003	33	216
26		aes256_encrypt_ecb	4	viterbi	0.030	22	1944
27		get_delta_matrix_weights1	28	get_delta_matrix_weights2	0.002	139	21952
28		get_delta_matrix_weights2	27	get_delta_matrix_weights1	0.010	77	31213
29		get_delta_matrix_weights3	28	get_delta_matrix_weights2	0.030	222	21952
30		get_oracle_activations1	31	get_oracle_activations2	2.907	67	2401
31		get_oracle_activations2	29	get_delta_matrix_weights3	0.051	19	1372
32		product_with_bias_input_layer	34	product_with_bias_output_layer	3.560	3	1372
33	backprop	product_with_bias_second_layer	32	product_with_bias_input_layer	0	30	686
34		product_with_bias_output_layer	32	product_with_bias_input_layer	$2.5e^{-5}$	24	392
35		backprop	1	ellpack	$4.4e^{-5}$	4	2048
36		add_bias_to_activations	20	aes_addRoundKey	0.002	5	1372
37		soft_max	29	get_delta_matrix_weights3	0.053	9	64
38		take_difference	29	get_delta_matrix_weights3	0.0002	8	512
39		update_weights	4	viterbi	$1.1e^{-4}$	3	1024

ations. It depicts four DSEs of the target design `last_step_scan`, from the running example, leveraging different sources of previous knowledge. Each plot depicts the ground truth of the target design space resulting from its exhaustive exploration–gray dots–as well as the Pareto frontier retrieved with the inference process–dark blue line. The top-left DSE shows the result of inferring configurations from `get_delta_matrix_weights2` (ID 27), the best-ranked source according to the defined similarity metric. In this case, the Pareto frontier is very well approximated and obtains a small ADRS of 0.004.

The top-right picture of Figure 6.7 shows an example of DSE characterized

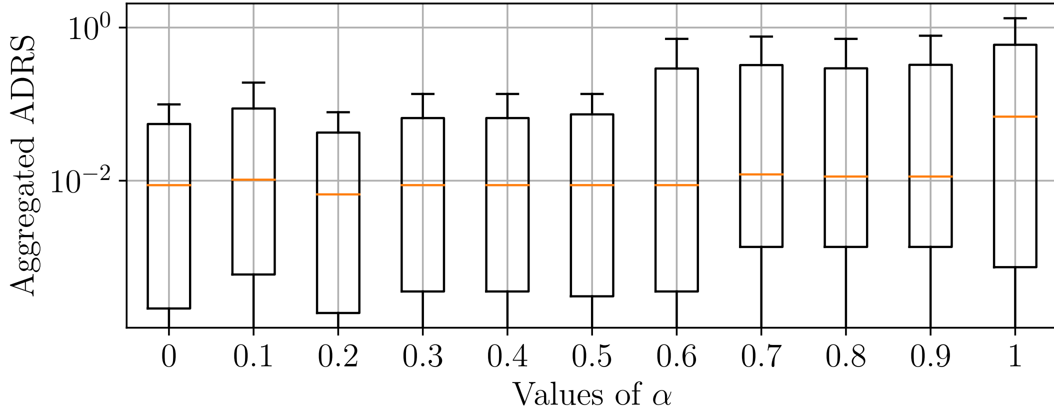


Figure 6.6. ADRS evolution while changing the value of α.

by a low SE similarity, resulting in a partial approximation of the Pareto frontier, since only a few knobs can be mapped from source to target. In this case, the inference process uses as source design the function `bulk` (ID 2), which is ranked 30th in order of similarity score. Similarly, the bottom-left picture shows the result of the DSE when `product_with_bias_output_layer` (ID 34) is employed as a source design. In this case, the similarity score is penalized by a low CSD similarity. Therefore, only a portion of the target design space can be explored, due to inadequate coverage of the knob values of X_T by the ones in X_S. The Pareto frontier is hence well approximated only for the X_T region for which previous knowledge is available, resulting in an ADRS of 1.5. Finally, in the plot at the bottom-right of Figure 6.7 are shown the result of the inference from `backprop` (ID 35). In this case, both a low SE similarity (few knobs can be mapped from source to target) and a low CSD similarity (for mapped knobs, knob values are distant between source and target spaces) can be observed, resulting in an extremely poor approximation of the Pareto frontier, since little previous knowledge can be harnessed. In this case, as reported in the figure, the retrieved Pareto frontier only comprises a single design point.

Figure 6.8 generalizes these findings by reporting the aggregated ADRS values when selecting the sources with the highest similarity score for each target, the second-best choices, etc. As in Figure 6.6, data are on a logarithmic scale. Boxes encompass the first and third quartile of the ADRS scores across all targets, while the line inside the boxes indicates the median case. An order of magnitude separates the best and third-best choice, after which performance remains almost constant and tangibly worse than in the case of the best-ranked source.

Finally, Figure 6.9 compares the aggregated ADRS values obtained by the pro-

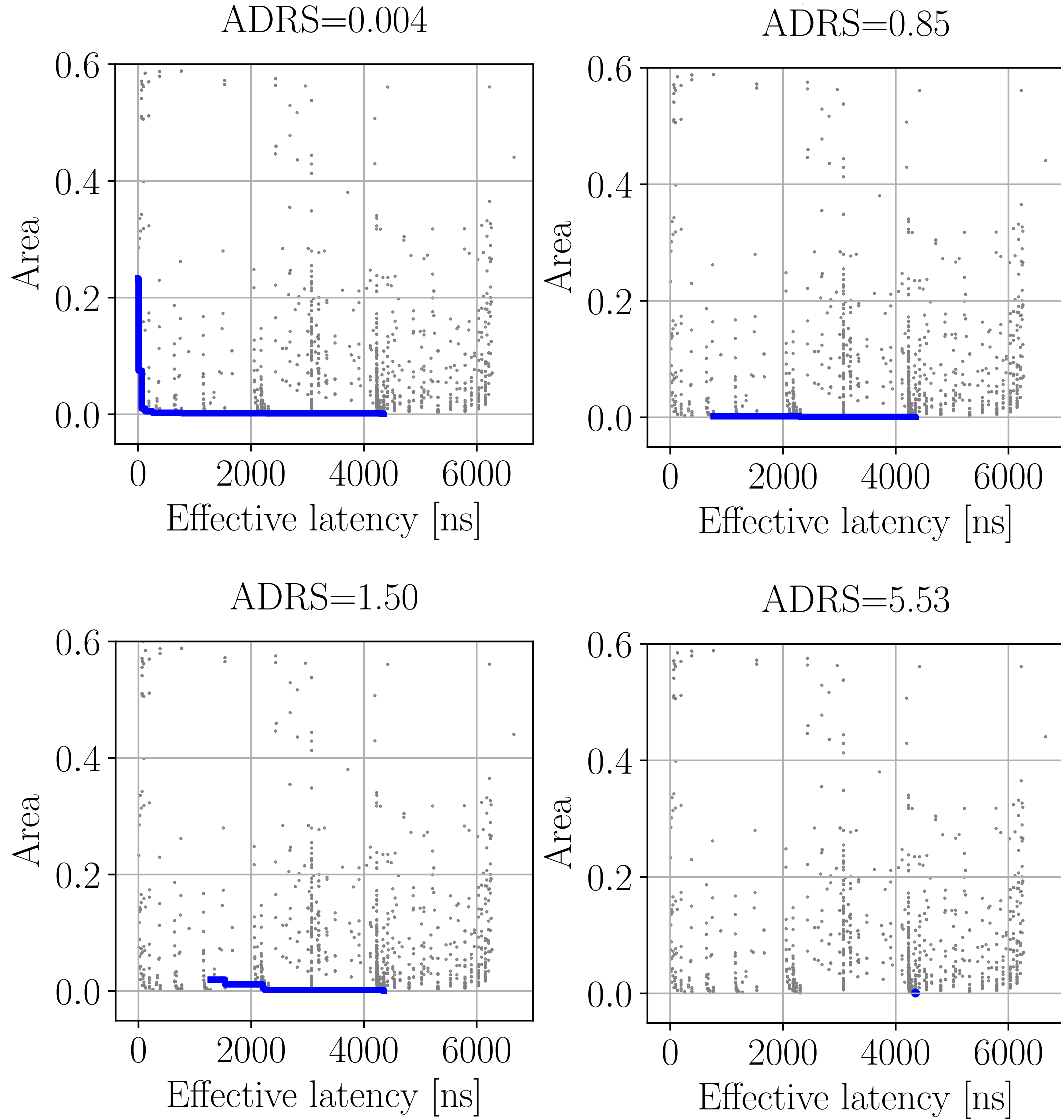


Figure 6.7. Example of DSEs targeting `last_step_scan`, inferring from different sources. (Top-left) Good Pareto-approximation obtained with the best candidate source. (Top-right, bottom-left, bottom-right) Low quality Pareto approximations, from sources having low CSD and/or AS similarity. Gray dots represent the ground-truth for the target design `last_step_scan`, while the dark blue line represents the Pareto frontier obtained performing the inference with different sources.

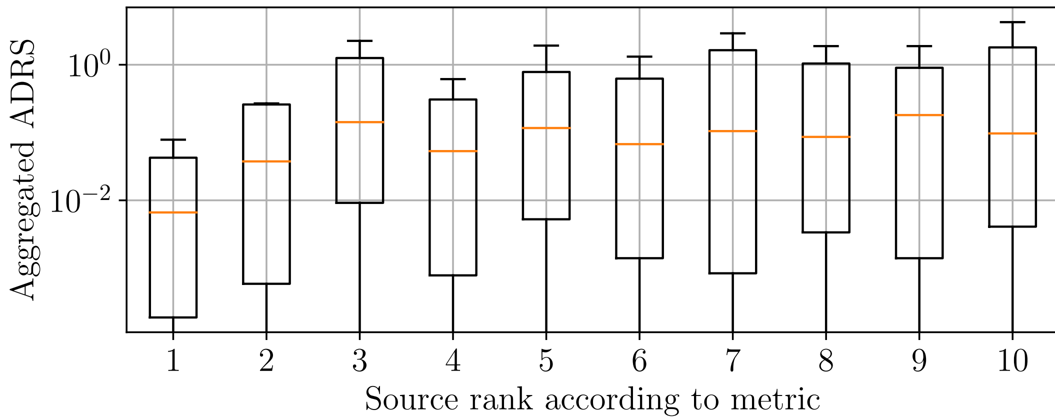


Figure 6.8. ADRS according to source selection by metric ranking.

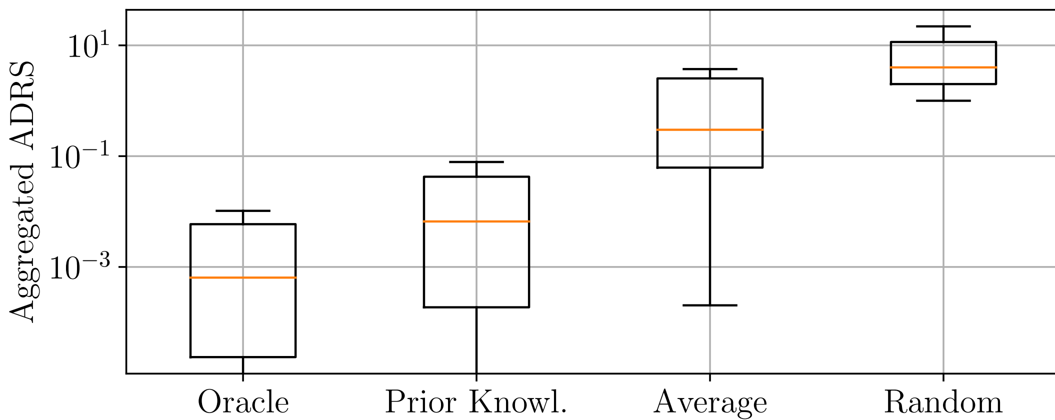


Figure 6.9. ADRS according to source selection criterion.

posed strategy (named Prev.Knowl. in the figure), i.e., leveraging the source with the highest similarity, with three alternatives: a perfect oracle always choosing a-posteriori the best source (Oracle), the average of selecting all sources for each target (Average), and a random sampling of the target design space (Random)– disregarding knowledge transfer altogether. For the latter case, several samplings equal to the ones required by Prev.Knowl. are performed, and the results are averaged over 100 different runs to minimize noise. A choice that is driven by the proposed similarity metric (Prev.Knowl.) performs three orders of magnitude better than a choice that is not considering past explorations at all (Random) and 12X better when compared with a blind choice for the source design (Average).

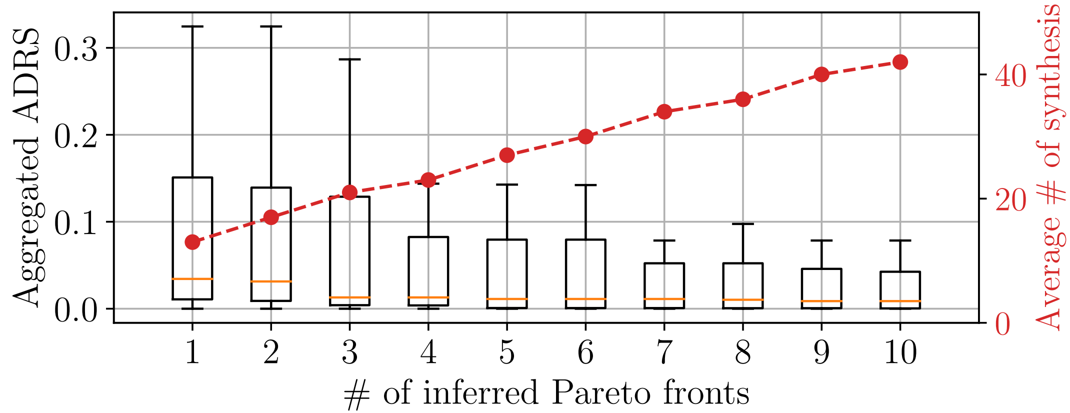


Figure 6.10. Cumulative evolution of the ADRS while inferring from multiple Pareto frontiers. The average number of synthesis performed while the number of Pareto frontiers increases is shown with the dashed line.

Table 6.3. Qualitative comparison with SoA methodologies. Average number of synthesis required to obtain an ADRS ≤ 0.04.

‖CS‖	Prev.Knowl.	Lattice [37]	Cluster [36]	RF-TED [65]	Zhong [146]
< 200	7	36	37	155	NA
< 700	10	64	64	391	19
< 1800	22	230	290	1588	31
< 6000	19	460	460	1903	32
< 16000	NA	NA	NA	NA	35
< 32000	38	NA	NA	NA	NA

Tuning the number of source Pareto frontiers. In a further round of experiments (Figure 6.10), I have investigated the effect of varying the number of selected source Pareto frontiers. Each boxplot shows the aggregated ADRS outcomes when inferring an increasing number of Pareto frontiers from the highest-similarity source to the target. While increasing the number of frontiers always lowers ADRS scores, diminishing returns can be observed for a number of frontiers ≥ 7. The number of required synthesis, instead, linearly increases with the amount of inferred Pareto frontiers.

Comparison with State of the Art. Table 6.3 compares Prev.Knowl. with four related works that also aim to automate the optimization of HLS designs. According to the taxonomy of Section 4, three of them are refinement-based approaches [36, 37] presented in Chapter 5, and [65], while one is a model-based

approach [146]. For fairness, the results are grouped in different brackets according to the configuration size of the employed benchmarks. When no benchmark is reported for a given size on past works, the corresponding table cell is marked with *NA*. In other cases, data shows the average number of synthesis runs required to reach an ADRS of 0.04, which [146] considers as an excellent Pareto frontier approximation (and which is attained by Prev.Knowl. in 29 out of 39 cases (see Table 6.2)).

The numbers in this table show that Prev.Knowl. greatly outperforms the refinement-based approaches (see the required number of synthesis in the first four columns), and this advantage grows with the size of the configuration space. The proposed strategy is *even* competitive with the model-based strategy of Zhong et al. [146] (see the last column), while being agnostic to the number and type of optimizations that can be considered.[2]

Leveraging previous knowledge across different clock constraints and platforms. All previous results assumed that the same clock constraint (10*ns*) and FPGA model (Xilinx Zynq xczu0eg) are employed for all synthesis runs. In practice, both these conditions may not be satisfied, as often past explorations may be performed for an FPGA than is different from the one of interest for a new design. Similarly, the clock constraints may, in general, not be the same for sources and target. Nonetheless, Prev.Knowl. is robust toward variations of FPGA models and operating frequencies because it relies on the Pareto-dominance relationship in the cost/performance space of implementations in each DSE composing the knowledge base, as opposed to relying on the actual values of area and latency. This relationship, and consequently the set of Pareto configurations of a design, is not tangibly affected by the employed clock period and FPGA.

To investigate this characteristic, I have observed the ADRS obtained by identifying the implementations of the first-rank Pareto frontier of the `last_step_scan` benchmark (13 out of 1600 implementations) synthesized with various clock periods, and inferring the related configurations for a different clock constraint. I have evaluated the result of the inference for target and sources with clock constraints of 5ns, 10ns, 25ns and 50ns. For all experiments, no changes in the inferred Pareto frontier were observed, and hence good approximations of the Pareto frontier were obtained. These results confirm that indeed the set of Pareto configurations are not tightly dependent on the operating frequency.

Similar remarks are obtained when varying the FPGA employed for the synthesis of source and target. I have observed the ADRS obtained by identifying the implementations of the first-rank Pareto frontier of the `last_step_scan`

[2]Zhong et al. only considers the loop unrolling and the dataflow directives.

Table 6.4. ADRS obtained by leveraging the knowledge of the get_delta_matrix_weights2 source while exploring the last_step_scan target, varying the clock constraints and FPGA models employed for the target benchmark.

Clock period (ns)	Technology	ADRS
10	ZynqMPU+	0.0044
5	ZynqMPU+	0.0044
25		0.0044
50		0.0044
10	Artix	0.0049
	Virtex	0.0045
	Kintex	0.0045

benchmark synthesized with various FPGAs (ZynqMPUltrascale+ xczu9eg, Virtex xc7vh580, Kintex xc7k352, and Artix xc7a100), and inferring the related configurations for a different platform. For all combinations of target and sources platforms, the Pareto configurations are the same. Even in this case, the inferred Pareto frontier perfectly approximates the one obtained by exhaustive exploration.

Concluding this round of experiments, Table 6.4 shows an example of applying Prev.Knowl., again considering last_step_scan as a target, while employing the knowledge base in Table II. The design get_delta_matrix_weights2 (ID 28) is identified as the most similar source and it is used to infer configurations up to its 10th-rank Pareto frontier–the same setting adopted for the results in Table II. In the first row of the table, provided for reference, both the clock constraint and the FPGA of the target design are equal to the ones used for the source. In rows 2-4, three different target clock constraints are used, while in rows 5-7 the target FPGA is different from the source one. In all cases, very similar, and small, ADRS are achieved, showcasing the robustness of the Prev.Knowl. methodology.

This work has been accepted at the CASES 2020 conference. The work has been presented at ESWEEK 2020 and it will be published in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD) [39].

6.2 A Database of Design Space Explorations

As shown in the previous Section and in Chapter 5, DSEs strategies are typically validated against exhaustive explorations performed ad-hoc by designers. More-

over, works such as [132], [39] and a recent editor's note from [31] highlight the importance of relying on prior knowledge to steer the HLS exploration process to reduce design process costs. However, performing the huge number of synthesis required for validation or for generating a high-quality knowledge base entails a very high effort, which at present must be repeated ex-novo when investigating the performance of a novel DSE methodology.

Against this backdrop, I aim at introducing DB4HLS, a database of high-level synthesis design space explorations. The database comprises more than 100000 design points, reporting the synthesis outcomes of exhaustive explorations performed on 39 functions from the MachSuite [94] benchmark suite. These DSEs are the one performed for the work leveraging prior knowledge presented in Section 6.1. In addition, I have proposed a simple domain-specific language to define design spaces, which concisely describes all available configurations, resulting in an open infrastructure that can be enriched by further contributions from the research community.

The database provides a rich set of DSEs by targeting the benchmarks of the MachSuite collection of designs [94]. DSEs for 39 out of 50 functions in the benchmark suite have been performed, discarding those having a variable latency due to input-dependent control flows, and those having very small design spaces. The considered functions present on average 40 lines of code, with the biggest having 308 lines of code.

For each function, an extensive DSE across their configuration spaces have been performed, generating up to tens of thousands of design points. These are reported in Table 6.2 introduced in Section 6.1.2.

All the implementations are generated with Vivado HLS [126] version 2018.2 to run syntheses with a target clock period of 10*ns* and targeting a ZynqMP Ultrascale+ (`xczu9eg`) FPGA chip. We constrained the design space sizes by employing only power-of-two values for directives having a numerical range (e.g., loop-unrolling and array-partitioning factors). Moreover, in some DSEs we forced two optimizations to take the same value, when intuitively such choice would lead to better cost/performance trade-offs (e.g., the partitioning factor of an array and the unrolling of a loop accessing it once every iteration).

Even when considering these constraints, the data collection required more than 4 years of single-core machine time. To speed up this process, GNU Parallel [122] was adopted to collect synthesis data from up to 60 parallel instances of Vivado HLS, allowing us to populate the database in approximately 25 days of wall-clock time.

Goal of the database is to provide a standardised synthesis data sets that will allow easier comparisons among DSE strategies, enabling fairer evaluations of

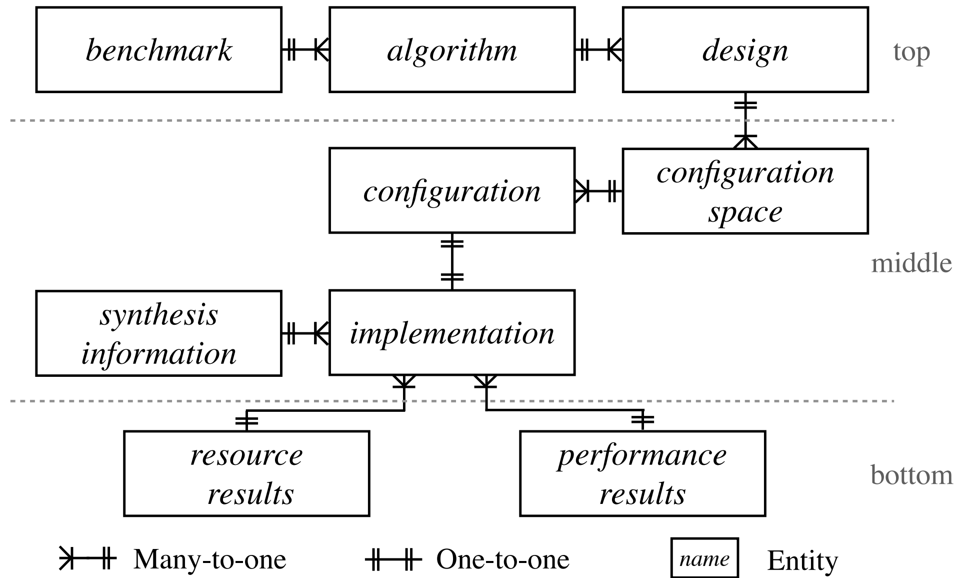


Figure 6.11. Scheme of the Entity-Relationship Diagram (ERD) of the DB4HLS syntheses database.

the strengths and weaknesses of each approach. It will also facilitate the development and assessment of future design exploration frameworks, spurring research in this challenging field.

6.2.1 DB4HLS Infrastructure

The DB4HLS infrastructure, in addition to the DSE data, offers:

1. database infrastructure hosting DSE in a structured and easy-to-access way.
2. a domain-specific language used to describe a configuration space for a target design.
3. an interface to generate new explorations and to further enrich the database.

The database structure, implemented in MySQL, comprises a description of the design targeted for exploration (top part of Figure 6.11), and that of the explored HLS optimizations applied to each design (middle part of Figure 6.11). Finally, it reports the resource and performance results obtained by synthesis (bottom part of the figure). Each of these components is described more in detail in the following.

Similarly to the taxonomy adopted in MachSuite [94], and the terminology introduced in Section 3.1, applications are identified by the *benchmark* they belong to (e.g.: `aes`), by the *algorithm* they realize (e.g.: `aes256_encrypt`) and by the *design* implementing such algorithms. As an example, two variants are provided by MachSuite for the `aes256_encrypt` algorithm (one using lookup tables to store encryption keys and one generating the values online), each corresponding to a separate design specified as C++ code.

A descriptor of the HLS optimizations considered for the DSEs are stored as entries in the *configuration space* table. Multiple explorations (hence, rows in the configuration space table) for the same design are possible, corresponding to different choices of optimizations, or to explorations targeting different tools/FPGAs, or even contributions from different researchers. An entry in the *configuration space* table is linked to many entries of the *configuration* table, where each entry indicates a specific element of the design space.

A line in the *configuration* table (that indicates the set of HLS optimizations defining a design space element) is linked to an entry in the *implementation* table. Furthermore, the *synthesis information* table provides additional information on each performed synthesis: the synthesis timestamp, the contributor that originated the data, the employed synthesis tool and version, and the targeted FPGA. Finally, each implementation links to one or more entries in the *resources* and *performance* tables, which report the synthesis outcomes. Resources are expressed as employed Flip-Flops, Look-Up Tables, Block RAMs (BRAM) and DSP blocks, while performance is reported in terms of effective latency.

6.2.2 A Domain-Specific Language for DSEs

Generating the different configurations associated with a DSE is a tedious and error-prone process when performed by hand. We therefore developed a Domain-Specific Language (DSL) to automatically and concisely define configuration spaces by employing Configuration Space Descriptors (CSDs).

Each line of a descriptor encodes a *knob*, which comprises a directive type, a label corresponding to its location in the design C/C++ code, and one or multiple sets of values. The number of sets is equal to the number of parameters required by the directive type. Values can be numerical when expressing optimizations such as loop unrolling or array partitioning factors, or categorical when determining the type of employed FPGA resources such as BRAM types. A shorthand is provided for expressing regular value series (e.g., a succession of power-of-two values). Finally, we provide a `@bind` decorator, which constraints the values associated with different directives.

Listing 6.5. `last_step_scan` design (C code).

```
1 void last_step_scan(int bucket[SIZE], int sum[RADIX]){
2   int i, j, k;
3   loop_1:for(i = 0; i < RADIX; i++){
4     loop_2:for(j = 0; j < BLOCK; j++) {
5       k = (i * BLOCK) + j;
6       bucket[k] = bucket[k] + sum[i];
7     }
8   }
9 }
```

Listing 6.6. Configuration Space of `last_step_scan`.

```
1 resource;last_step_scan;bucket;{RAM_2P_BRAM}
2 resource;last_step_scan;sum;{RAM_2P_BRAM}
3 array_partition;last_step_scan;bucket;1; {cyclic,block
    };{1->512,pow_2}
4 array_partition;last_step_scan;sum;1; {cyclic,block
    };{1->128,pow_2}@bind_a
5 unroll;last_step_scan;last_1;{1->128,pow_2}@bind_a
6 unroll;last_step_scan;last_2;{1,2,4,8,16}
7 clock;{10}
```

Snippet 6.5 shows the code of the function `last_step_scan` and Snippet 6.6 an example of DSL defined to describe the configuration space defined for its DSE. The DSL defines seven different knobs. Line 1 of Snippet 6.6 shows a knob with a single value: it associates a dual-ported BRAM to the array `bucket` that is the input of the function. Similarly, line 2 defines a dual-ported BRAM for the array `sum`. Line 3 instead defines a knob governing the array_partitioning directive defined by all pairs having one of two partitioning strategies (`cyclic` and `block`) as the first component, and the ten possible partitioning factors (all the powers of two from 1 up to 512) as the second one. The same is done in line 4, but defining a different set of partitioning factors (all the powers of two from 1 up to 128). Then line 5 and 6 define for `loop_1` and `loop_2` the associated set of unrolling factors to consider during the exploration, all the powers of two from 1 up to 128 and 16, respectively. Both line 4 and 5 have a binding decorator (`@bind_a`), that specifies that the array partitioning directive and the unrolling one must have the same partitioning and unrolling factor for all configurations described by the CSD. Finally line 7 defines the target clock.

The DSL generates the set of configurations of the design space as the Cartesian product of all knob values: $CS = K_1 \times K_2 \times \cdots \times K_N$; where N is the number of considered knobs, and K_i is the set of values related to each i knob, i.e. the set of values that the directive associated to knob i can assume, taking into account the restrictions imposed by the `bind` decorator. For a directive with multiple parameters, K_i is itself the Cartesian product among each set of values. The size of the configuration space is then given by its cardinality ($|CS|$).

The configuration space descriptor in Snippet 6.6 describes a configuration space with 1600 different configurations. Without the binding decorator, the cardinality of the configuration space would be 12800.

6.2.3 A Framework for Parallelising HLS Runs

Figure 6.12 gives a high-level view of the infrastructure, realized through Bash and Python scripts, which we provide to automate explorations and commit their outcomes in DB4HLS. Starting from a user-provided design and Configuration Space Descriptor (CSD), configuration files are automatically generated and stored in the database. Then, using GNU Parallel [122], a tunable number of instances of an employed HLS tool (we use Vivado HLS for data collection) are concurrently and independently executed, one for each configuration. As synthesis runs terminate, the retrieved performance and resources information is also stored in DB4HLS, and new HLS processes are launched until all configurations have been explored.

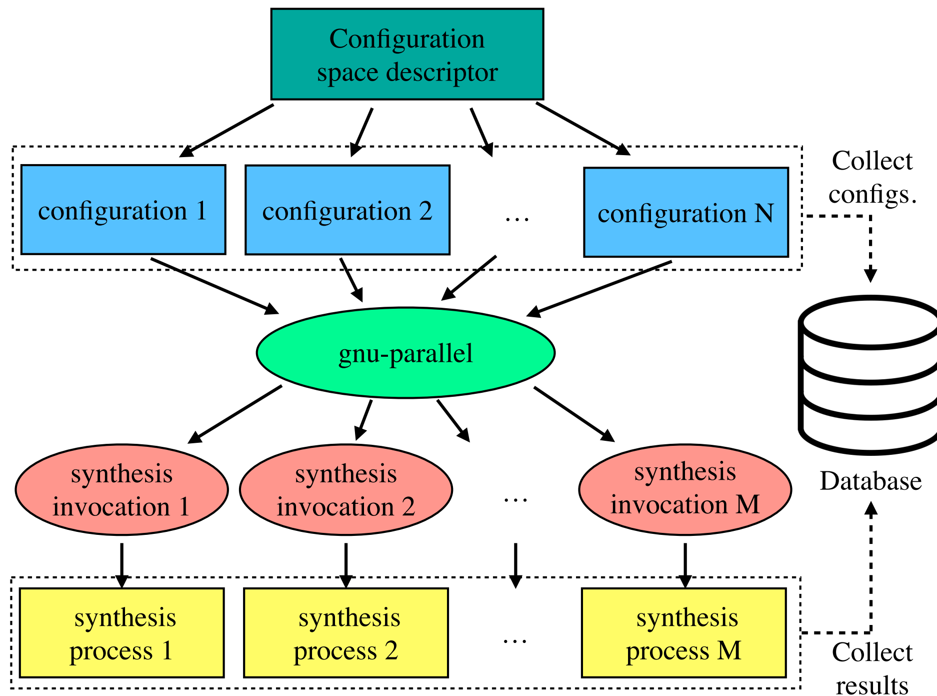


Figure 6.12. DB4HLS framework for parallel execution of HLS processes and data collection for efficient DSEs.

MySQL statements can then be used to retrieve data from the tables in the database and to access the design's implementations and the associated performance and resource results.

To conclude, aim of DB4HLS it to offer an extensive set of DSEs targeting functions from MachSuite [94]. The data collection is made publicly available and will be updated targeting functions from available benchmark suites. In addition, further design spaces can be effectively defined through a novel domain-specific language and a framework to easily contribute novel explorations to DB4HLS.

Chapter 7

Is Deep Learning a Viable Solution?

Various HLS-driven Design Space Exploration (DSE) strategies, which were discussed in Chapter 4, have been proposed to identify, or approximate, the set of Pareto implementations while minimizing the number of synthesis runs. These approaches aim to imitate or learn a model of the HLS tools to pre-estimate the effect of the HLS directives and steer the DSE process to identify Pareto optimal configurations. While mandating very few synthesis runs to identify Pareto solutions, such strategies are not able to accurately estimate performance and costs of the implemented designs.

Against this backdrop, a Machine Learning (ML) approach allows to model the behaviour of HLS directives leveraging prior knowledge–i.e., performance and cost of already existing hardware design–while being agnostic to the number and type of directives considered. However, devising a suitable representation of the problem is not straightforward. Prior attempts to apply Deep Learning (DL) approaches to the DSE problem, discussed in Section 4.2, have not been further explored, and ad-hoc heuristics have outperformed existing approaches.

To address the above-mentioned limitations, I aim at creating a DSE framework able to leverage prior knowledge coming from existing DSE methodologies and from the outcome of past DSEs. This can be performed by extending [39], which showcased promising results while leveraging prior knowledge, and by learning, from input graphs representing an abstraction of the design specifications, a model able to generalise the effect of HLS directives without needing a pre-characterization of the HLS tool.

In the next section, I will describe the work done in this research direction, as well as envisioned future steps.

7.1 Graph-Based Deep Learning for DSE.

The challenge of creating DSE frameworks able to imitate the behaviour of existing HLS tools with traditional ML, and DL models relying on prior knowledge, have not been fully exploited due to the limited expressiveness of vector-based representations of the input data. Graphs, on the other hand, are powerful mathematical abstractions that can be used to capture complex relations and structural information. Control and data flow graphs, in fact, are standard representations of computer programs behaviour. The HLS process itself inherently uses the notion of Control Flow Graphs (CFG) and Data Flow Graphs (DFG) to perform the scheduling, allocation, and binding stages used to generate the HW implementation of a SW functionality.

I aim at exploiting the recent advancements in Graph-Based Deep Learning (GDL) to process and evaluate, in the form of graphs, the effect of candidate HLS directives for a given computational kernel. In particular, my goal is to predict the performance and cost of the hardware implementation of such kernels using Graph Neural Networks (GNNs) [10][137][103].

Starting from input graphs representing an abstraction of the design specifications, I aim at learning a model able to generalise the effect of HLS directives without needing a pre-characterization of the HLS tool. GDL approaches [10] offer state-of-the-art tools to process data represented as graphs; in particular, GNNs are a class of cutting-edge techniques able to solve complex learning tasks, taking full advantage of relational and structural information. By adopting GDL, it is possible to avoid the shortcomings that, up to now, have limited the use of traditional ML approaches in the context of HLS-driven DSE.

The research objective is to adopt GNNs to learn a mapping from graph representations of designs' hierarchical control and computational structures to the performance and cost of the implementations, by using a collection of synthesized designs as a training set (e.g., the dataset of HLS implementations described in Section 6.2). The resulting model is expected to generalise to unseen designs, allowing the identification of the most promising HLS directives and leading to Pareto-optimal implementations of the target design. A key aspect of such approach is the model's ability to build a knowledge discriminating among different classes of inputs. Such differentiation can be further leveraged by including the knowledge of existing DSE models and strategies into the GNNs model.

The proposed framework, shown in Figure 7.1, is beneficial to researchers as it would push the boundaries of the state-of-the-art by offering the possibility to apply Deep Learning (DL) techniques in the context of HLS-driven DSEs.

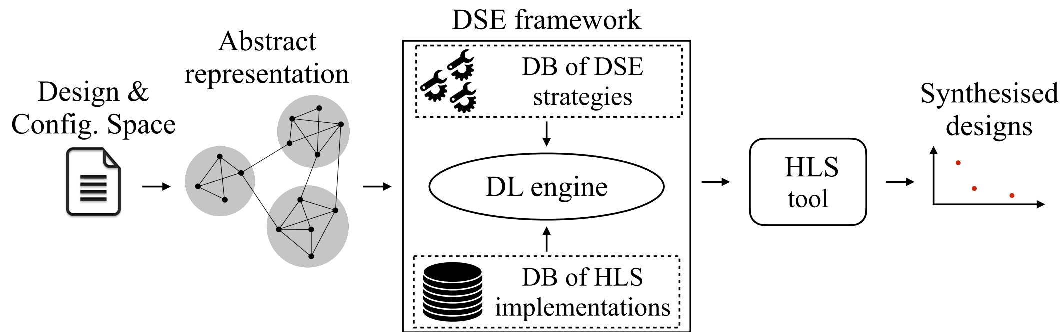


Figure 7.1. Overview of the DL approach for DSEs. A graph representation of a target design is given in input to the DSE framework. The DL engine of the framework leverages prior knowledge from existing DSE strategies and already performed HLS implementations to estimate Pareto-optimal implementations.

7.1.1 Graph Representation of HLS Designs

The first step in the direction of building a GNN model requires the definition of a graph-based representation of an HLS design. To generate such representation I have built a compiler analysis pass, using LLVM, to derive a simplified Control Data Flow Graph (CDFG) of a given design. A CDFG embeds the notion of both CFG and DFG. The first is a model of the flow of control between the basic blocks in a program, while the second models the interaction among data elements in the graphs.

Allen, in [2], defines the CFG as a directed graph $G = (N, E)$, where each node $n \in N$ corresponds to a basic block, and each edge $e = (n_i, n_j) \in E$ represent a possible flow from block n_i to block n_j. DFGs instead are bipartite graphs with actors and links, where actors can be considered similar to transitions in Petri nets, and links similar to places [56]. DFGs are usually used to model computation and to identify data dependencies, e.g., by modeling the interaction between variables and operators. By merging CFGs and DFGs, CDFGs are a powerful representation of both computational and control flow models of programs behaviour.

The DFG information, structured as a network of functional units, registers, multiplexers, and buses, is used to represent the computation taking place in a hardware implementation. This information is used by HLS tools, during the synthesis process, to allocate and bind the resources required by the hardware implementation. On the other hand, CFGs represent the structure of the programs. This information is used by HLS tools to generate the hardware that guarantees the correct temporal end spatial scheduling of the instructions on the

hardware resources.

The simplified CDFG I am proposing also aims at representing these two notions, but from the perspective of the HLS-driven DSE problem, abstracting low-level details and focusing on the high-level characteristics of a design. To this end, the elements of the standard CFG are modified to introduce nodes that can be directly mapped to HLS directive, while DFG actors are reduced to instructions directly interacting with main memory, and links represent use-def changes among actors.

In standard CFGs, each node is a basic block (BB). A BB is defined as a linear sequence of program instructions having one entry point, the first instruction executed, and one exit point, the last instruction executed. BBs may have multiple predecessors and successors, including themselves. The program entry blocks does not have predecessors, and program-terminating blocks do not have successors. Each BB is connected to a different one through a directed edge, and each edge corresponds to a possible transfer of control from a starting block to a target one.

In the formulation I propose, BBs are differentiated according to the type of instructions performed in it. Moreover, their granularity is increased. In fact, a standard BB is divided into sub-blocks according to the functionality of the BB. In particular, we define the following type of blocks:

- function definition block (F): this block identifies the entry point of a function.
- pointer parameter block (P): this block identifies a parameter passed in input to a function as a pointer.
- value parameter block (V): this block identifies a parameter passed in input to a function as a value.
- memory allocation block (A): this block identifies a set of instructions requiring to allocate resources.
- loop block (L): this block identifies the set of instructions defining a loop.
- memory reads block (R): this block identifies instructions reading from main memory.
- memory writes block (W): this block identifies instructions writing into main memory.

Table 7.1. List of information extracted with the LLVM pass for each of the different CDFG blocks.

Block type	Information extracted
F	number of parameters passed to the function, number of instructions executed, function name, number of invocation, function return type.
P	argument type, argument names, associated number of elements.
V	argument type, argument name.
A	variable type, variable name, associated number of elements.
L	number of instructions, nesting depth, number of iterations, presence of loop carried dependency, and stride (in case of loops).
R	name of the variable accessed (only variable accessing to the main memory).
W	name of the variable accessed (only variable accessing to the main memory).
C	name of the invoked function, name of parameters passed to it, number of total instructions executed by the function.
B	number of instructions executed.

- function calls block (C): this block identifies instructions invoking a different function.
- standard block (B): this block identifies instructions performing computations that do not belong to any of the above mentioned categories.

Table 7.1 summarizes the information extracted by the complier pass for each block of the CDFGs.

This information has two purposes. Label information–i.e., function names, pointer names, names of the accessed variable–are used to enhance the CFG with DFG information, tracking the use-def changes. The remaining information–e.g., number of instructions, number of iterations, array size, etc.–are used to generate a one-hot encoded vector associated with each node of the CFG. The vector is used to define a block according to its type and the relevant property of it. The vector includes the fields associated with the potential HLS directives values applied to the design.

Example: Figure 7.2-top shows an example of one-hot encoded vector structure given the different block types and the information extracted with the com-

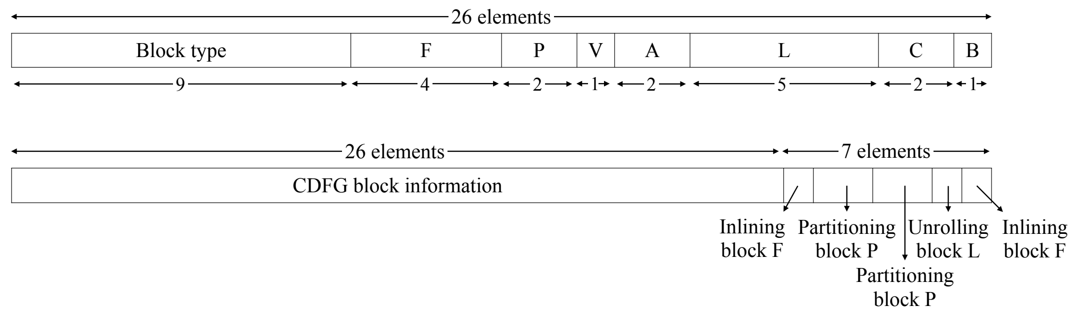


Figure 7.2. Structure of the one-hot encoded vector used to represent the CDFG blocks. (Top) Shows the structure of the vector without the elements specifying the pragma values. (Bottom) Shows the structure of the vector when pragmas are included.

piler pass. The one-hot encoded vector has: 9 elements used to identify the block type, 5 elements for L blocks, 4 elements for the F block, and 2 elements for P, A and C blocks, respectively. Lastly V and B block have only one element. A total of 26 elements are used to represent the information associated to each node of the CDFG. In addition to these elements the fields associated with the HLS directives considered in the DSE are concatenated as shown in figure Figure 7.2-bottom.

For each value, pointer parameter block, and memory allocation blocks, directed edges pointing to memory reads, memory writes, and function calls blocks are added. Instead of tracking all the variables *use-def chain*, we focus only on the ones directly interacting with memory. This is due to the fact that such interaction has very high costs with respect to the cost of local computations, often negligible. This information is added to allow the model to understand the dependencies among the design's subgraphs.

Given Snippet 7.1, its associated CDFG generated by the LLVM pass is shown in Figure 7.3. The figure shows the standard CFG of the function on the left, and the generated simplified CDFG on the right. In the CDFG representation in Figure 7.3-right, is it possible to observe how the structure of the original CFG in Figure 7.3-left is preserved but its granularity of the blocks is increased. For example, the BB6 block in the original CFG is split into 8 different blocks in the CDFG (B5, R1, B6, R2, B7, W1, and B8).

In addition to the control flow information, Figure 7.3-right highlights the dependencies among operations and variables interacting with memory. Directed connections from the arrays in input to the function links them to the read and write operations. These connections are shown as red arrows.

This CDFG and the attributes associated to the different nodes are the input

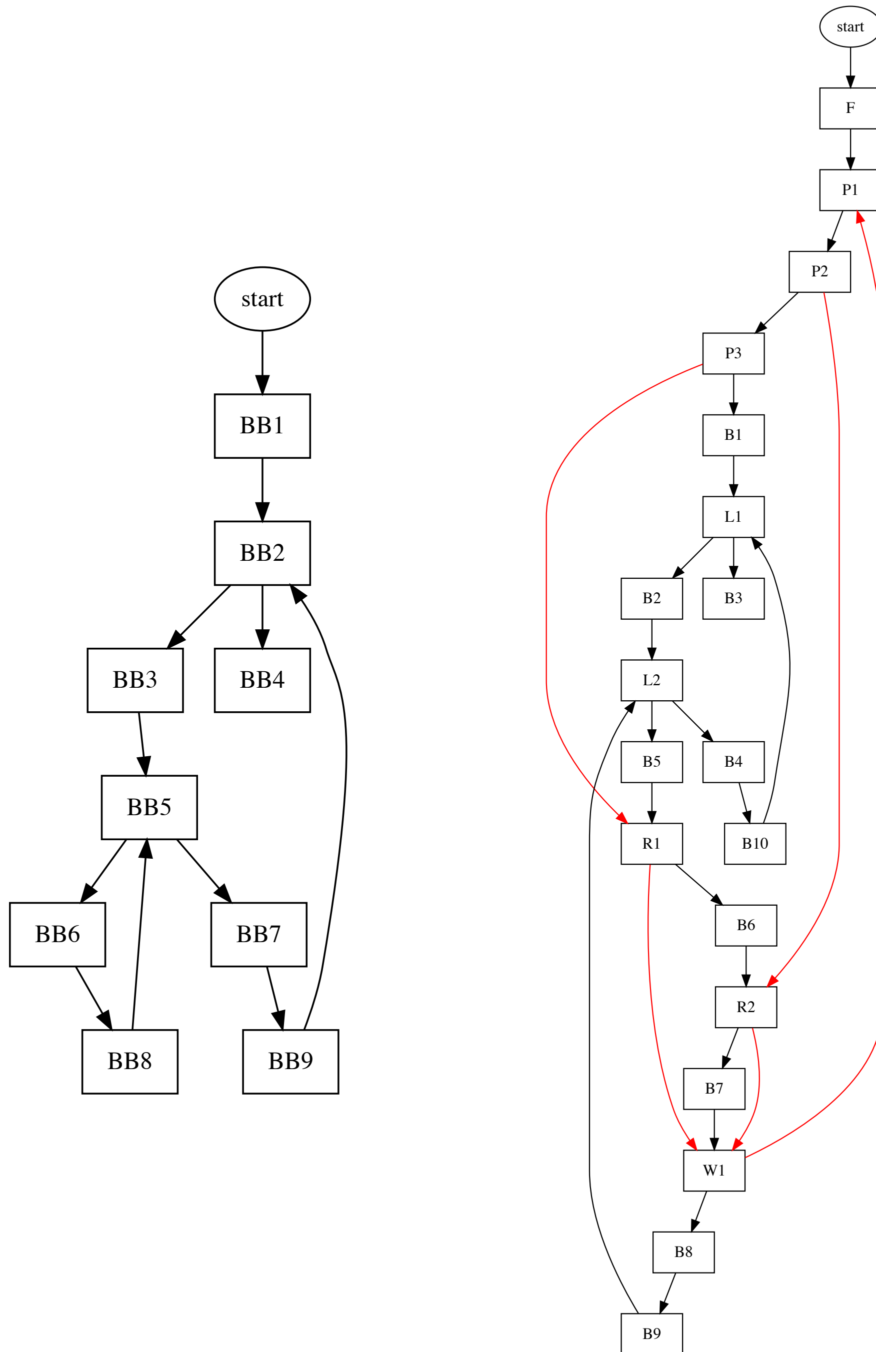


Figure 7.3. (Left) Example of standard CFG of the function `get_delta_matrix_weights3`, in Snippet , generated with LLVM. (Right) Custom version of the CDFG of function `get_delta_matrix_weights3`, generated with a LLVM pass

Listing 7.1. get_delta_matrix_weights3 (source).

```
1  void get_delta_matrix_weights3(
2      double delta_weights3[nodes_per_layer*possible_outputs],
3      double output_difference[possible_outputs],
4      double last_activations[nodes_per_layer]) {
5        int i, j;
6        outer:for(i=0; i<nodes_per_layer; i++) {
7            inner:for(j=0; j<possible_outputs; j++) {
8                delta_weights3[i*possible_outputs+j] =
9                    last_activations[i]*output_difference[j];
10           }
11       }
12 }
```

of the GNN model described in the next section.

7.1.2 Graph Neural Network for HLS

Deep Learning models have been successful at aiding researchers and industry to address many pattern recognition and data mining tasks, such as: object detection [95][96], translation [68, 136], and speech recognition [50]. However, while such approaches have been effective while dealing with Euclidean data, for non-Euclidean data (i.e., graphs), the complexity of the representation has imposed a challenge on existing traditional machine learning algorithms. To cope with graph structured data, traditional machine learning approaches map the input structured data into a simpler representation, e.g., vectors. However, this transformation often results in a loss of important information associated with topological dependency among the different nodes, significantly affecting the final results.

In the context of DSE problems related to hardware design, to the best of my knowledge only one work has tried to apply neural networks to address the exploration task [85]. In this work, a predictive model relying on a neural network and linear regression has been proposed to guide the DSE of a computer architecture. However, apart from this attempt, researchers have focused on different classes of machine learning models and heuristic strategies instead of pursuing deep learning models.

Graph Neural Networks (GNNs) have emerged as a valid tool for applying deep learning on structured data. GNNs are able to perform inference on graph

structures by taking into account arbitrary relationships among the graph nodes. In fact, a graph convolution can be considered as a generalised form of 2D convolution, and an image can be considered as a special case of graph[137]. In different works [103],[58], GNNs have been used to perform inference by mapping node features into categorical and numerical values. Bronstain et al. [10] offer an overview of deep learning methods dealing with non-Euclidean data. Similarly, [45] and [4] discuss the use of GNNs to address the network-embedding problem and the use of GNNs as a building block for learning relational data.

I aim at using a GNN model to learn, given the CDFG representation of an HLS-design, the effect of HLS directives on the resulting implementation. By using Graph Convolutional Neural Networks (GCNN) it is possible to generalise the convolution operation usually performed on grid data to graph-structured data. The CDFG node attributes are aggregated into a vector representation, while the graph structure is represented as a connectivity matrix. By training the GCNN over a set of different optimized designs, I aim to generate an abstract graph representation that aggregates the original node features with the neighbours' ones and predicts the cost and performance for unseen designs. The convolution operation performs a message-passing stage across the graph nodes. The message passing is defined as a message function M_t and a vertex update function U_t running for T time steps. In this phase, the hidden states h_v^t at each node v are updated according to the messages passed by the neighbours $N(v)$ of v, on messages m_v^{t+1} as follows:

$$m_v^{t+1} = \sum_{w \in N(v)} M_t(h_v^t, h_w^t, e_{vw}) \tag{7.1}$$

$$h_v^{t+1} = U_t(h_v^t, m_v^{t+1}) \tag{7.2}$$

Then, a readout phase computes a feature vector for the entire graph using a readout function R. In our case we use a pooling operator to downsample the nodes and generate the abstract graph representation before feeding a regression layer used to predict the cost and performance.

The message function, the vertex update function, the readout function and the regression one are all learnable functions.

Figure 7.4 shows the architecture of the model built to estimate the cost and performance for a given HLS-design and a set of applied pragmas.

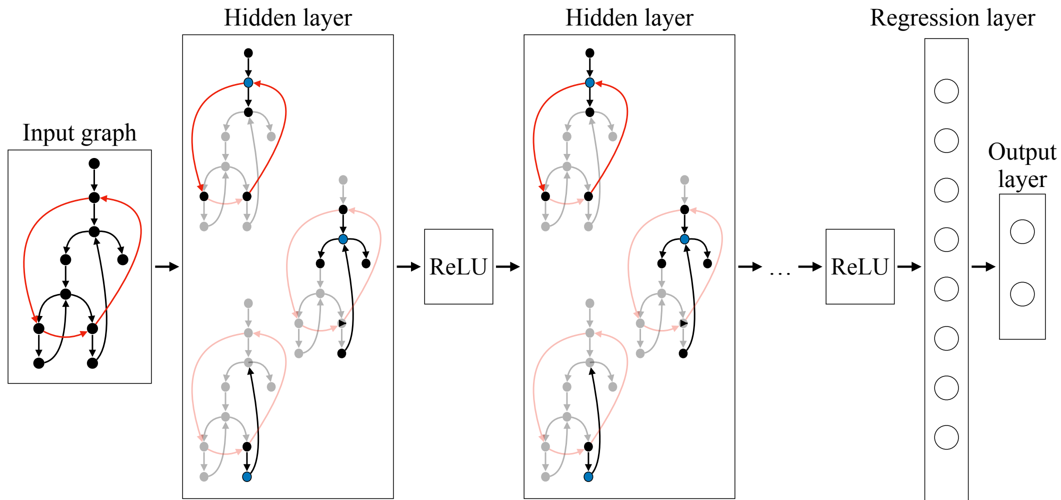


Figure 7.4. Architecture of the GCNN proposed to predict costs and performance of an HLS-design.

7.1.3 Challenges

Working on the GCNN model different criticalities have emerged. The first issue is related to the level of granularity of the input graph. Standard CDFGs used by the HLS tools include information about the variables and operations to synthesise in hardware. The proposed simplified CDFG does not include all this information. It abstracts the concept of operations and required functional units into the concept of basic block. The number of instructions included into a basic block is used as a weight of the computations performed by that block. Therefore, all the operations performed within a basic block are equally weighted, independently from their type. A similar approach has been used in my previous work [39] in which the computational part was not even included in the abstract representation of a design. However, in this approach, the GCNN model aims at estimating the performance and costs of an optimised implementation, while in [39] estimated Pareto implementations were predicted without the need to estimate area and latency.

This last point introduces another challenge. In literature, only a few attempts have tried to learn a model able to generalise the HLS behaviour independently from the input designs and from the directives due to the complexity of the synthesis processes and the large variety of possible inputs. Therefore, trying to learn a model able to precisely predict the area and latency outcome of the HLS tool may be ambitious. An alternative strategy may lead to predicting if an input design, with the associated set of optimisation leads to a Pareto-

optimal implementation. Alternatively, instead of predicting exactly the cost and performance or the Pareto-dominance, an ordering relation among the different configurations can be learned. This last option would avoid the need for an accurate estimation of area and latency. The model's goal would be learning the relational order among the different configurations, disregarding the area and latency estimation error, but preserving the Pareto-dominance relation among the estimated configurations.

The choice of the proper task to learn affects consequently another important aspect: the choice of a proper loss function to evaluate the model. Mean Squared Error (MSE) or Mean Absolute Error (MAE) may be good choices while trying to predict area and latency; however alternative loss functions have to be considered when the Pareto-dominance relation, or the ordering among the solutions, are the goal of the learning task. To this regard, a possible strategy would be considering multi-task learning as a multi-objective optimization problem, as done by Sener et al. in [110], and formulate the problem accordingly.

Moreover, if the learned model fails to produce a high-quality estimation of performance and cost, the model can still be used as a building block for a more sophisticated reinforcement learning approach. The reinforced learning algorithm's goal would be learning a sequence of incremental optimisation steps to navigate the design space and estimate the Pareto elements identifying the fastest way to retrieve the Pareto frontier.

Lastly, as pointed out by Schafer et al., one of the main challenges of supervised learning methods for DSEs is to determine the size of the training data [108]. In fact, while DB4HLS, shown in Section 6.2, offers a large quantity of data for the training, due to the nature of the DSEs, the amount of data associated to different designs may be unbalanced. Thus, while some designs may have tens of thousands of configurations, others may have only hundreds. Therefore, learning a general model from mostly unbalanced training data may be difficult. To cope with this, alternative solutions have been explored, especially by heuristics, aiming to learn how to address the DSE for a specific design instead of learning a general strategy. To this end, a reinforcement learning approach mentioned before may be a valid option. Alternatively, transfer learning strategies can be adopted to start from a general knowledge obtained across a large variety of designs and refine it with a design-specific one.

The set of pieces built until now are small steps of an ambitious project still far from its realisation. This work-in-progress needs further development and the above-mentioned criticalities and possibilities need to be addressed and explored. During the next months, I plan to investigate these aspects, obtain some preliminary results, and consolidate the different steps performed until now.

Chapter 8

Conclusion

HLS has opened many challenges. Among them, the DSE one, subject of this dissertation, has seen many efforts from researchers. However, a general solution to the DSE problem is still far, and DSEs challenges keep evolving.

In fact, the growing market and applicability of ASICs and FPGAs, and the diffusion of heterogeneous hardware architectures, require more and more efficient strategies to generate optimised hardware accelerators able to satisfy performance and efficiency requirements. Machine learning accelerators are becoming ubiquitous. These are adopted in many domains and are applied to different scenarios, targeting high performance architectures and embedded devices. To contrast the growth of their complexity, domain specific accelerators are required both to speed-up the training process to save time and money, and addressing energy saving requirements[118].

To deal with this aspect, HLS methodologies must specialise in order to target domain-specific accelerators and focus on specific aspects of the exploration. For example, the exploration of accelerators dealing with classification tasks need to take into account the drawbacks that some optimisations have on classification performance. By embedding a models of the performance loss in DSE strategies, efficient explorations of the accelerator design space can be achieved.

Moreover, the possibility to reconfigure at runtime FPGAs allows to change the accelerators functionality online to adapt to application scenario changes and external events. HLS plays a key role in this framework, enabling the rapid generation and exploration of accelerator variants to deal with the changing requirements.

On an orthogonal direction, approximate computing enables the generation of RTL components able to trade inaccuracy with performance gain, and area/-power savings. Applications that are tolerant to such inaccuracies can exploit

approximated hardware to reduce costs and improve performance. Including the possibility to explore the use of approximated hardware [102][100][101] in the synthesis during DSEs can lead to more efficient accelerators.

Future DSE strategies must be able to consider all these aspects. Although, including additional objectives and metrics to the exploration task will increase even more the size of the design space and the problem complexity. Therefore, more sophisticated methodologies able to deal with the increasing dimensionality of the problem need to be devised.

I believe the future of DSE relies in the ability of upcoming models to consider all these additional dimensions. DSEs must be able to evaluate optimisations targeting designs at different abstraction levels–i.e., algorithmic level, software implementation level, HLS-directive level, and hardware components one–in order to completely exploit the advantages of the HLS design flow.

Lastly, as a personal remark, I strongly believe that additional effort is required by the research community. Despite the many publications in the field, the hardware community is less open to public releasing the produced artefacts, with respect to the software one. Therefore, a fair evaluation of existing methodologies is a non trivial goal to achieve. This is due to two main factors: there is a lack of datasets to evaluate the existing methodologies, and DSE tools are often not publicly available online. The database proposed in Section 6.2 is a first step in the direction tackling these problems. The electronic design automation community should move in this direction in order to guarantee a fairer evaluation of the existing methodologies and support advancements in this field.

8.1 Contributions during the Ph.D.

Herein follows a list of all the publications I have contributed during my doctoral studies. Publications are listed in temporal order and categorized among journals, conferences, and unpublished works:

Publications in peer-reviewed scientific journals

1. *Leveraging Prior Knowledge for Effective Design-Space Exploration in High-Level Synthesis* [39].
   L. Ferretti, J. Kwon, G. Ansaloni, G Di Guglielmo, L. Carloni, and L. Pozzi. Presented at ESWEEK 20-25 September 2020. To be published in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD) 2020.

This work is the result of an internship at Columbia University in New York. I have proposed a novel design space exploration strategy investigating, for the first time, the feasibility of effectively harnessing the knowledge from past synthesis outcomes to guide the optimization of new designs. The proposed approach, by leveraging prior knowledge from a database of existing design space explorations, dramatically reduces the number of syntheses required by the explorations while retrieving a close approximation of the Pareto frontier.

2. *RegionSeeker: Automatically Identifying and Selecting Accelerators From Application Source Code* [143].
   G. Zacharopoulos, L. Ferretti, E. Giaquinta, G. Ansaloni, L. Pozzi.
   IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD) 2018.

   In this work, a fully automated identification and selection methodology of hardware accelerators, from application source code, is presented. The proposed methodology advances the state-of-the-art and provides efficient solutions by offering an up to 4.6x speedup and, on average, approximately 30% higher speedup compared to state-of-the-art identification techniques.

3. *Cluster-Based Heuristic for High Level Synthesis Design Space Exploration* [36].
   L. Ferretti, G. Ansaloni, L. Pozzi.
   IEEE Transactions on Emerging Topics in Computing (TETC) 2018.

   In this work, I have presented a design space exploration methodology able to efficiently navigate the space of possible high level synthesis optimizations. The proposed methodology uses clustering to divide the problem in subproblem and perform different local explorations reducing the complexity of the task. This approach was shown to outperform state-of-the-art alternatives.

Peer-reviewed conference proceedings

1. *Compiler-Assisted Selection of Hardware Acceleration Candidates from Application Source Code* [142].
   G. Zacharopoulos, L. Ferretti, G. Ansaloni, G. Di Guglielmo, L. Carloni and L. Pozzi.
   IEEE 37th International Conference on Computer Design (ICCD) 2019.

   This research paper extends the RegionSeeker framework accepted in the TCAD journal as listed above. The proposed method performs automatic

identification and selection of hardware accelerators while taking into account the overhead due to memory communication and the platform constraints. The evaluation of the methodology is performed for a complex application such as H.264 decoder.

2. *Tailoring SVM Inference for Resource-Efficient ECG-Based Epilepsy Monitors* [38].
L. Ferretti, G. Ansaloni, L. Pozzi, A. Aminifar, D. Atienza, L. Cammoun, P. Ryvlin.
Design, Automation & Test in Europe Conference (DATE) 2019.

   I have conceived and explored multiple optimizations by tailoring Support Vector Machine (SVM) inference engines devoted to the detection of epileptic seizures from ECG-derived features. The combination of the different optimization strategies resulted in 12.5X and 16X gains in energy and area, respectively, with a negligible loss, 3.2% in classification performance.

3. *Lattice-Traversing Design Space Exploration for High Level Synthesis* [37].
L. Ferretti, G. Ansaloni, L. Pozzi.
IEEE 36th International Conference on Computer Design (ICCD) 2018.

   In this work, I have proposed a design space exploration methodology based on the observation that Pareto-implementations share a low variance among their configurations. Therefore, I have devised a strategy selecting the HLS directives that minimise the variance of new candidate solutions, with respect to the best-performing ones that have already been visited. This approach results in close approximations of the real Pareto frontier while requiring a lower workload and fewer synthesis runs with respect to existing approaches.

Accepted but not yet printed

1. *Adaptive Iterated Local Search with Random Restarts for the Balanced Travelling Salesman Problem.*
J. Pierotti, L. Ferretti, L. Pozzi, J. Thereisa van Essen.
To be published in Advances in Intelligent Systems and Computing.

   In this work, an adaptive variant of the iterated local search metaheuristic featuring random restart for the Balanced Travelling Salesmen Problem is presented. This algorithm introduces an uneven reward-and-punishment rule to enable a fast response to dynamic changes during the search for new solutions. The proposed approach has obtained notable results, achieving the 5th position at the MESS 2018 metaheuristic competition.

Unpublished work

1. *DB4HLS: A Database of High-Level SynthesisDesign Space Explorations.*
L. Ferretti, J. Kwon, G. Ansaloni, G Di Guglielmo, L. Carloni, and L. Pozzi.
Will be submitted to IEEE Embedded Systems Letters (ESL).

   This work extendes the database created for the work "Leveraging Prior Knowledge for Effective Design-Space Exploration in High-Level Synthesis" accepted at ESWEEK and listed above. I have created a framework to efficiently perform and collect design space explorations with HLS. The open structure of DB4HLS allows the incremental integration of new explorations offering a valuable tool for the research community investigating automated strategies for the optimization of HLS-based hardware designs.

8.2 What's next?

Herein, I discuss the future plans for my research. Some of the future works have already been presented in Chapter 7, where the possibility to use DL to tackle DSEs problems, and the challenges it presents have been discussed.

However, I am considering other possible directions aiming at expanding the works done until now and address the challenges discussed previously in this chapter. Following the directions traced by the work leveraging on the previous knowledge (Chapter 6), I aim to:

- Extend the work by moving from a separate representation of specifications and configuration space to a unified representation of those in form of a graph–i.e., the graph representation discussed in Chapter 7. This allow the use of graph similarities to identify the proper sources and to exploit benefits from microkernel characterisations as done in a recent work [132]. Moreover, searching for subgraph similarities would allow a better coverage of large applications. Besides, I aim at creating a model to improve the existing inference stage in order to offer a better coverage of the design space for which no knowledge is available–i.e., differences among the source and target set of directive values. Lastly, I would like to investigate the possibility to combine the knowledge from different DSE problems in case of multiple similar sources.

- Release the database of DSEs described in Section 6.2 and expand the database with other benchmark suites such as CHStone [47] and Rosetta [148].

- Devise a DSE framework for classification task problems that explores a design space including different ML models, and optimizations targeting the model at different levels of abstraction. Given cost and performance requirements, the framework holistically identifies a choice of the proper classification algorithm, its software implementation, HLS optimisations and hardware components able to satisfy designer requirements.
- Embed the knowledge of existing models into a comprehensive framework able to choose the best exploration strategy according to the input problem characteristics. Such framework would be able to merge the capabilities of the existing independent ones to solve more complicated tasks.

Bibliography

[1] Ahmad, I., Dhodhi, M. K. and Hielscher, F. H. [1994]. Design-Space Exploration for High-Level Synthesis, *Proceeding of 13th IEEE Annual International Phoenix Conference on Computers and Communications*.

[2] Allen, F. E. [1970]. Control flow analysis, *ACM Sigplan Notices* **5**(7): 1–19.

[3] Barbacci, M. R. [1981]. Instruction set processor specifications (isps): The notation and its applications, *IEEE Transactions on Computers* **100**(1): 24–40.

[4] Battaglia, P. W., Hamrick, J. B., Bapst, V., Sanchez-Gonzalez, A., Zambaldi, V., Malinowski, M., Tacchetti, A., Raposo, D., Santoro, A., Faulkner, R. et al. [2018]. Relational inductive biases, deep learning, and graph networks, *arXiv preprint arXiv:1806.01261* .

[5] Beltrame, G., Fossati, L. and Sciuto, D. [2010]. Decision-Theoretic Design Space Exploration of Multiprocessor Platforms, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **29**(7): 1083–1095.

[6] Bilavarn, S., Gogniat, G., Philippe, J.-L. and Bossuet, L. [2006]. Design Space Pruning Through Early Estimations of Area/Delay Tradeoffs for FPGA Implementations, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **25**(10): 1950–1968.

[7] Blu [2020]. Bluespec, `https://bluespec.com/technology/`.

[8] Borkar, S. and Chien, A. A. [2011]. The future of microprocessors, *Communications of the ACM* **54**(5): 67–77.

[9] Breiman, L. [2001]. Random Forests, *Machine learning* **45**(1): 5–32.

[10] Bronstein, M. M., Bruna, J., LeCun, Y., Szlam, A. and Vandergheynst, P. [2017]. Geometric deep learning: going beyond euclidean data, *IEEE Signal Processing Magazine* **34**(4): 18–42.

[11] Camposano, R. [1988]. Design process model in the yorktown silicon compiler, *25th ACM/IEEE, Design Automation Conference. Proceedings 1988.*, IEEE, pp. 489–494.

[12] Canis et al., A. [2013]. LegUp: An open-source High-level Synthesis tool for FPGA-based processor/accelerator systems, *ACM Transactions on Embedded Computing Systems (TECS)* **13**(2): 1–27.

[13] Cat [2020]. Catapult High-Level Synthesis, `https://www.mentor.com/hls-lp/catapult-high-level-synthesis/c-systemc-hls`.

[14] Chi, Y., Cong, J., Wei, P. and Zhou, P. [2018]. Soda: stencil with optimized dataflow architecture, *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, IEEE, pp. 1–8.

[15] Choi, Y.-k., Chi, Y., Wang, J. and Cong, J. [2020]. Flash: Fast, parallel, and accurate simulator for hls, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* .

[16] Choi, Y.-k. and Cong, J. [2018]. Hls-based optimization and design space exploration for applications with variable loop bounds, *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, IEEE, pp. 1–8.

[17] Clark, N., Zhong, H., Fan, K., Mahlke, S., Flautner, K. and Nieuwenhove, K. [2004]. Optimode: Programmable accelerator engines through retargetable customization, *Hot Chips*, Vol. 16.

[18] Cong, J., Liu, B., Neuendorffer, S., Noguera, J., Vissers, K. and Zhang, Z. [2011]. High-level synthesis for fpgas: From prototyping to deployment, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **30**(4): 473–491.

[19] Cong, J. and Wang, J. [2018]. Polysa: polyhedral-based systolic array autocompilation, *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, IEEE, pp. 1–8.

[20] Cong, J., Wei, P., Yu, C. H. and Zhang, P. [2018]. Automated accelerator generation and optimization with composable, parallel and pipeline architecture, *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, IEEE, pp. 1–6.

[21] Cong, J., Zhang, P. and Zou, Y. [2011]. Combined loop transformation and hierarchy allocation for data reuse optimization, *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 185–192.

[22] Cong, J., Zhang, P. and Zou, Y. [2012]. Optimizing memory hierarchy allocation with loop transformations for high-level synthesis, *Proceedings of the 49th annual design automation conference*, pp. 1233–1238.

[23] Coussy, P., Gajski, D. D., Meredith, M. and Takach, A. [2009]. An introduction to high-level synthesis, *IEEE Design & Test of Computers* **26**(4): 8–17.

[24] Cyb [2020]. CyberWorkBench High Level Synthesis from C/C++/SystemC to ASIC/FPGA, `https://www.nec.com/en/global/prod/cwb/index.html`.

[25] Dai, S., Zhou, Y., Zhang, H., Ustun, E., Young, E. F. and Zhang, Z. [2018]. Fast and accurate estimation of quality of results in high-level synthesis with machine learning, pp. 129–132.

[26] De Man, H., Rabaey, J., Six, P. and Claesen, L. [1986]. Cathedral-ii: A silicon compiler for digital signal processing, *IEEE Design & Test of Computers* **3**(6): 13–25.

[27] De Micheli, G., Ku, D., Mailhot, F. and Truong, T. [1990]. The olympus synthesis system, *IEEE Design & Test of Computers* **7**(5): 37–53.

[28] Dennard, R. H., Gaensslen, F. H., Yu, H.-N., Rideout, V. L., Bassous, E. and LeBlanc, A. R. [1974]. Design of ion-implanted mosfet's with very small physical dimensions, *IEEE Journal of Solid-State Circuits* **9**(5): 256–268.

[29] Deshwal, A., Jayakodi, N. K., Joardar, B. K., Doppa, J. R. and Pande, P. P. [2019]. MOOS: A Multi-Objective Design Space Exploration and Optimization Framework for NoC Enabled Manycore Systems, *ACM Trans. Embed. Comput. Syst.* .

[30] Director, S., Parker, A., Siewiorek, D. and Thomas, D. [1981]. A design methodology and computer aids for digital vlsi systems, *IEEE Transactions on Circuits and Systems* **28**(7): 634–645.

[31] Doppa, J. R., Rosca, J. and Bogdan, P. [2019]. Autonomous design space exploration of computing systems for sustainability: Opportunities and challenges, *IEEE Design Test* **36**(5): 35–43.

[32] Edwards, S. A. [2006]. The challenges of synthesizing hardware from c-like languages, *IEEE Design & Test of Computers* **23**(5): 375–386.

[33] eet [2020]. Subaru Replaces ASICs with Xilinx FPGA for Latest Vision-Based ADAS, `https://www.eetimes.com/subaru-replaces-asics-with-xilinx-fpga-for-latest-vision-based-adas/`.

[34] Elliott, J. P. [1999]. *Understanding behavioral synthesis: a practical guide to high-level design*, Springer Science & Business Media.

[35] Esmaeilzadeh, H., Blem, E., St. Amant, R., Sankaralingam, K. and Burger, D. [2011]. Dark silicon and the end of multicore scaling, *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pp. 365–376.

[36] Ferretti, L., Ansaloni, G. and Pozzi, L. [2018a]. Cluster-based heuristic for high level synthesis design space exploration, *IEEE Transactions on Emerging Topics in Computing* (99): 1–9.

[37] Ferretti, L., Ansaloni, G. and Pozzi, L. [2018b]. Lattice-traversing design space exploration for high level synthesis, *Proceedings of the International Conference on Computer Design*, pp. 210–217.

[38] Ferretti, L., Ansaloni, G., Pozzi, L., Aminifar, A., Atienza, D., Cammoun, L. and Ryvlin, P. [2019]. Tailoring svm inference for resource-efficient ecg-based epilepsy monitors, *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, pp. 948–951.

[39] Ferretti, L., Jihye, K., Ansaloni, G., Di Guglielmo, G., Carloni, L. and Pozzi, L. [2020]. Leveraging prior knowledge for effective design-space exploration in high level synthesis.

[40] Fornaciari, W., Sciuto, D., Silvano, C. and Zaccaria, V. [2002]. A Sensitivity-Based Design Space Exploration Methodology for Embedded Systems, *Design Automation for Embedded Systems* **7**(1): 7–33.

[41] Gajski, D. D., Dutt, N. D., Wu, A. C. and Lin, S. Y. [2012]. *HighâĂŤLevel Synthesis: Introduction to Chip and System Design*, Springer Science & Business Media.

[42] Gajski, D. D., Zhu, J., Dömer, R., Gerstlauer, A. and Zhao, S. [2012]. *SpecC: Specification language and methodology*, Springer Science & Business Media.

[43] Geiger, R. L., Allen, P. E. and Strader, N. R. [1990]. Vlsi design techniques for analog and digital circuits.

[44] Granacki, J., Knapp, D. and Parker, A. [1985]. The adam advanced design automation system: overview, planner and natural language interface, *22nd ACM/IEEE Design Automation Conference*, IEEE, pp. 727–730.

[45] Hamilton, W., Ying, Z. and Leskovec, J. [2017]. Inductive representation learning on large graphs, *Advances in neural information processing systems*, pp. 1024–1034.

[46] Han [2020]. Handel-C Synthesis Methodology, `https://www.mentor.com/products/fpga/handel-c/`.

[47] Hara, Y., Tomiyama, H., Honda, S., Takada, H. and Ishii, K. [2008]. Chstone: A benchmark program suite for practical c-based high-level synthesis, *2008 IEEE International Symposium on Circuits and Systems*, IEEE, pp. 1192–1195.

[48] Haubelt, C. and Teich, J. [2003]. Accelerating Design Space Exploration Using Pareto-Front Arithmetics [SoC design], *Proceedings of the Asia and South Pacific Design Automation Conference*, IEEE, pp. 525–531.

[49] Hemani, A., Karlsson, B., Fredriksson, M., Nordqvist, K. and Fjellborg, B. [1994]. Application of high-level synthesis in an industrial project, *Proceedings of 7th International Conference on VLSI Design*, IEEE, pp. 5–10.

[50] Hinton, G., Deng, L., Yu, D., Dahl, G. E., Mohamed, A.-r., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T. N. et al. [2012]. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups, *IEEE Signal processing magazine* **29**(6): 82–97.

[51] Holzer, M., Knerr, B. and Rupp, M. [2007]. Design Space Exploration with Evolutionary Multi-Objective Optimisation, *International Symposium on Industrial Embedded Systems*, IEEE, pp. 126–133.

[52] Int [2011]. System Drivers 2011, http://www.itrs.net/.

[53] Int [2020]. Intel High Level Synthesis Compiler, https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html.

[54] Jolliffe, I. T. [1986]. Principal Component Analysis and Factor Analysis, *Principal Component Analysis*, Springer, pp. 115–128.

[55] Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A. et al. [2017]. In-datacenter performance analysis of a tensor processing unit, *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 1–12.

[56] Kavi, K. M., Buckles, B. P. and Bhat, U. N. [1986]. A formal definition of data flow graph models, *IEEE Transactions on computers* (11): 940–948.

[57] Kish, L. B. [2002]. End of moore's law: thermal (noise) death of integration in micro and nano electronics, *Physics Letters A* **305**(3-4): 144–149.

[58] Klicpera, J., Bojchevski, A. and Günnemann, S. [2018]. Predict then propagate: Graph neural networks meet personalized pagerank, *arXiv preprint arXiv:1810.05997* .

[59] Knapp, D. W. [1996]. *Behavioral synthesis: digital system design using the synopsys behavioral compiler*, Prentice-Hall, Inc.

[60] Ku, D. C. and De Micheli, G. [1988]. Hardware c-a language for hardware design, *Technical report*, STANFORD UNIV CA COMPUTER SYSTEMS LAB.

[61] Lahti, S., Sjövall, P., Vanne, J. and Hämäläinen, T. D. [2018]. Are we there yet? a study on the state of high-level synthesis, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **38**(5): 898–911.

[62] Lattner, C. and Adve, V. [2004]. LLVM: A compilation framework for lifelong program analysis & transformation, *Proceedings of the international symposium on Code generation and optimization*, p. 75.

[63] Liu, G., Tan, M., Dai, S., Zhao, R. and Zhang, Z. [2017]. Architecture and synthesis for area-efficient pipelining of irregular loop nests, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **36**(11): 1817–1830.

[64] Liu, H.-Y. and Carloni, L. P. [2013a]. On Learning-Based Methods for Design-Space Exploration with High-Level Synthesis, *Proceedings of the 50th Design Automation Conference*, IEEE, pp. 1–7.

[65] Liu, H.-Y. and Carloni, L. P. [2013b]. On learning-based methods for design-space exploration with high-level synthesis, *Proceedings of the 50th Design Automation Conference*, pp. 1–6.

[66] Liu, H.-Y., Petracca, M. and Carloni, L. P. [2012]. Compositional system-level design exploration with planning of high-level synthesis, *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 641–646.

[67] Liu, S., Lau, F. and Schafer, B. C. [2019]. Accelerating FPGA prototyping through predictive model-based HLS design space exploration, p. 97.

[68] Luong, M.-T., Pham, H. and Manning, C. D. [2015]. Effective approaches to attention-based neural machine translation, *arXiv preprint arXiv:1508.04025* .

[69] Mann, C. C. [2000]. The end of moore's law?, *Technology Review* **103**(3): 42–42.

[70] Mann, H. B. and Whitney, D. R. [1947]. On a test of whether one of two random variables is stochastically larger than the other, *The annals of mathematical statistics* pp. 50–60.

[71] Mariani, G., Palermo, G., Zaccaria, V. and Silvano, C. [2012]. OSCAR: An Optimization Methodology Exploiting Spatial Correlation in Multicore Design Spaces, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **31**(5): 740–753.

[72] Martin, G. and Smith, G. [2009]. High-level synthesis: Past, present, and future, **26**(4): 18–25.

[73] Martins, L. G., Nobre, R., Cardoso, J. M., Delbem, A. C. and Marques, E. [2016]. Clustering-based selection for the exploration of compiler optimization sequences, **13**(1): 8.

[74] Marwedel, P. [1984]. The mimola design system: Tools for the design of digital processors, *21st Design Automation Conference Proceedings*, IEEE, pp. 587–593.

[75] Mat [n.d.]. Statistics and machine learning toolbox, https://ch.mathworks.com/help/stats/.

[76] Meng, P., Althoff, A., Gautier, Q. and Kastner, R. [2016]. Adaptive threshold non-pareto elimination: Re-thinking machine learning for system level design space exploration on fpgas, *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, pp. 918–923.

[77] Moo [n.d.]. Moore's law is dead. now what?, https://www.technologyreview.com/2016/05/13/245938/moores-law-is-dead-now-what/.

[78] Moore, G. E. et al. [1965]. Cramming more components onto integrated circuits.

[79] Mukherjee, R., Ghosh, P., Dasgupta, P. and Pal, A. [2013]. A multi-objective perspective for operator scheduling using fine-grained dvs architecture, *arXiv preprint arXiv:1303.1645* .

[80] Mukherjee, R., Ghosh, P., Kumar, N. S., Dasgupta, P. and Pal, A. [2012]. Multi-objective low-power cdfg scheduling using fine-grained dvs architecture in distributed framework, *2012 International Symposium on Electronic System Design (ISED)*, IEEE, pp. 267–271.

[81] Mukherjee, R., Ghosh, P. and Pal, A. [2012]. Hotspot minimization using fine-grained dvs architecture at 90 nm technology, *2012 Asia Pacific Conference on Postgraduate Research in Microelectronics and Electronics*, IEEE, pp. 13–18.

[82] Nane, R., Sima, V.-M., Pilato, C., Choi, J., Fort, B., Canis, A., Chen, Y. T., Hsiao, H., Brown, S., Ferrandi, F. et al. [2015]. A survey and evaluation of fpga high-level synthesis tools, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **35**(10): 1591–1604.

[83] Olukotun, K. and Hammond, L. [2005]. The future of microprocessors, *Queue* **3**(7): 26–29.

[84] Orailoglu, A. and Gajski, D. D. [1986]. Flow graph representation, *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, pp. 503–509.

[85] Ozisikyilmaz, B., Memik, G. and Choudhary, A. [2008]. Efficient System Design Space Exploration Using Machine Learning Techniques, *Proceedings of the 45th Design Automation Conference*, ACM, pp. 966–969.

[86] Palermo, G., Silvano, C. and Zaccaria, V. [2009]. ReSPIR: a Response Surface-Based Pareto Iterative Refinement for Application-Specific Design Space Exploration, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **28**(12): 1816–1829.

[87] Palesi, M. and Givargis, T. [2002]. Multi-Objective Design Space Exploration Using Genetic Algorithms, *Proceedings of the 10th International Workshop on Hardware/Software Codesign*, pp. 67–72.

[88] Park, N. and Parker, A. [1986]. Sehwa: A program for synthesis of pipelines, *23rd ACM/IEEE Design Automation Conference*, IEEE, pp. 454–460.

[89] Paterson, M. and Dančík, V. [1994]. Longest common subsequences, *International Symposium on Mathematical Foundations of Computer Science*, pp. 127–142.

[90] Paulin, P. G., Knight, J. P. and Girczyc, E. F. [1986]. Hal: a multi-paradigm approach to automatic data path synthesis, *23rd ACM/IEEE Design Automation Conference*, IEEE, pp. 263–270.

[91] Piccolboni, L., Mantovani, P., Guglielmo, G. D. and Carloni, L. P. [2017]. Cosmos: Coordination of high-level synthesis and memory optimization for hardware accelerators, *ACM Transactions on Embedded Computing Systems (TECS)* **16**(5s): 1–22.

[92] Pilato, C. and Ferrandi, F. [2013]. Bambu: A modular framework for the high level synthesis of memory-intensive applications, *2013 23rd International Conference on Field programmable Logic and Applications*, IEEE, pp. 1–4.

[93] Putnam, A., Caulfield, A. M., Chung, E. S., Chiou, D., Constantinides, K., Demme, J., Esmaeilzadeh, H., Fowers, J., Gopal, G. P., Gray, J. et al. [2014]. A reconfigurable fabric for accelerating large-scale datacenter services, *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, IEEE, pp. 13–24.

[94] Reagen, B., Adolf, R., Shao, Y. S., Wei, G.-Y. and Brooks, D. [2014]. Machsuite: Benchmarks for accelerator design and customized architectures, *Proceedings of the IEEE International Symposium on Workload Characterization*, pp. 110–119.

[95] Redmon, J., Divvala, S., Girshick, R. and Farhadi, A. [2016]. You only look once: Unified, real-time object detection, *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 779–788.

[96] Ren, S., He, K., Girshick, R. and Sun, J. [2015]. Faster r-cnn: Towards real-time object detection with region proposal networks, *Advances in neural information processing systems*, pp. 91–99.

[97] Reyes Fernandez de Bulnes, D., Maldonado, Y. and Trujillo, L. [2020]. Development of multiobjective high-level synthesis for fpgas, *Scientific Programming* **2020**.

[98] ROC [n.d.]. Roccc, http://roccc.cs.ucr.edu/, `http://roccc.cs.ucr.edu/`.

[99] Sanguinetti, J. [2006]. A different view: Hardware synthesis from systemc is a maturing technology, *IEEE Design & Test of Computers* **23**(5): 387–387.

[100] Scarabottolo, I., Ansaloni, G., Constantinides, G. A. and Pozzi, L. [2019]. Partition and propagate: An error derivation algorithm for the design of approximate circuits, *2019 56th ACM/IEEE Design Automation Conference (DAC)*, IEEE, pp. 1–6.

[101] Scarabottolo, I., Ansaloni, G., Constantinides, G. A., Pozzi, L. and Reda, S. [2020]. Approximate logic synthesis: A survey, *Proceedings of the IEEE* .

[102] Scarabottolo, I., Ansaloni, G. and Pozzi, L. [2018]. Circuit carving: A methodology for the design of approximate hardware, *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 545–550.

[103] Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M. and Monfardini, G. [2008]. The graph neural network model, *IEEE Transactions on Neural Networks* **20**(1): 61–80.

[104] Schafer, B. C. [2015]. Probabilistic multiknob high-level synthesis design space exploration acceleration, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **35**(3): 394–406.

[105] Schafer, B. C., Takenaka, T. and Wakabayashi, K. [2009]. Adaptive Simulated Annealer for High Level Synthesis Design Space Exploration, *International Symposium on VLSI Design, Automation and Test*, IEEE, pp. 106–109.

[106] Schafer, B. C. and Wakabayashi, K. [2012a]. Divide and Conquer High-Level Synthesis Design Space Exploration, *ACM Transactions on Design Automation of Electronic Systems (TODAES)* **17**(3): 29.

[107] Schafer, B. C. and Wakabayashi, K. [2012b]. Divide and conquer high-level synthesis design space exploration, *ACM Transactions on Design Automation of Electronic Systems (TODAES)* **17**(3): 1–19.

[108] Schafer, B. C. and Wang, Z. [2019]. High-level synthesis design space exploration: Past, present and future, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* pp. 1–1.

[109] SDA [2020]. SDAccel: Enabling Hardware-Accelerated Software, `https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html`.

[110] Sener, O. and Koltun, V. [2018]. Multi-task learning as multi-objective optimization, *Advances in Neural Information Processing Systems*, pp. 527–538.

[111] Shao, Y. S., Reagen, B., Wei, G.-Y. and Brooks, D. [2014]. Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures, *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, IEEE, pp. 97–108.

[112] Shathanaa, R. and Ramasubramanian, N. [2018]. Design space exploration for architectural synthesis—a survey, *in* P. K. Sa, S. Bakshi, I. K. Hatzilygeroudis and M. N. Sahoo (eds), *Recent Findings in Intelligent Computing Techniques*, pp. 519–527.

[113] Silvano, C., Fornaciari, W., Palermo, G., Zaccaria, V., Castro, F., Martinez, M., Bocchio, S., Zafalon, R., Avasare, P., Vanmeerbeeck, G., Ykman-Couvreur, C., Wouters, M., Kavka, C., Onesti, L., Turco, A., Bondik, U., Mariani, G., Posadas, H., Villar, E., Wu, C., Dongrui, F., Hao, Z. and Shibin, T. [2010]. MULTICUBE: Multi-Objective Design Space Exploration of Multi-core Architectures, *IEEE Computer Society Annual Symposium on VLSI*, pp. 488–493.

[114] Silvano et.al, C. [2010]. MULTICUBE: Multi-Objective Design Space Exploration of Multi-core Architectures, *IEEE Computer Society Annual Symposium on VLSI*, pp. 488–493.

[115] So, B., Hall, M. W. and Diniz, P. C. [2002]. A Compiler Approach to Fast Hardware Design Space Exploration in FPGA-based Systems, *Proceedings of the ACM SIGPLAN'02 Conference on Programming Language Design and Implementation* **37**(5).

[116] SPA [n.d.]. SPARK, http://mesl.ucsd.edu/spark/, `http://mesl.ucsd.edu/spark/`.

[117] Str [2020]. Stratus High Level Synthesis, `https://www.cadence.com/ko_KR/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html`.

[118] Strubell, E., Ganesh, A. and McCallum, A. [2019]. Energy and policy considerations for deep learning in nlp, *arXiv preprint arXiv:1906.02243* .

[119] Syn [2020]. Synphony High Level Synthesis, `https://news.synopsys.com/index.php?s=20295&item=123096`.

[120] Sys [2011]. SystemC, `https://standards.ieee.org/standard/1666-2011.html`.

[121] Tan, M., Liu, G., Zhao, R., Dai, S. and Zhang, Z. [2015]. Elasticflow: A complexity-effective approach for pipelining irregular loop nests, *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, IEEE, pp. 78–85.

[122] Tange, O. [2011]. Gnu parallel - the command-line power tool.

[123] Theis, T. N. and Wong, H. . P. [2017]. The end of moore's law: A new beginning for information technology, *Computing in Science Engineering* **19**(2): 41–50.

[124] Trimberger, S. M. S. [2018]. Three ages of fpgas: A retrospective on the first thirty years of fpga technology: This paper reflects on how moore's law has driven the design of fpgas through three epochs: the age of invention, the age of expansion, and the age of accumulation, *IEEE Solid-State Circuits Magazine* **10**(2): 16–29.

[125] Tripp, J. L., Gokhale, M. B. and Peterson, K. D. [2007]. Trident: From high-level language to hardware circuitry, *Computer* **40**(3): 28–37.

[126] Viv [2020]. Vivado High-Level Synthesis, `https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html`.

[127] Wagner, R. A. and Fischer, M. J. [1974]. The string-to-string correction problem, *Journal of the ACM (JACM)* **21**(1): 168–173.

[128] Wakabayashi, K. [2004]. C-based behavioral synthesis and verification.

[129] Wang, E., Davis, J. J., Zhao, R., Ng, H.-C., Niu, X., Luk, W., Cheung, P. Y. and Constantinides, G. A. [2019]. Deep neural network approximation for custom hardware: Where we've been, where we're going, *ACM Computing Surveys (CSUR)* **52**(2): 1–39.

[130] Wang, S., Liang, Y. and Zhang, W. [2017]. Flexcl: An analytical performance model for opencl workloads on flexible fpgas, *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6.

[131] Wang, Y., Li, P. and Cong, J. [2014]. Theory and algorithm for generalized memory partitioning in high-level synthesis, *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, pp. 199–208.

[132] Wang, Z., Chen, J. and Schafer, B. C. [2020]. Efficient and robust high-level synthesis design space exploration through offline micro-kernels pre-characterization, *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 145–150.

[133] Ward Jr, J. H. [1963]. Hierarchical Grouping to Optimize an Objective Function, *Journal of the American Statistical Association* **58**(301): 236–244.

[134] While, L., Bradstreet, L. and Barone, L. [2011]. A fast way of calculating exact hypervolumes, *IEEE Transactions on Evolutionary Computation* **16**(1): 86–95.

[135] Wu, F., Xu, N., Yu, J., Zheng, F. and Bian, J. [2009]. Exploiting power-area tradeoffs in high-level synthesis through dynamic functional unit allocation, *2009 International Conference on Communications, Circuits and Systems*, IEEE, pp. 1092–1096.

[136] Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K. et al. [2016]. Google's neural machine translation system: Bridging the gap between human and machine translation, *arXiv preprint arXiv:1609.08144* .

[137] Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C. and Philip, S. Y. [2020]. A comprehensive survey on graph neural networks, *IEEE Transactions on Neural Networks and Learning Systems* .

[138] Xydis, S., Palermo, G., Zaccaria, V. and Silvano, C. [2015]. SPIRIT: Spectral-Aware Pareto Iterative Refinement Optimization for Supervised High-Level Synthesis, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **34**(1): 155–159.

[139] Yassa, F. F., Jasica, J. R., Hartley, R. I. and Noujaim, S. E. [1987]. A silicon compiler for digital signal processing: Methodology, implementation, and applications, *Proceedings of the IEEE* **75**(9): 1272–1282.

[140] Yu, K., Bi, J. and Tresp, V. [2006]. Active Learning via Transductive Experimental Design, *Proceedings of the 23rd international conference on Machine learning*, ACM, pp. 1081–1088.

[141] Zacharopoulos, G., Barbon, A., Ansaloni, G. and Pozzi, L. [2018]. Machine learning approach for loop unrolling factor prediction in high level synthesis, *2018 International Conference on High Performance Computing & Simulation (HPCS)*, IEEE, pp. 91–97.

[142] Zacharopoulos, G., Ferretti, L., Ansaloni, G., Di Guglielmo, G., Carloni, L. and Pozzi, L. [2019]. Compiler-assisted selection of hardware acceleration candidates from application source code, *2019 IEEE 37th International Conference on Computer Design (ICCD)*, IEEE, pp. 129–137.

[143] Zacharopoulos, G., Ferretti, L., Giaquinta, E., Ansaloni, G. and Pozzi, L. [2018]. RegionSeeker: Automatically identifying and selecting accelerators from application source code, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* pp. 1–13.

[144] Zhang, Z., Fan, Y., Jiang, W., Han, G., Yang, C. and Cong, J. [2008]. Autopilot: A platform-based esl synthesis system, *High-Level Synthesis*, Springer, pp. 99–112.

[145] Zhao, J., Feng, L., Sinha, S., Zhang, W., Liang, Y. and He, B. [2017]. Comba: A comprehensive model-based analysis framework for high level synthesis of real applications, *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, IEEE, pp. 430–437.

[146] Zhong, G., Venkataramani, V., Liang, Y., Mitra, T. and Niar, S. [2014a]. Design Space Exploration of Multiple Loops on FPGAs Using High Level Synthesis, *Proceedings of the International Conference on Computer Design*, pp. 456–463.

[147] Zhong, G., Venkataramani, V., Liang, Y., Mitra, T. and Niar, S. [2014b]. Design Space Exploration of Multiple Loops on FPGAs Using High Level Synthesis, *Proceedings of the International Conference on Computer Design*, pp. 456–463.

[148] Zhou, Y., Gupta, U., Dai, S., Zhao, R., Srivastava, N., Jin, H., Featherston, J., Lai, Y.-H., Liu, G., Velasquez, G. A. et al. [2018]. Rosetta: A realistic high-level synthesis benchmark suite for software programmable fpgas, *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 269–278.

[149] Zitzler, E., Thiele, L., Laumanns, M., Fonseca, C. M. and Da Fonseca, V. G. [2003]. Performance assessment of multiobjective optimizers: An analysis and review, *IEEE Transactions on evolutionary computation* **7**(2): 117–132.

[150] Zuluaga, M., Krause, A., Milder, P. and Püschel, M. [2012]. Smart Design Space Sampling to Predict Pareto-Optimal Solutions, *ACM SIGPLAN Notices*, pp. 119–128.

[151] Zuo, W., Li, P., Chen, D., Pouchet, L.-N., Zhong, S. and Cong, J. [2013]. Improving polyhedral code generation for high-level synthesis, *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*, IEEE, pp. 1–10.