DESIGN AND OPTIMIZATION OF BEHAVIORAL DATAFLOWS

SHUANGNAN LIU

PhD

The Hong Kong Polytechnic University

2019

The Hong Kong Polytechnic University
Department of Electronic and Information Engineering

Design and Optimization of Behavioral Dataflows

Shuangnan Liu

A thesis submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

April, 2019

CERTIFICATE OF ORIGINALITY

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, it reproduces no material previously published or written, nor material that has been accepted for the award of any other degree or diploma, except where due acknowledgment has been made in the text.

_____ (Signed)

Shuangnan Liu (Name of student)

Abstract

Dataflow computing is a computational paradigm that uses the flow of data streams to accelerate the computation of tasks. Thus, it is also known as stream computing. It is essential in multiple domains, such as image processing and digital signal processing, where data processing throughput is critical. Modern heterogeneous System on Chips (SoCs) exploit this computational paradigm to accelerate the computation of specific computationally intensive functions mapped as hardware accelerators on the SoC. Moreover, designing these accelerators in low-level hardware description languages is tedious, error-prone and takes a relatively long time. Thus, companies have started using High Level Synthesis (HLS).

This thesis investigates the use of HLS to design and optimize dataflow hardware systems and uses Field Programmable Gate Arrays (FPGAs) as a test bed to demonstrate the usability of the developed methods.

In particular, this thesis first investigates the effects of pin multiplexing on individual hardware accelerators given as untimed behavioral descriptions that we call Behavioral IPs (BIPs), written in C or SystemC and addresses the issues of port assignments and mappings. Next, it explores the design space of dataflow systems considering the inter-module connections to identify a set of Pareto optimal configurations as multiple conflicting objectives need to be optimized such as area and latency.

Design teams now typically also prototype and emulate Application Specific Integrated Circuit (ASIC) designs on FPGAs. Thus, we study how to automatically convert optimized dataflows for an ASIC technology to FPGAs based on machine learning techniques. The proposed method can avoid having to fully re-explore the design when

an FPGA is targeted and achieves a speedup from hours to seconds while preserving the accuracy. This technique is extended to map complete dataflow systems from ASIC to FPGA platforms given full consideration of the design space of individual modules and the inter-module connections.

Finally, the thesis describes a strategy to map dataflows onto runtime reconfigurable FPGAs given specific area and performance constraints.

Publications

Journal

- **S. Liu**, F. Lau, and B. C. Schafer, "*Predictive Compositional Method to Re-optimize Complex Dataflows from ASIC to FPGA*", submitted to TCAD.
- **S. Liu**, M. Shah, S. Xu, and B. C. Schafer, "*Flexible Runtime Reconfigurable Overlay Architecture and Optimization for Behavioral Data-flow Applications*", submitted to IEICE.
- S. Xu, **S. Liu**, A. Mahapatra, M. Villaverde, F. Moreno, and B. C. Schafer, "*Design Space Exploration of Heterogeneous MPSoCs with Variable Number of Hardware Accelerators*", Microprocessors and Microsystems, Springer, pp.1-15, 2019.

Conference

- **S. Liu**, F. Lau, and B. C. Schafer, "*Investigation and Optimization of Pin Multiplexing in High-Level Synthesis*" in Proceedings of the 2018 on Great Lakes Symposium on VLSI - GLSVLSI '18, Chicago, IL, USA, 2018, pp. 427-430.
- **S. Liu** and B. C. Schafer, "*Learning-based interconnect-aware dataflow accelerator optimization*" in 2017 27th International Conference on Field Programmable Logic and Applications (FPL), Ghent, Belgium, 2017, pp. 1-7.
- **S. Liu**, F. Lau and B. C. Schafer, "*Accelerating FPGA Prototyping through Predictive Model-Based HLS Design Space Exploration*", Design Automation Conference (DAC), pp. 1-6, 2019.

Acknowledgments

This research was supported financially by Department of Electronic and Information Engineering, The Hong Kong Polytechnic University, and the research funding of Dr. Benjamin Carrion Schafer.

First, I would like to thank my supervisor, Dr. Benjamin Carrion Schafer, for guiding me on making research projects, writing publishable academic papers. I would also like to thank my supervisor, Prof. Francis C.M. Lau, for giving me the valuable advices of researches and support on campus.

I would also like to thank all the colleagues that I met in The Hong Kong Polytechnic University and The University of Texas at Dallas. It is my pleasure to meet you all.

Finally, I would like to give my thanks to my wife and my parents for their selfless support.

Contents

List of Figures

List of Tables

Glossary

ADRS	Average Distance from Reference Set
ANN	Artificial Neural Network
ASIC	Application Specific Integrated Circuit
BDL	Behavioral Description Language
BIP	Behavioral IP
BRAM	Block Random Access Memory
CB	Connection Box
CDFG	Control Data Flow Graph
CLB	Configurable Logic Block
CMOS	Complementary Metal Oxide Semiconductor
CNN	Convolutional neural network
CPU	Central Processing Unit
CWB	CyberWorkBench
DII	Data Initiation Intervals
DSE	Design Space Exploration
DSP	Digital Signal Processor
DVFS	Dynamic Voltage and Frequency Scaling
ES	Exhaustive Search
ESL	Electronic System Level
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
HDL	Hardware Description Language
HLS	High Level Synthesis
HPC	High Performance Computing
IC	Integrated Circuit
IDE	Integrated Development Environment
IO	Input/Output
LUT	Lookup Table

LVDS	Low Voltage Differential Signaling
MPSoC	Multi-Processor System-on-Chip
NoC	Network on Chip
NP	Nondeterministic Polynomial Time
NRE	Non Recurring Engineering
PCB	Printed Circuit Board
PCIe	Peripheral Component Interconnect Express
PCNT	Port Constraint
PE	Process Element
PLL	Phase Locked Loop
PREL	Port Relation
QoR	Quality of Results
RAM	Random Access Memory
RCS	Resource Constrained Scheduling
RTL	Register Transfer Level
S2CBench	Synthesizable SystemC Benchmark Suite
SA	Simulated Annealing
SB	Switch Box
SoC	System on Chip
SRAM	Static Random Access Memory
TAT	Turn Around Time
TCS	Time Constrained Scheduling
TRCS	Time-Resource Constrained Scheduling
UCS	Unconstrained Scheduling
VLSI	Very Large Scale Integration

Chapter 1

Introduction

Moore's law [75], which is the observation that the number of transistors in an Integrated Circuit (IC) doubles every 18 to 24 months, continues to drive the IC market growth. These ICs, also known as Very Large Scale Integration (VLSI), are reaching a number of billions of transistors per chip. Complete systems, also called System on Chips (SoCs) have become common. These SoCs include Central Processing Units (CPUs), memories, peripherals and application-specific accelerators. These accelerators exploit the inherent parallelism of main applications to run faster and consume less power. Some examples include Digital Signal Processor (DSP), image processing, and encryption algorithms.

To further optimize these accelerators, they can be arranged as dataflow or stream computing circuits. This arrangement allows fully pipelining the data across different accelerators' modules and hence, further speeding up the computation. Besides, due to the need to reduce the Turn Around Time (TAT) of ICs, companies have started to rely on High Level Synthesis (HLS) in particular to design the dedicated hardware accelerators, which are often firstly designed using a software language like C, C++ or MATLAB. It thus makes sense to exploit HLS to have a direct path between these untimed behavioral descriptions and the hardware implementations.

Fig. 1.1 shows an overview of the thesis, using a heterogeneous SoC as an example platform. The slaves (hardware accelerators) in this example represent the hardware accelerators designed to offload computationally intensive tasks from the masters

(micro-processor and memory).

The common thread among the different contributions is the optimization of hardware accelerators organized as data flows and in particular when these are specified as untimed behavioral descriptions for HLS.

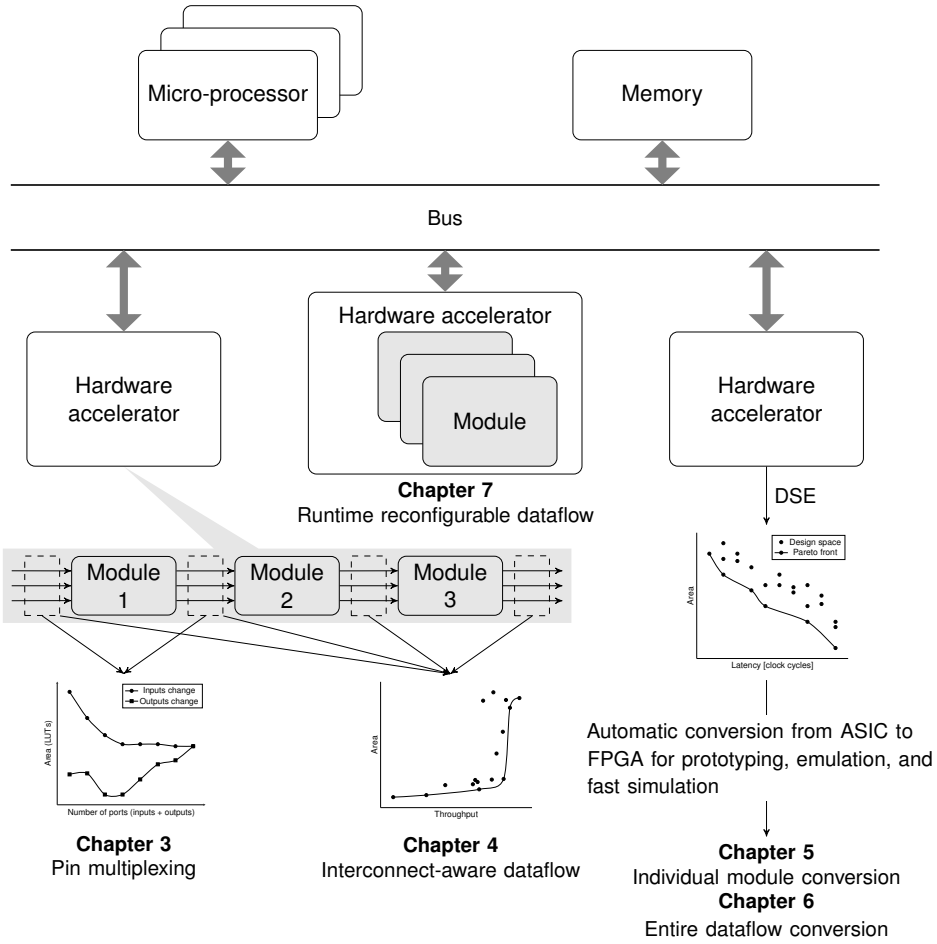


Figure 1.1: An overview of the thesis

To further increase the design productivity, raising the level of design abstraction is a feasible option. This fact has been proved when the design level was raised from technology-specific level such as Complementary Metal Oxide Semiconductor (CMOS) to a generic design practice that is Hardware Description Language (HDL) [19]. In raising the HDL-based design methodology, the community starts to embrace behavioral synthesis, which is also known as HLS. Using HLS allows designing hardware in the higher level language like C or C++ and also allows the hardware and software engineers speak the same language. Besides, designing in higher level means that the HLS tools will care about the error-prone details of Register Transfer Level (RTL) designs

and help the design team to meet the time-to-market requirement.

One of the most important features of using HLS over HDL is Design Space Exploration (DSE). DSE aims to explore the design alternatives before implementation so as to focus on the desired design points based on parameters of interest [54]. This exploration aims to help the designers to make decisions on implementing the micro-architectures of an individual function block considering different metrics such as power, performance, and cost. In HLS, it is possible to create different micro-architectures of the same functionality without changing the source code. While in HDL, this process has to involve the error-prone manual modifications of the source code.

1.1 Contributions

The thesis makes in particular five contributions in the aforementioned fields. These contributions corresponds to Chapters 3 – 7 as annotated in Fig. 1.1.

1.1.1 Investigating the effect of Pin Multiplexing in High-Level Synthesis

Pin [1] multiplexing is the task of sharing physical I/O ports among logic input and output signals of a circuit. Sharing pins is very important intra-chip, to reduce the interconnect congestion and inter-chip to reduce the number of IC pins needed, which impact the packaging costs and Printed Circuit Board (PCB) design costs. Moreover, in the Field Programmable Gate Array (FPGA) case, these have a very limited number of I/O ports, which force multi-FPGA designs to pin multiplexing their connections.

Pin multiplexing is an error-prone and time-consuming task in RTL as this has to be done manually. In HLS nevertheless, it is possible to control the number of I/O ports without modifying the behavioral source code. This is achieved by setting port constraints that specify the limit of available physical I/O ports and determining which logical ports to be mapped onto the same physical port. Note that the micro-architecture (datapath and Finite State Machine (FSM)) may also change when using different port constraint files because the HLS tool will try to optimize the circuit according to the input/output patterns. While in RTL, the micro-architecture does not change as the pin

[1] *Pin* and *port* are exchangeable in this thesis. Usually, the pin has only one bit, and port can yield multiple bits.

multiplexing is typically done as a wrapper around the RTL description. However, the assignment and mapping between original logic I/O signals and physical I/O ports is a Nondeterministic Polynomial Time (NP) problem. This thesis investigates the effect of pin multiplexing on the resultant circuit in HLS and presents a heuristic method to assign logic signals to physical ports efficiently.

1.1.2 Interconnection-Aware Optimization of Dataflows

The I/O ports are the interface between circuits. In a dataflow system composed of multiple modules connected in a chain, it is possible to tune the number of inter-module connections by changing simultaneously the output ports of the module ahead and input ports of the following module. We observe that adjusting the number of connections can improve the critical path delay and the routing congestion on FPGAs. Therefore, this second contribution presents a method to generate Pareto optimal data flow configurations of dataflow systems by adjusting the number of interconnects when area and throughput are considered as metrics. This problem is a multi-objective problem. Thus, the result is a trade-off curve of design Pareto-optimal configurations and not a single optimal solution. The proposed method adjusts the inter-module connections using a learning-based iterative approach, which can accurately predict the desired trade-off curve by synthesizing only a small portion of the entire design space.

1.1.3 Predictive Conversion of DSE from ASIC to FPGA

C-based VLSI design enables automatic design space exploration. This is typically done by setting different synthesis options in the form of pragmas (synthesis directives) or functional unit constraints. Setting these options does not necessarily require the modification of the source code, which means the same source code is reused in DSE. Some of the synthesis options are the number of utilized functional units and synthesis options to synthesize arrays, loops, and functions. The combination of these options increases exponentially which makes it impossible to explore the entire design space. Thus, heuristic methods have been proposed in the past. However, these methods still have to explore a portion of the whole design space, which might take hours or days for large designs. In this project, we consider the case that an Application

Specific Integrated Circuit (ASIC) design that has been explored previously needs to be emulated on an FPGA. Traditionally a full HLS DSE would be required as we show in this work that the synthesis options that lead to Pareto-optimal designs in the ASIC case do not lead to Pareto-optimal designs for the FPGA case. To address this, we present a predictive model which takes as input the DSE data for ASIC and outputs the DSE results for FPGA. The model completely avoids the time-consuming FPGA synthesis processes.

1.1.4 Complete Dataflow Systems Optimization on FPGAs

The previous contributions mainly focus either on individual modules or on the interconnections. However, a comprehensive optimization of dataflow systems should consider two aspects simultaneously: 1) the interconnections and 2) the synthesis options of all the modules. Thus, we also consider the interaction of these two elements to optimize complete dataflow systems. We introduce a compositional approach which combines with previous researches of pin multiplexing and DSE conversion. The method utilizes the existing ASIC DSE data to predict the performance of dataflow systems when mapped onto FPGAs without any synthesis.

1.1.5 Runtime Reconfigurable Optimization of Behavioral Dataflows

The last contribution presents a framework to map complete behavioral dataflows on FPGAs. The main drawback of stream computing is that it consumes a large number of hardware resources. Thus, we propose to make use of partial reconfiguration available in modern FPGAs to reduce the overall hardware resources. One of the modules of a dataflow will be mapped onto a reconfigurable block of the FPGA. After the execution, the data is stored in the FPGA internal Block Random Access Memory (BRAM), and the next module is mapped onto the same block for the subsequent computation. This implementation methodology is also called time-multiplexed FPGAs. The method explores the micro-architectures of individual modules and intelligently select the optimal design configurations to meet the area and latency constraints.

1.2 Thesis Organization

Chapter 2 introduces the background knowledge of the thesis. The discussion mainly consists of three parts: 1) dataflow computing systems which are widely used as hardware accelerators, 2) a brief introduction of FPGA including the architecture and other embedded dedicated hardware, 3) an introduction of HLS.

Chapter 3 investigates and optimize pin multiplexing in behavioral level. This chapter describes the heuristic method and compares the proposed approach to other methods such as simulated annealing.

Chapter 4 focuses on optimizing the inter-module connections of dataflow systems on FPGAs. The predictive iterative approach is discussed, and other state-of-the-art methods are presented as a comparison.

Chapter 5 aims at the DSE of individual modules on FPGAs. By utilizing existing DSE data for ASICs, a predictive model can estimate DSE results on FPGA platforms. The model has two levels and will be described in detail.

Chapter 6 introduces a compositional approach to optimize complete dataflow systems on FPGAs. The approach borrows the idea in chapter 5 which is further combined with a heuristic.

Chapter 7 discusses the reconfigurable scheme and the approach to optimizing the mapping of dataflow applications.

Chapter 8 summarizes the work discussed in the thesis and provides the potential directions to extend these works.

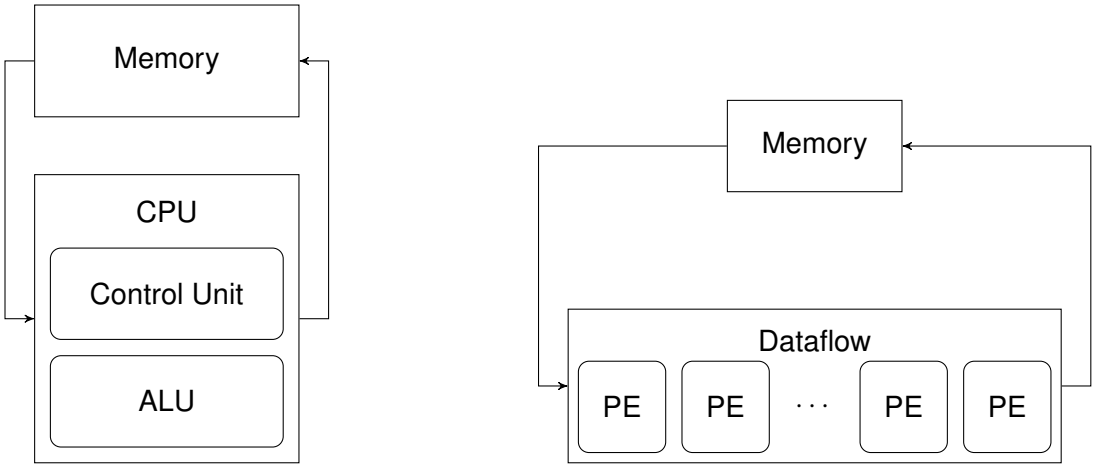
Chapter 2

Background

This chapter covers the necessary background information in order to understand the contributions of this thesis fully. It first discusses the architecture of dataflow computing systems. Then it gives a brief introduction of FPGAs including the architecture of FPGAs. Finally, this chapter discusses HLS including the primary steps involved in HLS and summarizes the advantages of HLS as well as state-of-the-art tools.

2.1 Dataflow

Dataflow is a term that is widely used in both hardware and software domains. In a nutshell, dataflow exploits the concurrency in computational tasks to execute these more efficiently. In the case of hardware, it refers to a type of hardware architecture which is different from the classic von Neumann architecture [80]. This chapter only focuses on dedicated dataflows in the hardware domain, and FPGAs will be used to prototype these dataflows. The dataflow concept was first introduced in [58]. This work referred to it as a *systolic architecture* for the reason that it works much like the blood circulating from and to the heart. One other comparison is with the assembly line of vehicles. For instance, the people at one stage repeat the same task at a specific frequency, where they receive the products from the previous step and pass the processed product to the next step. In the VLSI case, each station is represented by a module that does a specialized computation and then passes it to the next module in the dataflow.



(a) von Neumann architecture (b) Dataflow architecture

Figure 2.1: Comparison between von Neumann and dataflow architecture

Dataflow architecture is an alternative to the classic von Neumann architecture. Fig. 2.1 shows an overview of the two architectures. Assuming the data is stored in *Memory*, in case of von Neumann architecture, the data is passed to a CPU which repeats the fetch, decode, execute cycles continuously for each instruction. After the computation, the processed data is sent back to the memory, and the next cycle is repeated.

However, in case of dataflow case, this consists of multiple interconnected processing units Process Elements (PEs). The first PE reads the data from *Memory*, then the processed data is passed to the second PE while at the same time the new set of data will be read by the first PE. The same mechanism applies to the rest PEs. For example, the second PE passes the processed data to the third one, meanwhile, read the new data from the first PE. This makes the computation more efficient as fewer memory accesses are required. Moreover, the PEs are dedicated functions that can more efficiently compute a specific task compared to the general purpose CPU.

Dataflows are mostly implemented as special-purpose hardware which is commonly applied to the situation where repetitive processes are executed on a large set of data. Some examples include the fields of image processing, digital signal processing, and network routing, etc. For example, [58] introduced several types of dataflow architecture to implement convolutional computation. In this example, the PEs have the same functionalities. In other situations, like JPEG encoder, the PEs have different

functionalities. This thesis mainly focuses on the latter case, although the techniques developed can also be applied to the first type of dataflows.

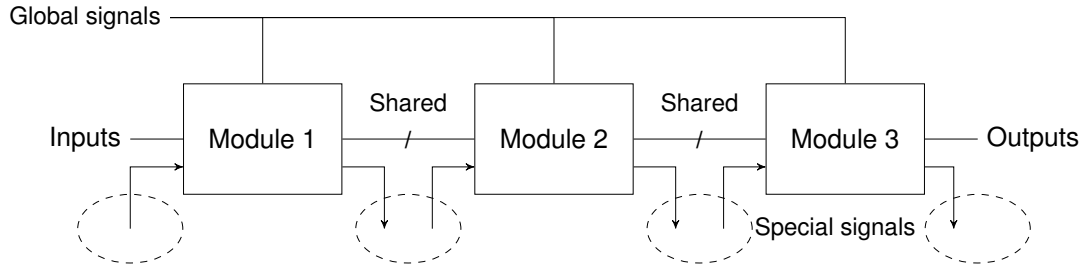


Figure 2.2: A typical architecture of dataflow

Typically, there are two ways to build a faster computing system: 1) increasing the clock frequency and 2) exploiting concurrency. Dataflows exploit both by deeply pipelining the processing units to increase the clock frequency and by maximizing the concurrency of the application.

Moreover, the modularity of dataflow systems enables the ease of design, and its maintenance, and the adaptability of various constraints. Fig. 2.2 shows a typical block diagram of a dataflow. There are three modules (PEs), that share the same global signals such as *clock* and *reset*. The first module accepts the inputs and communicates with the second module through the *Shared* interconnect. The second module processes the data passed by the previous module and passes it to the third module which then generates the outputs of the circuit. There might also be some *Special* signals which are specific to different modules.

Fig. 2.3 shows an example of utilizing dataflow for a JPEG encoder. Without any dataflow structure, we can design the entire encoder as a single module. In this case, the circuit will have a certain latency, which implies that an $8\times$ 8-pixel block is passed to the encoder and new output is generated after N clock cycles, where N is the latency of the encoder in terms of clock cycles. Here, we assume the latency is 20 clock cycles. To increase the throughput, the encoder can be designed as a dataflow as shown in the same figure. This dataflow includes three modules: DCT, RLE, and Huffman encoding, where the latencies for each module are 10 (DCT), 5 (RLE), and 5 (Huffman). The module with the largest latency is the bottleneck of the entire circuit (the DCT in this case). This implies that the encoder can process a pixel block every ten clock cycles.

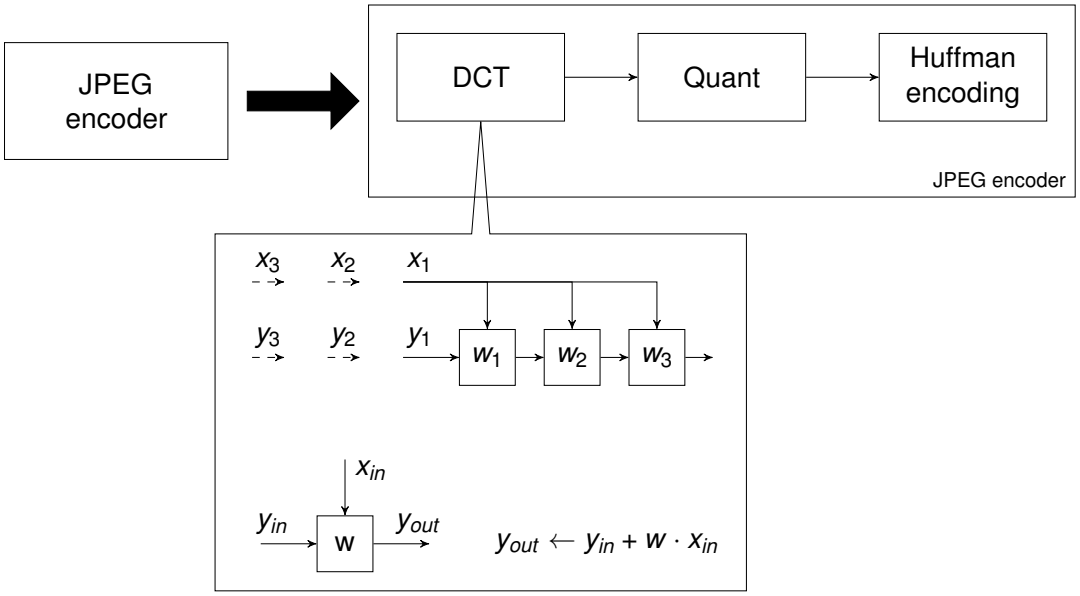


Figure 2.3: JPEG encoder using dataflow paradigm

Once the dataflow is in a steady state (filled with data), the encoder can produce the processed pixel block every ten clock cycles, as opposed to the 20 clock cycles of the initial implementation. Also, further optimization can be applied to DCT since it primarily performs matrix multiplication. The block at the bottom shows the structure of the PEs, each of which performs the same operation – multiplication and addition. In this way, the dataflow paradigm can be applied to the module itself, further reducing the latency to 5 clock cycles. This, in turn, means that the latency of the encoder is also reduced to 5 clock cycles, thus improving the performance of the entire design. Note that matrix multiplication is essential in many applications like Convolutional neural network (CNN).

Because of the potential to increase performance, dataflows have been used in many applications mainly prototyped on FPGAs, like CNN implementation on FPGAs using dataflow structure [65, 111, 114]. In addition to CNN, MaxCompiler [106] improves the execution speed of computation intensive tasks, that are described in MaxJ, by exploring the parallelism sub-tasks that are assigned to PEs.

The main drawback of dataflows is that they need to be designed and optimized manually, which is a tedious and error-prone process. For example, the synchronization between the different modules is critical to achieving good performance. Thus, the designers have to ensure that the latencies/Data initiation Interval of the elements are

matched so that the previous stage can pass the complete, correct data to the next step. Otherwise, First In First Out (FIFO)s are needed and faster modules will have to wait for the slower modules, which in turn limit the maximum speedup. Moreover, the larger amount of interconnects between modules might lead to congestion problems in the interconnect, which in turn leads to longer delays. Since dataflows are often implemented on FPGAs as either final products or as test beds for ASIC designs. We describe what FPGAs are in detail in the next section.

2.2 Field Programmable Gate Array (FPGA)

FPGAs are pre-fabricated silicon devices which can be electrically programmed in the field to become almost any kind of digital circuit or system. FPGAs were first introduced by Xilinx in 1985. They provide fast and efficient solutions from low-end to the high-end market, FPGAs also allow cheaper and faster time-to-market solutions compared to ASIC especially for small to medium productions.

The architecture of an FPGA is critical to its performance. A Configurable Logic Block (CLB) is the fundamental element of an FPGA. Thus, the implementation of a CLB is important. A fine-grained (e.g., transistor based) structure would lead to high routing complexity, while a coarse-grained (e.g., processor-based) architecture would lead to a low utilization ratio. To this end, Lookup Table (LUT) based CLBs provide a good trade-off. In practice, a CLB contains multiple LUTs, and the implementation is vendor dependent.

A LUT acts as the truth table having k input signals and outputs the logic results of the inputs. Fig. 2.4 shows an example of a three-input LUT implemented as a set of muxes. It performs the logic given in the truth table by setting different mux values. For instance, if $B = 1$, $A = 0$, and $S = 1$, the output Z is 1. Thus, a k-input LUT can perform any 2^k logic functions. Moreover, LUTs with more inputs can be built using LUTs with fewer inputs [51].

Fig. 2.5 shows an overview of an FPGA architecture which contains a 2-d array of CLBs. This is typically called a mesh-based structure or island-style structure. The I/O signals are placed around the CLBs to minimize the communication overhead with

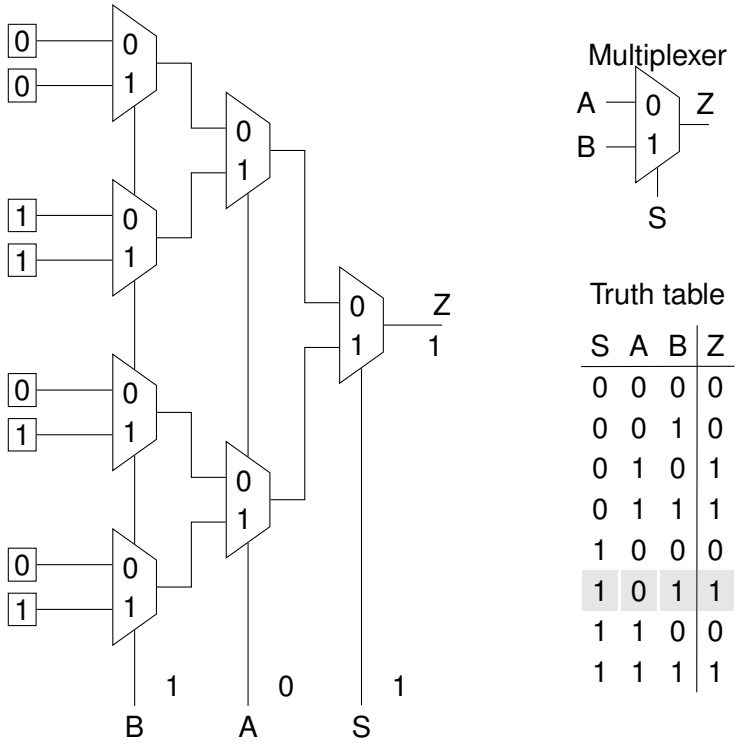


Figure 2.4: The architecture of a 3-input LUT

external circuits. Note that the I/O ports are configurable, which means they can act as input, output, or inout, and the specifications can also be tuned. In-between the CLBs, there are switch boxes Switch Box (SB) and Connection Box (CB), which can be reprogrammed to control the interconnect between CLBs.

Due to the nature of pre-fabrication and high configurability, FPGA designs lower the Non Recurring Engineering (NRE) costs and accelerate the time-to-market compared to ASICs. However, ASICs are still one to two orders of magnitude more efficient. Implementing the same design on an ASIC leads to smaller, faster, and more power efficient designs, which could amortize the high NRE costs among mass productions. Nevertheless, FPGAs and ASICs are closely related in modern hardware designs. For example, FPGA are typically used in ASIC prototyping, emulation or hardware accelerated verification. One main problem is that ASIC designs need to be converted to FPGA designs, often manually. Thus, part of this thesis deals with the automatic conversion of optimized designs for ASICs to FPGAs.

To minimize the performance gap between FPGA and ASIC, multiple dedicated components are embedded on FPGAs. These components are ASIC-like hardware

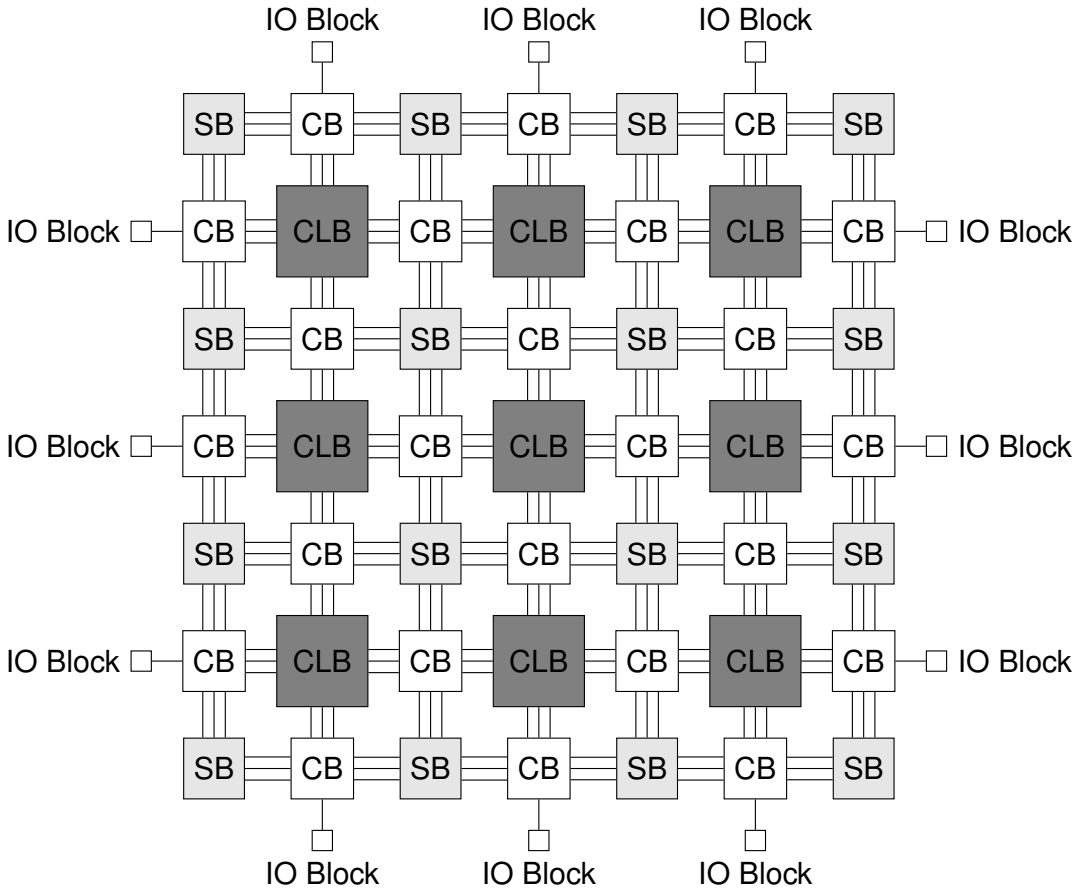


Figure 2.5: The architecture of an FPGA

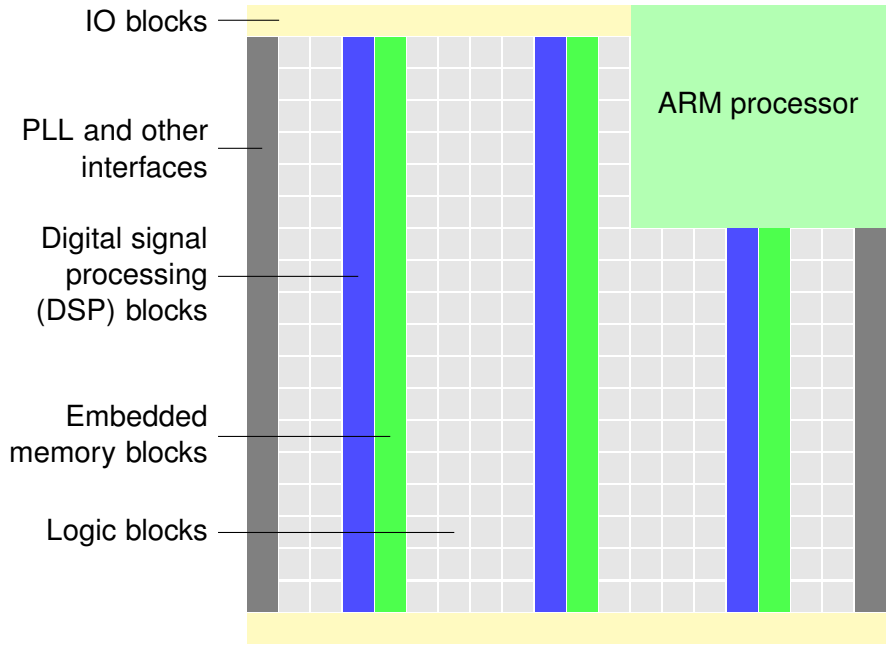


Figure 2.6: The FPGA structure with integrated dedicated hardware

resources including memories, DSPs, PLLs, and high-speed interfaces as shown in Fig. 2.6. In particular:

2.2.1 BRAM

FPGAs include a large amount of on-chip memories. BRAMs are Static Random Access Memory (SRAM) based dual-port Random Access Memory (RAM)s which can be configured to implement single-port memory, dual-port memory, and FIFOs. Also, they can be cascaded to create even larger memories. BRAMs are normally distributed in a compact columnar fashion to achieve maximum performance.

2.2.2 DSP Macros

DSP macros support full-precision multiplication for operators with various bit width. A DSP macro mainly consists of a pre-adder, a multiplier, and an add/subtract/logic unit. These units have high configurability thus can work at different modes.

In general, multiplication consumes 20$\times$ LUTs than addition, and is not area efficient for many multi-media applications. Also, the wire delay, when mapped onto the reconfigurable fabric, would make multipliers very slow. To improve the performance, DSP macros can be manually instantiated or inferred by the synthesis tools. These DSPs, which are fabricated to operate at high speed and consume low power, are distributed in columns on the FPGA fabric to allow compact designs.

2.2.3 Embedded Processors

To further empower FPGAs, some FPGA families now include microprocessors embedded in them to execute fewer computation intensive tasks. The microprocessors can be integrated as a soft IP or a hard IP. For example, Intel's Nios processor [52] and Xilinx's MicroBlaze [117] are soft processors designed to allow custom hardware instructions. In terms of hard processors, Intel Cyclone V SoC [50] and Xilinx Zynq SoC [118] FPGAs integrate ARM-based processors.

2.2.4 Other Dedicated Hardware

The Phase Locked Loops (PLLs) are dynamically configurable phase-locked loops to generate the clock signals spreading the entire FPGA. The high-speed interfaces including Peripheral Component Interconnect Express (PCIe) channels and transceivers. With the integrated hardware, designers can offload the hardware-preferred tasks, thus improving the performance of FPGAs.

2.3 High-Level Synthesis (HLS)

HLS is a VLSI design methodology which takes as input a high-level language like C or C+ and generates efficient synthesizable Verilog or VHDL. It was first introduced in the 1980s and did not gain much attention until the early 2000s when the new generation of HLS tools was introduced that generated RTL code comparable to hand-coded code.

For ASIC design, SystemC has become the dominant input language. SystemC was standardized by the IEEE in the 1466 LRM and has a synthesizable subset for HLS. It is a class library of C++, and it allows the simulation of concurrent executions of hardware designs. Therefore, SystemC is widely used in Electronic System Level (ESL) modeling. In the meantime, a particular subset of SystemC syntax is synthesizable, which means that the HLS tools can transfer these codes into RTL and further pass it to the physical design.

HLS is currently widely adopted in both industry and academia. The use of HLS has proved to enable the reduction of the design time, which in turn allows companies to meet time-to-market windows [72, 73]. There is a wide range of commercial HLS tools: Vivado HLx [119], CyberWorkBench [78], Catapult HLS [71], Stratus HLS [9] and also some mature open source academic tools such as LegUp [20], and Bambu [108].

HLS is the transformation from behavioral to structure [10], such as from high-level languages (C, C++) to RTL. The high-level specifications are first converted to intermediate representations like control and data flow diagram (CDFG [1]), DeJong's hybrid flow graph [53], SSIM flow graph, and Finite state machine with data [24]. CDFG is one of the mostly used diagrams. Then the transformation goes through three main steps: scheduling, allocation, and binding. The three steps generate the hardware

structure given the technology libraries and time/resource constraints. Finally, the RTL code is generated using HDL like VHDL and Verilog.

2.3.1 Mechanism

To create hardware circuits, HLS takes as inputs 1) the design described in C/SystemC, 2) the target clock frequency and 3) the technology libraries. The resultant RTL code consists of two parts: controller and datapath as shown in Fig. 2.7. A controller is often implemented as a FSM, and it controls the state transition to ensure the correctness of the circuit. The datapath is also known as the micro-architecture. It is a collection of functional units that are organized and connected properly. A typical view of the datapath is shown in Fig. 2.8. It shows (a) the elements which compose the datapath and (b) the particular organization of these elements.

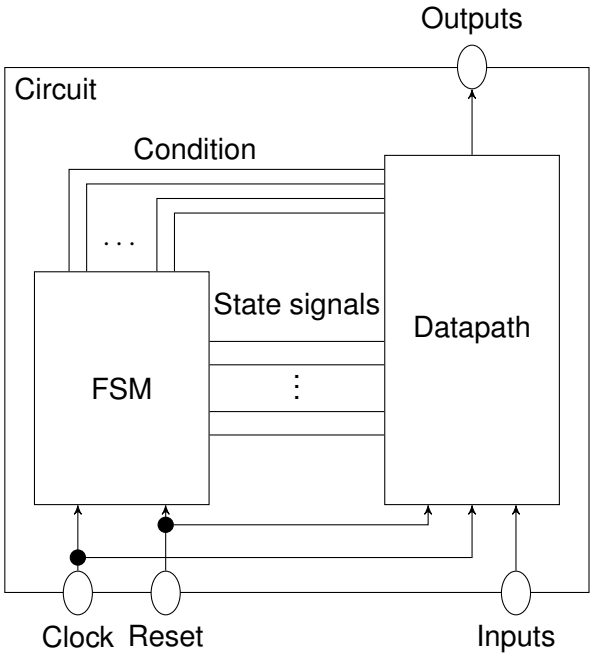
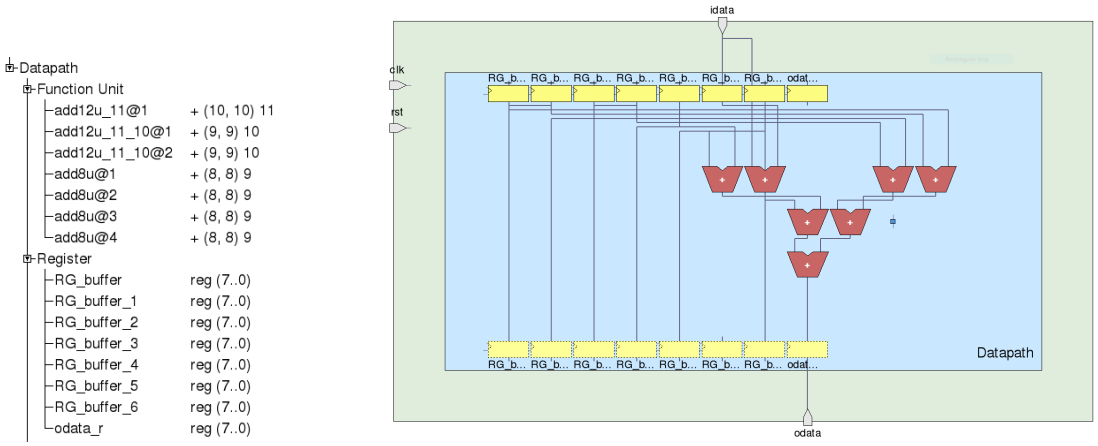


Figure 2.7: The diagram of a synthesized circuit

An example is shown in Fig. 2.9, which computes the moving average of eight numbers. There is a `buffer` of size eight, which holds the seven latest values (lines 1 - 3) and the new input data (line 4), and lines 6 - 9 compute the average of the eight numbers by adding up all the numbers and dividing by 8 (partial bit extraction is only needed for this). Finally, the program writes the average value to the output.



(a) Datapath elements (b) Datapath structure

Figure 2.8: A view of datapath using CyberWorkBench [78] (a commercial HLS tool)

In the code, we set the bit-width of `buffer` to 8, and the input data is unsigned. As a result, the datapath has seven adders and eight registers. Fig. 2.8(a) also presents the specifications of the adders. For instance, `add8u@1` is an adder which has two 8-bit inputs and one 9-bit output denoted as `(8, 8) 9`. The top three adders have larger bit-width because they are computing the intermedium values. The registers whose name start with RG are the implementation of `buffer`, and the `odata_r` is the register holding the output value.

```
1  for (i = 7; i > 0; i --) {
2      buffer[i] = buffer[i - 1];
3  }
4  buffer[0] = idata;
5  sum = 0;
6  for (i = 0; i < 8; i++) {
7      sum += buffer[i];
8  }
9  odata = sum / 8;
```

Figure 2.9: Code snippet of average-8

The next subsections describe the main steps behind HLS, particularly, compilation and a three-step transformation which includes scheduling, allocation, and binding.

Compilation

The initial step of HLS is to convert the behavioral description to an intermediate representation like Control Data Flow Graph (CDFG). A CDFG is a directed graph. The nodes indicate the operation on the data, and the edges control the flow of the data. The nodes in a CDFG can be classified into the following types:

- Operational nodes: They perform arithmetic, logical, or relational operations, such as addition, equality checking.
- Control nodes: They control the operations like conditions and loop constructs, such as case statements, for loop.
- Storage nodes: They represent the read and write operations, such as registers. The edges transfer the data processed by the nodes, such as read the values from the predecessor nodes and output the values to the successors.

For example, given the source code shown in Fig. 2.10, the compiler may generate two graphs: A control flow graph and data flow graph. Data flow CDFG represents the parallel evaluation for all branches of a control node. This is in a real sense closer to hardware realization of the circuit. Optimizations and other steps of HLS can, in turn, be performed on the CDFG. The control flow graph CDFG is an almost one-to-one mapping between the nodes of CDFG and the code.

Before generating the hardware structure, some optimization techniques are performed. There are three main optimization categories:

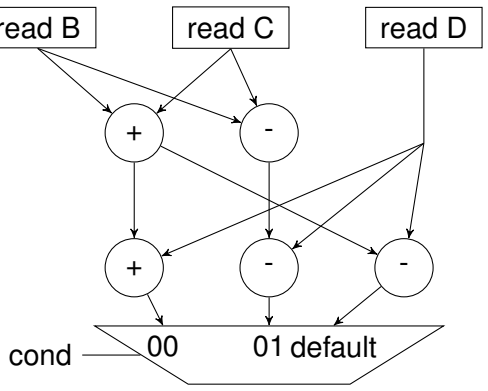
1. Compiler based optimizations [77]
2. Flow-graph based optimizations [84]
3. Hardware library-based optimizations

Scheduling

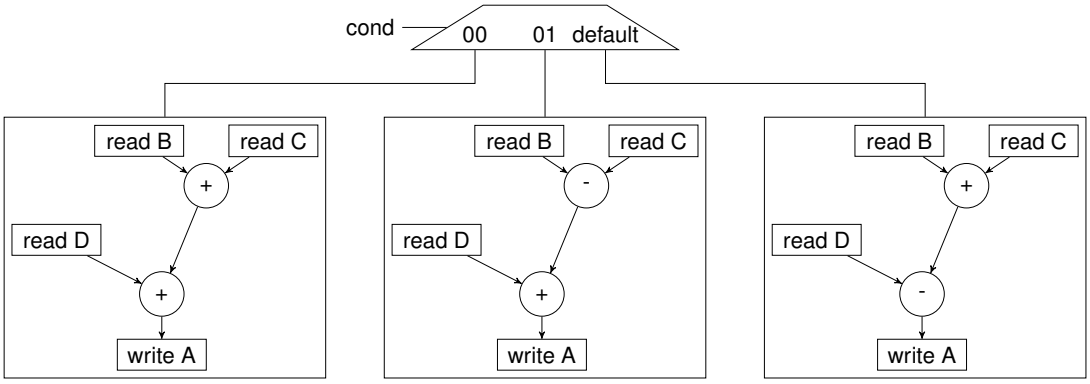
Scheduling determines the sequence, in which the operations are executed, to produce a control step schedule. A control step includes a set of operations in parallel. The operations can be categorized into different types, and each operator in the library can execute (a) certain type(s). The scheduling sequence cannot violate data dependency. There are four situations while performing scheduling:

```
int example(int B, int C,
            int D, int cond) {
    int A = 0;
    switch (cond) {
        case 0b00:
            A = B + C + D;
            break;
        case 0b01:
            A = B - C + D;
            break;
        default:
            A = B + C - D;
    }
    return A;
}
```

(a) Example code



(b) Data flow graph



(c) Control flow graph

Figure 2.10: An example of CDFG

1. Unconstrained Scheduling (UCS)
2. Time Constrained Scheduling (TCS)
3. Resource Constrained Scheduling (RCS)
4. Time-Resource Constrained Scheduling (TRCS)

Fig. 2.11 explains the different cases by using the following example code snippet: `f=(a+b+c+d)*e`. Each operator has a unique ID (o_i) and belongs to a specific type (t_i). For example, o_1, o_2, and o_3 are operators of type t_1 since they compute the addition, and o_4 is an operator of type t_2 which performs multiplication.

Un-constrained Scheduling (UCS) This is the easiest type of scheduling. It takes the nodes in a CDFG and assigns them to the control steps in sequence. Fig. 2.11(a)

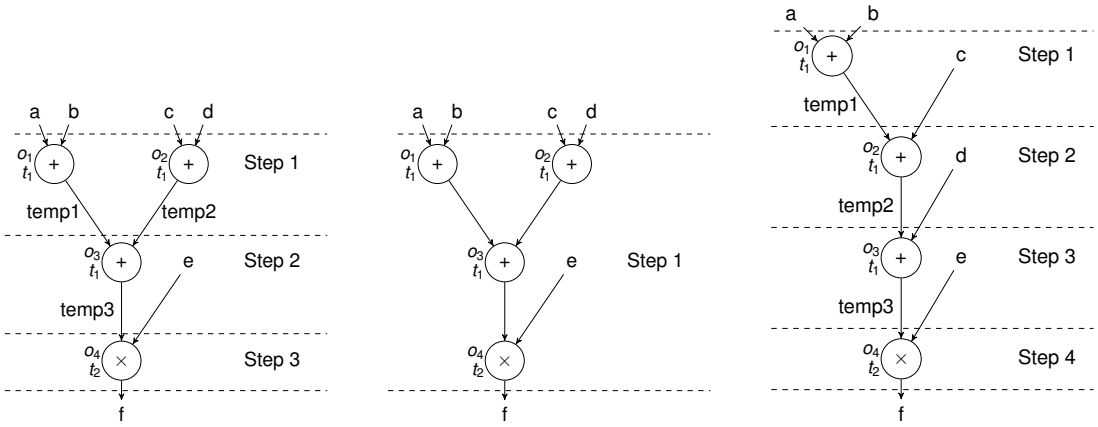


Figure 2.11: The scheduling results given different constraints

shows a possible scheduling result using UCS. The operators are distributed among three control steps. The first control step consists of two operators (o_1 and o_2) which are of type t_1 (addition). They take the inputs a, b, c, and d, then produces the outputs temp1 and temp2. The second control step contains one operator o_3 of type t_1. It takes as inputs the data from step 1 and produces the processed data temp3. The last control step has the operator o_4 of type t_2 (multiplication), which takes as input temp3 and e, then produces f as the output. Note that the operators in different control steps may often be shared. Therefore, scheduling determines that the circuit consumes two operators of type t_1 and one operator of type t_2.

Time Constrained Scheduling (TCS) TCS considers time constraints such as the maximum number of control steps. For example, if the maximum number of control steps is three, the scheduling result might be the same as that shown in the above figure. However, if the maximum number is one, the operations should be completed in a single control step as shown in Fig. 2.11(b). Therefore, this scheduling leads to the maximum number of hardware resources: three adders and one multiplier.

Resource Constrained Scheduling (RCS) Resource constraints may also apply to scheduling. For example, if only one adder and one multiplier are allowed for the circuit, the scheduling will generate the structure as shown in Fig. 2.11(c). In the case of RCS, allocation might be performed before scheduling.

Time-resource Constrained Scheduling (TRCS) Both resource constraint and time constraint can be applied during scheduling. Note that if one of them is over-constrained, the scheduling may fail. In this example, if the resource constraint is one adder and one multiplier, the smallest latency is 4. However, if the time constraint is smaller than 4, the scheduling may not succeed.

Scheduling Algorithms Scheduling is a complex task; thus many algorithms have been studied trying to solve this problem [112]. These algorithms can be summarized into the following categories:

- Heuristics
  - As soon as possible (ASAP) [112]
  - As late as possible (ALAP) [112]
  - List scheduling (LS) [89, 46]
  - Forced directed scheduling (FDS) [91]
- Exact
  - 0-1 integer linear programming (ILP) [38, 79, 16, 49, 35]

Allocation

Scheduling determines the types of operators and their quantity. Allocation determines the exact operator modules which are available in the library. Therefore, the area, power, and speed are also determined after allocation.

The operations could be addition (+), comparison (>, <) and shifting (<<, >>), which are pre-defined in the technology libraries. Meanwhile, the bit-width of the operations will be analyzed to optimize resource utilization and speed of the resultant architecture. In this example, assume the bit width of the inputs is 8. Then the bit width of operator o_1 should be two 8-bit inputs and one 9-bit output, denoted as $(8, 8)9$. Therefore, o_2 is $(9, 9)10$, o_3 is $(10, 10)11$, and o_4 is $(11, 11)22$.

In a typical design library, a type of operator may consist of multiple choices with different metrics. For example, a multiplier is much slower than an adder. Once the multiplier is selected, we can choose a slow adder to save area and power.

Binding

Operators are shared among operations. Sharing also introduce other elements, e.g., registers and muxes. Binding [90] assigns operations to operators and variables to registers.

1. Storage binding: assign inputs, outputs, and temporary variables to registers. Two variables that are not alive simultaneously can share the same register.
2. Functional-unit binding: assign operations to functional units. Two operations having the same type do not appear in the same control step can share the same operator.
3. Interconnection binding: assign interconnection units such as muxes or buses to a data transfer.

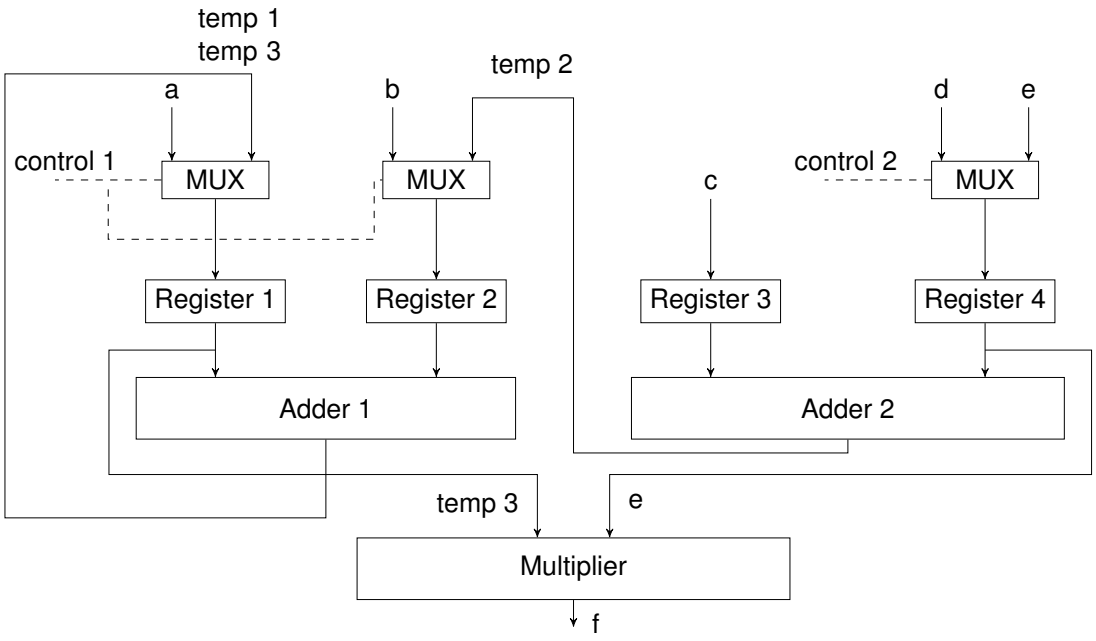


Figure 2.12: The CDFG after binding

Fig. 2.12 shows the binding results under a time constraint. It contains three control steps.

Step 1

- control 1 = 0: bind a to Register 1, b to Register 2
- control 2 = 0: bind d to Register 3

- Bind c to Register 3
- Bind a + b to adder1, bind c + d to adder 2

After this step, adder1 generates temp 1, adder2 generates temp 2.

Step 2

- control 1 = 1: bind temp 1 to Register 1, bind temp 2 to Register 2
- control 2 = X: do not use Register 4
- Do not use Register 3
- Bind temp 1 + temp 2 to adder1

After this step, adder1 generates temp 3.

Step 3

- control 1 = 1: bind temp 3 to Register 1, do not use Register 2
- control 2 = 1: bind e to Register 4
- Do not use Register 3
- Bind temp 3 * e to Multiplier

After this step, Multiplier generates the output.

Binding also generates three control signals which can be used to built the controller (FSM):

- Step 1: 00
- Step 2: 1X
- Step 3: 11

2.3.2 Advantages of HLS Over RTL

Raising the abstraction level brings many benefits. One unique advantage of HLS over traditional RTL VLSI design is that different micro-architectures can be easily generated without having to re-write the input behavioral description. This can be achieved by inserting *pragmas*, which are usual comments in C/C++ but preceding with special terms that are recognized by the HLS tool. The *pragmas* are directives that tell the tool

how to synthesize a certain part of the circuit. For instance, a typical option to synthesize a loop is *unroll* as shown in Fig 2.13. The two loops are identical, each code has a

```
// pragma unroll_times = 0
for (i = 0; i < 4; ++i) {
    B[i] = A[i] + 1;
}
```

```
// pragma unroll_times = all
for (i = 0; i < 4; ++i) {
    B[i] = A[i] + 1;
}
```

Figure 2.13: An example of loop unrolling

normal C-style comment above the loop. Note the keyword *pragma* indicates that this line of comment is a special directive. The following `unroll_times = 0` tells the tool not to unroll the loop, which produces the structure shown in Fig. 2.14 (left). It takes four clock cycles to complete the loop, and in each clock cycle, it performs an addition. Meanwhile, the adder can be shared among different clock cycles; thus this structure trades speed for area. For the second case that uses `unroll_times=all`, the resultant structure is shown in Fig. 2.14 (right). It only takes one clock cycle to complete the loop. However, it has to use four adders since the four additions are performed simultaneously. Therefore, this structure is fast but requires more logic resources. Besides

```
// Clock 1
B[0] = A[0] + 1;
// Clock 2
B[1] = A[1] + 1;
// Clock 3
B[2] = A[2] + 1;
// Clock 4
B[3] = A[3] + 1;
```

```
// Clock 1
B[0] = A[0] + 1;
B[1] = A[1] + 1;
B[2] = A[2] + 1;
B[3] = A[3] + 1;
```

Figure 2.14: The results of loop unrolling using two different pragmas

loops, HLS is also able to perform various synthesis strategies on other elements such as functions and arrays, which produce a large amount of pragma combinations, thus generating different structures. This is typically called design space exploration.

DSE is the task of finding the design points which the designers are interested in. The design points are the distinct micro-architectures which have the same functionality, and the points may have different attributes, which is also known as objectives, like area, speed, and power. Multiple objectives indicate that the solution is a trade-off

curve that consists of Pareto optimal designs rather than a single global optimal design. For example, in Fig. 2.15, the design space contains all the points which have the same functionality but different micro-architectures, thus have different metrics. However, designers are only interested in the points on the trade-off curve. These points are called Pareto-optimal points, and the curve is known as Pareto front. The Pareto-optimal points are equally important. For instance, assume *Objective 1* is latency (clock cycles) and *Objective 2* is area. So design point *A* is larger than design point *B*. However, it is highly likely that design *A* is also faster than design *B*. In this case, the choice of the design points depends on the underline application, e.g., either for high performance or for compact area. DSE is important because it helps the designers to learn the upper and lower bound of the designs and guides the designers on how to build different micro-architectures to balance different objectives.

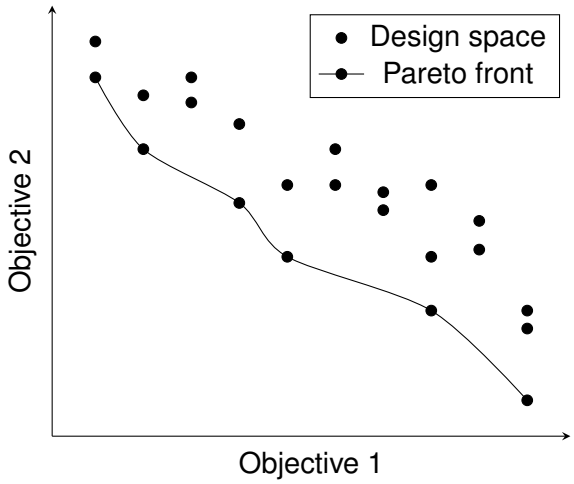


Figure 2.15: An example of DSE results

Due to the increasing complexity of modern SoCs, the design space is growing exponentially. To solve this multi-objective problem, heuristic algorithms are widely adopted, and the genetic algorithm is proved to be one of the most efficient algorithms [102, 43, 55]. This type of algorithms mimics the evolution of living species that the positive factors leading to desired results will be preserved and be propagated to the next iterations. Note that the algorithms are designed to optimize (minimize or maximize) a single value. Thus it is required to transfer the multiple objectives to a single value. This transition is known as *cost function* (2.1), where the summation of

the coefficients (θ_i) is 1.

$$cost = \theta_1 \cdot (Objective1) + \theta_2 \cdot (Objective2) + \cdots \tag{2.1}$$

So in Fig. 2.15, if the goal is to minimize the cost function, in case of design *A*, $\theta_1 = 1, \theta_2 = 0$; and for design *B*, $\theta_1 = 0, \theta_2 = 1$.

To further improve the heuristic methods, machine learning techniques appears to be a promising solution [88, 13]. Traditional heuristic algorithms utilize the predictive nature of machine learning algorithms to skip the actual synthesis of trivial design points and meanwhile focus on the potential important points. In this way, only a small portion of the design points is sufficient to estimate the performance of the entire design space.

2.3.3 Commercial and Academic HLS Tools

CyberWorkBench

CWB [78] is an Integrated Development Environment (IDE) which allows the All-in-C design and verification of SoCs. It was developed at NEC's R&D laboratories. The All-in-C concept is enabled by the preferred input language called Behavioral Description Language (BDL) which is an extended version of ANSI-C. The concept contains two principal ideas:

- all-modules-in-C: It allows all modules including control intensive and computation intensive applications described in behavioral C language. Meanwhile, it accepts the legacy RTL code as black boxes, thus saving the effort of converting those RTL codes back to C sources.
- all-processes-on-C: The processes include synthesis, verification, and debugging. The behavioral C descriptions are converted automatically to RTL code under the design constraints. For verification, the cycle accurate simulation model is generated to observe the accuracy and performance of the converted RTL code. CWB also allows writing properties or assertions in the C descriptions to check the bugs. To prove the equivalence between the C description and RTL code, the

C-RTL equivalence prover checks functionality matching using the information of the conversion, which ensures that the optimizations should not change the functionality accidentally.

Catapult

Catapult [71] was developed by Mentor. It is an interactive C synthesis tool, and it takes as inputs the pure ANSI C++. Catapult also allows the verification in C environment. This is enabled by wrapping the generated RTL code in the SystemC foreign module; thus the C code for functionality testing can be reused to check the correctness of the outputs of the RTL code.

Catapult has achieved successful experience in the industry for ASIC designs while showing the advantages of HLS over traditional RTL design, including the accelerated time-to-market and the optimizations which are non-trivial to obtained in RTL.

Vivado HLS

Vivado HLS [119] is based on the commercial platform-based ESL synthesis system called AutoPilot which is an extension of xPilot originally developed at UCLA. AutoPilot was designed by AutoESL Design Technologies Inc. which was acquired by Xilinx in 2011.

One of the key features of Vivado HLS is platform modeling, which means that it takes full advantage of the characteristics of the target platform so that maximum optimizations can be conducted. When targeting FPGAs, the hardware resources, such as arithmetic units, control units, interfaces, and memories, are pre-characterized regarding the delay, area, and power.

Vivado HLS is one of the most popular commercial HLS tools for FPGAs, and it is widely adopted in academia.

LegUp

LegUp [20] is an HLS compiler mainly for FPGA platforms developed at the University of Toronto. LegUp has two unique features which make it distinct from other HLS tools. The first feature is that LegUp supports hardware synthesis of Pthreads and

OpenMP. This enables the easy adoption of parallelism on FPGAs. The second feature is the universal support of FPGAs from major vendors which are Intel, Xilinx, Lattice, Microsemi, and Achronix. This allows little to zero modifications of the source code when implementing across vendor-specific FPGAs.

Bambu

Finally, there are also a variety of academic HLS tools that have shown to produce good quality results. For example, Bambu [108] which is developed by the Politecnico de Torino in Italy.

Chapter 3

Pin Multiplexing of Behavioral Modules

This chapter investigates the effect of pin multiplexing on the resultant micro-architecture of behavioral descriptions after HLS. A method is presented to find the most efficient pin assignments and pin mappings for Behavioral IPs (BIPs) to minimize the performance degradation introduced by having to multiplex the IOs and minimize the area increase due to the multiplexer logic. The proposed method is a fast heuristic based on the scheduling results of High-Level Synthesis seen as a black box and hence is flexible enough to work with any HLS tool. This method is also extended to deal with multiple BIPs directly connected to determine the optimal number of pins, and thus wires, between the two components to reduce the critical path delay introduced due to the interconnect. Experimental results show that our proposed method is very effective compared to an exhaustive search and a simulated annealing method at a fraction of the time, and much better than randomly selecting the pins to be multiplexed.

3.1 Introduction

The design complexity of ICs continues to increase, mainly because the number of transistors is still doubling every 18 to 24 months following Moore's law [75]. Current ICs are mainly heterogeneous Multi-Processor System-on-Chip (MPSoC) which contains multiple processor cores, memories, interfaces, and hardware accelerators. This increase in complexity has led to the adoption of new design methodologies. One paradigm shift which is slowly taking place is the use of higher levels of abstractions to

increase the design productivity. This increase in the level of abstraction is based on the use of HLS. HLS takes as the input a behavioral description and generates a RTL description which can be efficiently executed. HLS has the additional benefit that the RTL code generated by HLS can automatically include test circuitry, low power control circuitry (e.g., clock gating) and also automatically multiplex the input and output pins.

Pin multiplexing is extremely useful. At the chip level, it reduces the number of the chip's pins as the packaging cost can be as high as the die cost [5]. It is well known that the number of Input/Output (IO) pins is not growing fast enough compared to the increase in logic resources. Thus, one solution that has been adopted is time-division multiplexing of the IO pins to overcome this issue [56]. This is especially true in FPGAs, which are often used for emulation or prototyping. In general, every pin in a complex system is shared by an average of 30 IO signals [8]. Logic emulation and prototyping enable the functional verification of complex ICs before chip fabrication. However, traditional FPGA-based verification platforms have poor inter-chip communication bandwidth, limiting the gate utilization to less than 20% [3] in the past.

At the component level, pin multiplexing reduces the number of internal interconnects. This reduces the congestion of complex ICs and facilitates the routability while reducing the wire delay by reducing the fanout. Congestion can be formally defined as the ratio of routing demand to the available routing resources in each region of a design. Timing closure is still one of the most critical issues in state-of-the-art VLSI designs and is getting even harder at sub-micron technologies due to the importance of wire delays [4].

Raising the level of abstraction using HLS has one fundamental advantage over traditional RTL VLSI design. Different micro-architectures can be generated from an initial behavioral description. Each micro-architecture has the unique area vs. performance property, thus designs with different trade-offs can be automatically generated. This is impractical at the RT-level as it would imply having to manually re-write the RTL code to describe each of these micro-architectures. Pin-multiplexing affects how HLS synthesizes the micro-architecture as it requires internal logic resources to multiplex the pins. At the same time, the latency of the circuit increases with pin-multiplexing as data is read slower into it. Thus, it is essential to understand how pin multiplexing

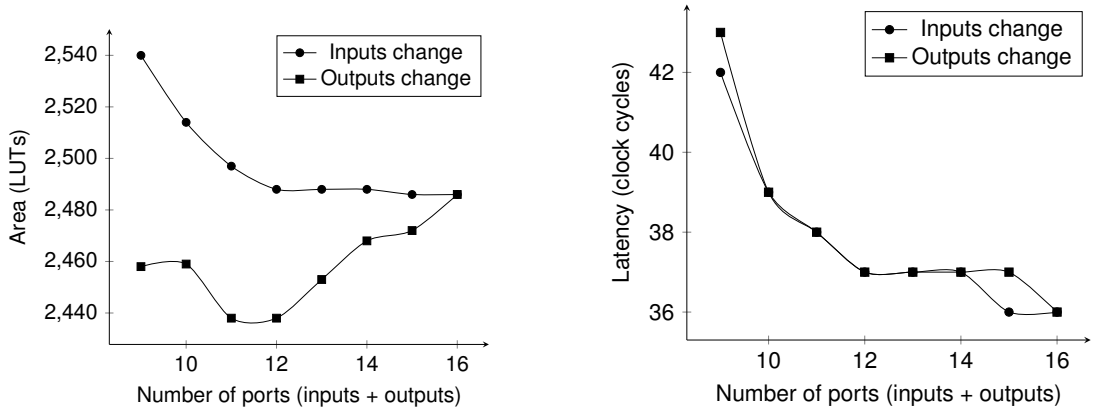
affects the synthesized design at the behavioral level and find efficient methods to map logic IO signals to physical ports. In particular, the main contributions of this work can be summarized as follows:

- Investigate the effect of pin multiplexing on area and latency overhead of a BIP.
- Propose a method to efficiently multiplex BIPs' pins using the scheduling result of HLS as its main port mapping criteria.
- Expand the method to deal with the delay optimization of multiple BIPs directly connected by determining the optimal number of pins/interconnects required.
- Perform comprehensive simulations to compare the quality of the proposed method against those of an exhaustive search method, a random assignment method, and a simulated annealer based method.

3.2 Motivational Example

In traditional VLSI design, during partitioning, the number of logic signals is often greater than that of physical ports; hence these have to be multiplexed. At the RT-level, this typically involves having to re-write the circuit description to accommodate this pin multiplexing. One of the advantages of C-based VLSI design based on HLS is that pin multiplexing can be automated. This is typically done by specifying as inputs to the HLS process a Port Constraint (PCNT) file and a Port Relation (PREL) file. The *PCNT* file contains the number of ports (the number of physical input and output ports) and the type (in, out, or inout) as well as the bit width of each port, and the *PREL* file specifies which logic signals should be mapped to the same physical port.

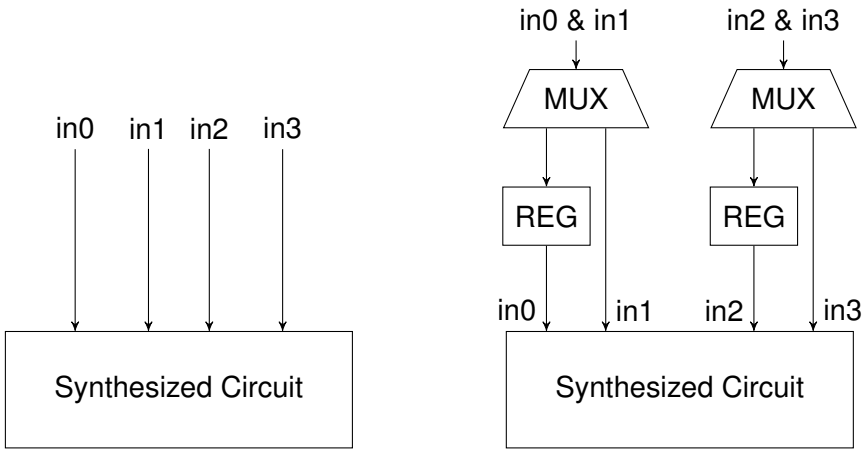
Fig. 3.1 shows a motivational example for this work for a simple design which computes the average of 8 numbers. Fig. 3.1(a) shows how the area in terms of LUTs of a BIP changes when its input or output signals are multiplexed. Originally the BIP is synthesized without any port constraint file (*PCNT* nor *PREL*), and allocates as many physical ports as logic signals. In this case, there are $m + n$ ($m = 8$, $n = 8$ in this example) physical ports. This configuration leads to the fastest performance (lowest latency) as shown in Fig. 3.1(b). When the number of physical ports decreases, the latency of the design increases as data requires longer to be read into and written



(a) Impact on area (b) Impact on latency

Figure 3.1: The impact of pin multiplexing on area and latency

out of the circuit. Fig. 3.1(b) shows how the latency increases when the physical input ports are decreased from *m* to a single one, while keeping the number of physical output ports constant to *n*, and it also shows the change when the physical output ports decrease from *n* to 1, while keeping the physical input ports constant to *m*. At the same time Fig. 3.1(a) shows how the same changes of physical port affect the area.



(a) Without pin multiplexing (b) With pin multiplexing

Figure 3.2: The impact of pin multiplexing on the micro-architecture

The main reason that the micro-architecture changes when the ports are multiplexed can be seen in Fig. 3.2. Without multiplexing, the circuit reads the four input signals in parallel and does not require any extra logic as shown in Fig. 3.2(a). However, Fig. 3.2(b) illustrates the extra logic inserted due to pin multiplexing. The synthesizer inserts *MUX* and inserts *REG* to hold values of the input or output signal that have

been read or written first. Moreover, because the latency of the synthesized circuit now changes, the HLS process can take advantage of these and create a different micro-architecture which uses fewer resources.

From these results, we can make two important observations:

Observation 1: Reducing the number of ports increases the latency of a design, hence degrades its performance. The amount of degradation nevertheless differs when reducing either the input ports or the output ports, and it is not monotonically increasing.

Observation 2: Reducing the input and output ports count impacts the area of a BIP differently. In this example, when the number of input ports is reduced, the area increases; while the area decreases when the number of output ports decreases. In both cases, this effect is also not monotonic. As for the behavior of other designs, sometimes, the area in both cases increases, but at different rates.

Based on these observations, the problem addressed in this work can be formulated as following:

Problem Formulation: Given a BIP in synthesizable C or C++ with m logic input signals (LS_{in}) and n logic output signals (LS_{out}), and a maximum number of physical ports (P_{max}), allocate the optimal number of physical ports for the inputs (P_{in}) and outputs (P_{out}), such that P_{max}=P_{in} + P_{out} and find the optimal mapping for each logic signal

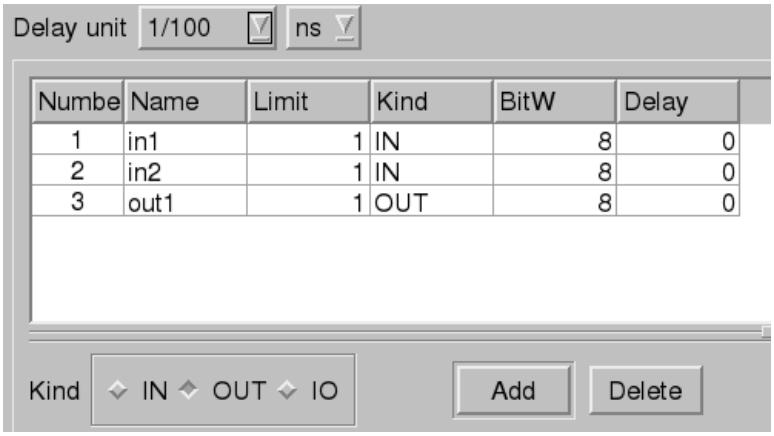
$$\{LS_{in1}, LS_{in2}, ..., LS_{inm}\} \rightarrow P_{in} \quad and \quad \{LS_{out1}, LS_{out2}, ..., LS_{outn}\} \rightarrow P_{out}$$

in order to minimize the area increase as well as the performance penalty (latency increase) of the BIP.

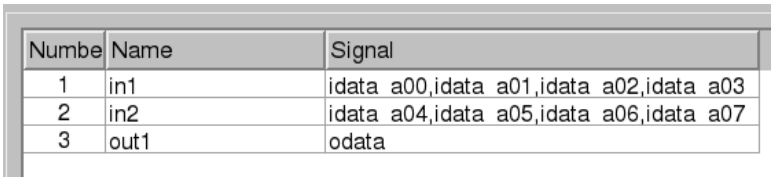
3.3 Pin Multiplexing in HLS

In this thesis, another widely used feature powered by HLS is pin multiplexing. It is a technique for mitigating the constraint of pin limitations on FPGAs. A pin is a physical I/O on an FPGA to communicate with the external environment. FPGAs are widely used for verification purpose of complex SoC designs, such as prototyping or logic emulation. An FPGA is usually not big enough to hold the entire SoC. Therefore, engi-

neers will partition the SoC onto multiple inter-connected FPGAs. However, the number of interconnections of two adjacent FPGAs may exceed the number of I/O pins, which means the original logic signals will be multiplexed onto the physical pins. In RTL, pin multiplexing indicates the manual modification of the source code, which may introduce extra effort and errors. In HLS, pin multiplexing can be performed automatically by specifying two pin constraint files: pin count constraint and pin relation constraint files. The HLS tool takes as input the two files and regenerates the micro-architecture without changing the source code. For example, Fig. 3.3 shows the two constraint files. There are eight input logic signals ($idata_{a00}, \cdots, idata_{a07}$) and one output logic signal



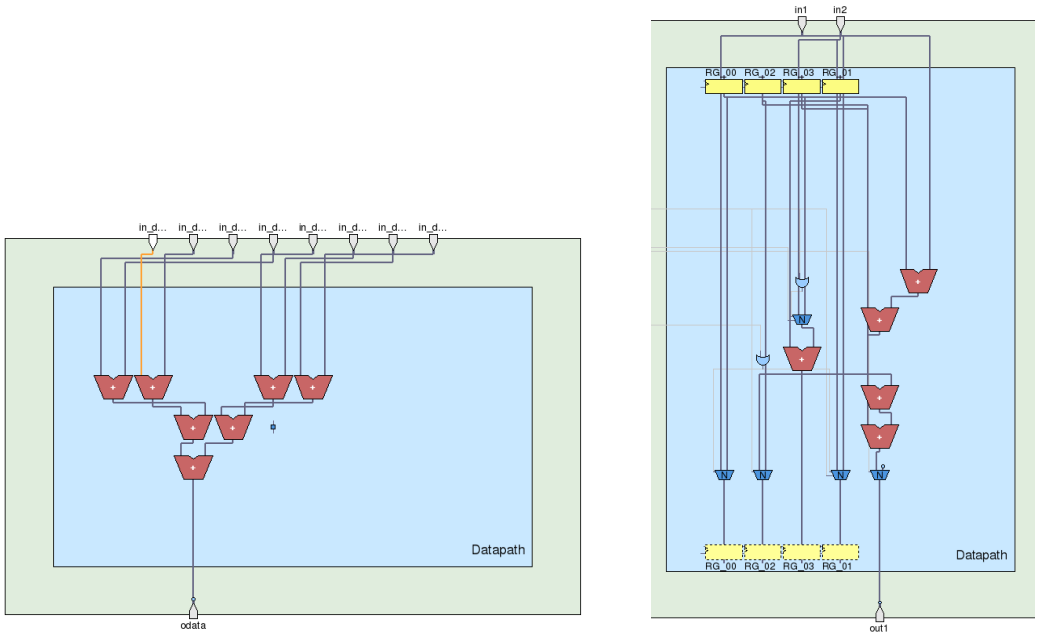
(a) pin count constraint



(b) pin relation constraint

Figure 3.3: The two pin constraint files

(*odata*). Fig. 3.3(a) specifies two input physical ports (*in*1 and *in*2), each of which has 8 bits; it also specifies an output physical port (*out*1) with 8-bit. Fig. 3.3(b) describes the relation between logic signals and physical ports. In this case, the first four signals share *in*1 and the other four share *in*2. There is only one output logic signal, so there is no need to share. Fig. 3.4 presents the datapath before and after applying pin multiplexing. Here the two circuits perform the functionality of average8, the only difference is the number of input signals has been changed to two (right) instead of eight (left). It is observed that the architecture without pin multiplexing is simple. However, after



(a) Datapath without pin multiplexing

(b) Datapath with pin multiplexing

Figure 3.4: The change of datapath after pin multiplexing for the average8 benchmark

applying pin constraints, only two values can be read at each clock cycle, so extra logic elements like multiplexers and registers are required to maintain the correct functionality. Meanwhile, since the circuit cannot read the inputs as fast as before, there is no need to consume the same amount of functional units. Thus the number of adders is reduced to five.

Pin multiplexing is an important technique in FPGA-based verification. The effect of applying it varies for different designs. The following chapters will demonstrate the utilization of pin multiplexing.

3.4 Related Work

Pin multiplexing methodologies have been widely studied in multi-FPGA systems, mainly due to their limited number of pins. In [3] the authors studied the dependency of input and output signals in a single FPGA at the RT-level to share a physical wire among several logic signals. The proposed methodology was applied to logic emulation in [109] and [25], where the authors proved that the FPGA utilization improved due to pin multiplexing.

Hauck et al. [40] proposed a pin assignment method for multi-FPGA systems based on a force-directed pin assignment solution and compared it with a random assignment, showing that it can lead to better results. The use of Input and Output Serialization and Deserialization (ISERDES/OSERDES) with Low Voltage Differential Signaling (LVDS) has shown to increase the inter-FPGA communication data rate by $10\times$ [107]. However, the complexity of this method is high, and it is only usable on boards which support high-speed serial connections.

Other automatic pin assignment methods at the floor planning level were presented in [93, 92]. In this work, the authors proposed a cost function of delay constraint to determine the shape and placement of each cell. In [69], the authors studied pin assignments at the PCB level to reduce total wire length.

In all previous work, the designs' architectures did not change as the pin multiplexing is done as a *wrapper* around the original RTL code.

This work is different from previous works as it addresses the issue of pin assignment at the behavioral level. In particular, it studies the effect of pin multiplexing on the resultant micro-architecture and proposes a fast method to bind logic ports to the same physical pins reducing the area and latency degradation due to these. One additional advantage of the proposed method is that it relies on a pre-characterization stage and also on the results of the HLS scheduling process, considering the HLS process as a black box. Hence, it is independent of the HLS tool used.

3.5 Proposed Method

This section describes in detail the proposed pin multiplexing method. It is based on two phases. The first phase, called port allocation, assigns physical ports to the inputs (P_{in}) and outputs (P_{out}), based on the maximum number of available ports(P_{max}). The second phase, called port binding, binds individual input logic signals (SL_{in}) and output logic signals (LS_{out}) to individual physical ports, minimizing the effect of area increase and performance degradation (latency increase).

Fig. 3.5 shows an overview of the complete flow, divided into the two aforementioned phases. The inputs to the proposed method are:

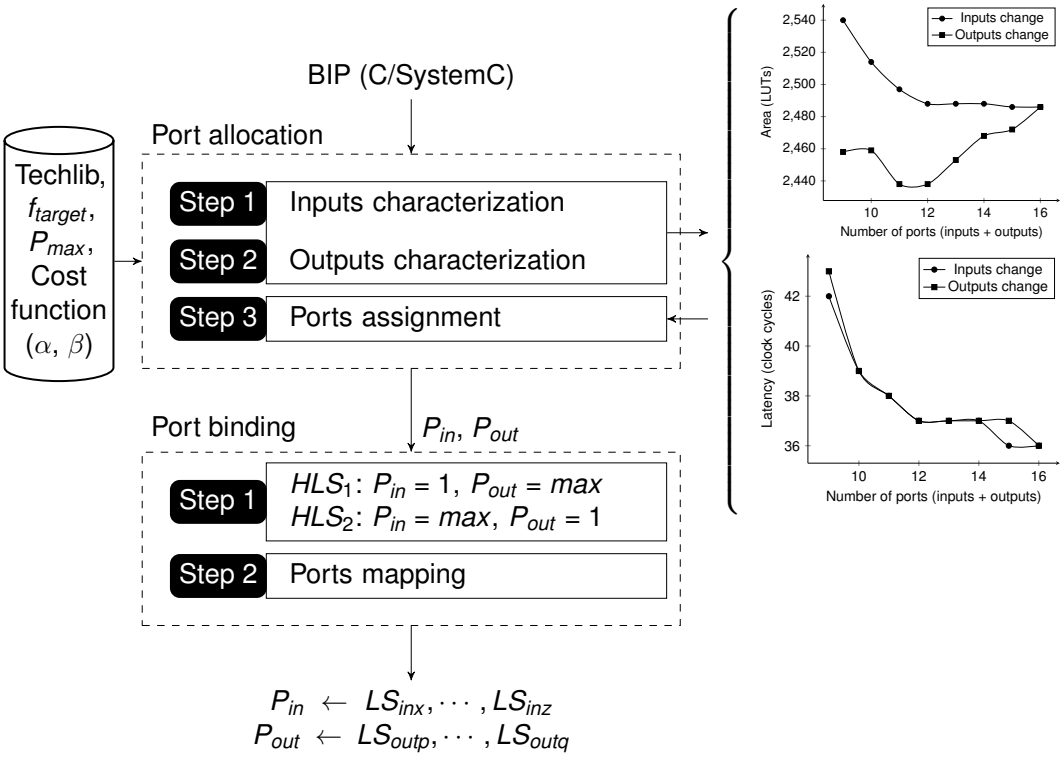


Figure 3.5: Overview of the proposed method

1. *BIP*: the behavioral description of the circuit in C/SystemC.
2. *Techlib*, f_{target}: the technology library and target frequency for HLS.
3. P_{max}: the number of total physical ports allowed, which has to be smaller than the sum of logic input and output signals, $P_{max} < LS_{in} + LS_{out}$.
4. *Cost function*: Cost function weights to determine if the area or latency degradation or any combination of both should be minimized.

The port allocation phase (phase 1) of the method is based on three steps as shown in Fig. 3.5. Step 1 and 2 *pre-characterize* the area and latency degradation of the inputs and outputs separately for a different number of input and output ports. Based on these results, step 3 allocates the available ports into inputs and outputs, thus $P_{max} = P_{in} + P_{out}$. Phase 2, in turn, is based on two steps. The first step forces the HLS tool to *reveal* the order in which each input and output is read/written by setting separately the number of input ports P_{in}=1 and then the output ports P_{out} = 1. Based on the scheduling result, step 2 assigns each logic signal to each physical port. As mentioned before, one of the advantages of this method is that it uses the HLS process

as a black box. Thus it has the flexibility to be used with any HLS tool.

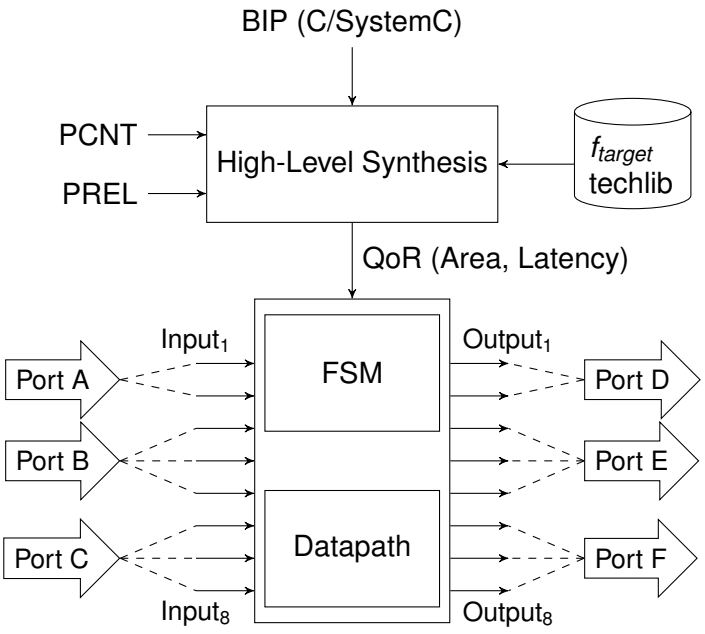


Figure 3.6: An example of port allocation and port binding

Fig. 3.6 shows a graphical example of how the two phases work. Given a maximum of 6 physical ports $P_{max} = Port\{A, B, C, D, E, F\}$, and given a *BIP* to be synthesized in HLS with $LS_{in} = m$ input signals ($Input_{\{1,2,\dots,8\}}$) and $LS_{out} = n$ output signals ($Output_{\{1,2,\dots,8\}}$), with m and n being both 8, the first phase determines the number of physical ports assigned as input and output. In the example in Fig. 3.6, three physical ports, Port A, Port B, and Port C, are assigned as inputs, and the other three physical ports are assigned as outputs. Thus fulfilling the condition that $P_{max} = P_{in} + P_{out}$. The second phase, then determines which of the logic signals should be assigned to the same physical port. In this example, Port A is shared by $Input_1$ and $Input_2$, Port B is shared by $Input_3$, $Input_4$ and $Input_5$, Port C is shared by the rest logic signals. Similarly, the assignment of the output is performed.

3.5.1 Phase 1: Port Allocation

The observations made in section 3.2, especially the second one, is used in this first allocation phase. As shown in the motivational example, assigning more ports to either inputs or outputs affects the area of the synthesized micro-architecture differently. Because this work relies on the synthesis result of an HLS tool seen as a black box, we

cannot determine a priori the synthesis optimization done by this tool in either case. Hence a *pre-characterization* stage is performed to determine the impact of reducing the input and output physical ports on the area and latency of the resultant micro-architecture. The outcome of this stage allocates the maximum number of ports (P_{max}) to inputs and outputs ports (P_{in} and P_{out}). This phase consists of three main steps, which can be summarized as follows:

Step 1: Inputs' Characterization. In order to understand how the port reduction affects the area and latency of the resultant micro-architecture, this first step keeps the number of physical output ports the same as the logic output signals ($P_{out} = LS_{out}$) and reduces sequentially the number of physical input ports (P_{in}), from their maximum number $P_{in} = LS_{in}$, leading to configurations with $(m, n), (m-1, n), (m-2, n), \ldots, (1, n)$ ports until a single physical input port is allowed, $P_{in} = 1$.

For each unique configuration, the HLS tool takes as additional inputs the port constraints, and the area (A) and latency (L) are extracted from the synthesis report. The result of this first step is similar to the example shown in Fig. 3.1 in section 3.2, fully characterizing the impact of time-multiplexing the logic input signals on the area and latency of the synthesized design. The outcome of this step leads to a design list $DL_{inputs} = \{(m, n, A_{m\_n}, L_{m\_n}), \ldots, (1, n, A_{1\_n}, L_{1\_n})\}$, where A_{m_n} and L_{m_n} are the area and latency of the synthesized design with m physical input ports and n physical output ports.

Step 2: Outputs' Characterization. The same *pre-characterization* performed in step 1 is repeated in this step for the outputs. This step leads to another design list $DL_{outputs} = \{(m, n, A_{m\_n}, L_{m\_n}), \ldots, (m, 1, A_{m\_1}, L_{m\_1})\}$. Hence after executing step 1 and step 2, the effect of time-multiplexing the input and output ports on the area and latency degradation is fully made visible.

Step 3: Port Count Assignment. Based on the results of steps 1 & 2, the effect on the area and latency of any port combination can be determined by combining the results of both steps. Here, we mainly investigate the relative performance of a design of a particular configuration. For example, in order to compare A_{2_6} and A_{3_5} without running HLS of these configurations, A_{2_6} is represented by A_{2_n} and A_{m_6}; similarly, A_{3_5} is represented by A_{3_n} and A_{m_5}. If $A_{2\_n} + A_{m\_6} > A_{3\_n} + A_{m\_5}$, then

A_{2_6} is greater than A_{3_5}, otherwise, $A_{2\_6} \leq A_{3\_5}$. This relation is consistently observed among multiple behavioral designs, the reason under the hood could be that the HLS tool handles the pin multiplexing of inputs and outputs separately, so that utilizing the above inequalities is able to reflect the difference of the configurations.

As mentioned in the motivational example, a different number of input and output ports affect either the area or latency degradation differently. Thus, one other input required by our method is a cost function with the optimization goal needed – either area, latency or a trade-off between area and latency degradation. For this purpose a cost function as follows is used: $C = \alpha A + \beta L$, where A is the area, L is the latency, α and β are the weights to determine if the assignment should minimize the total area or latency. Different weights would lead to different assignments. One option would be to execute the entire assignment process twice: once setting α=1, β=0 and once setting α=0, β=1. This would lead to the best area reduction and best latency reduction assignments, which should be very useful to designers as indicators of upper boundaries.

The result of this assignment is $P_{max} = P_{in} + P_{out}$, where P_{in} are the physical ports assigned to the inputs and P_{out} the physical ports assigned to the outputs. One advantage of using this *pre-characterization*-based method is that the complexity of synthesis iterations is linear $\mathcal{O}(m+n)$ with the number of logic input and output signals.

3.5.2 Phase 2: Port Binding

Once the maximum numbers of ports (P_{max}) have been allocated to inputs (P_{in}) and outputs (P_{out}) ports, the next step binds each logic signal (LS) to a specific physical port.

The number of possible bindings follows the Stirling numbers of the second kind [37]. The Stirling numbers of the second kind $S(I, k)$ counts the ways to divide a set of I objects into k nonempty subsets. In our case I is LS_{in} or LS_{out}, and $k = [1, P_{in}/P_{out}]$. When the design only has one input or output port ($k = 1$), only one mapping exists, which also leads to the slowest of all system configurations because all the logic signals have to be connected to that port. This case corresponds to $S(I, 1) = 1$. By increasing the number of physical ports, more binding combinations exist until $LS_{in}/2$

or $LS_{out}/2$, which has the largest number of binding combinations ($S(LS_{in}, LS_{in}/2)$ or $S(LS_{out}, LS_{out}/2)$). Finally increasing the number of physical ports until $P_{in} = LS_{in}$ or $P_{out} = LS_{out}$ leads again to a single binding as each logic signal is mapped to its own port, and this can be expressed as $S(LS_{in}, LS_{in}) = 1$ or $S(LS_{out}, LS_{out}) = 1$. This configuration also leads to the fastest design as no multiplexing is required. The numbers of mappings in each case can be calculated as [37]:

$$S(I,k) = \frac{1}{k!}\sum_{i=0}^{k}(-1)^{k-i}\begin{pmatrix} k \\ i \end{pmatrix} i^{I}$$

The number of possible combinations is a function of ${P_{in}}^{LS_{in}}$ or ${P_{out}}^{LS_{out}}$, which makes an exhaustive search not practical. It is because each time a new combination is generated, it has to be synthesized in order to obtain the area and latency of that particular assignment. Thus, faster heuristics are required.

The proposed IO to pin assignment method is based on forcing the HLS process to determine the priority of reading and writing data to the IO ports when only having a single physical port. Thus, we call this method **Priority IO Scheduling (PIOS)**. Intuitively, if a signal with lower priority is read first, the area and latency of the circuit will increase as this signal needs to be stored and cannot be used until the rest of the data is available. Moreover, the complexity of FSM will increase, further increasing the area and delay of the resultant circuit.

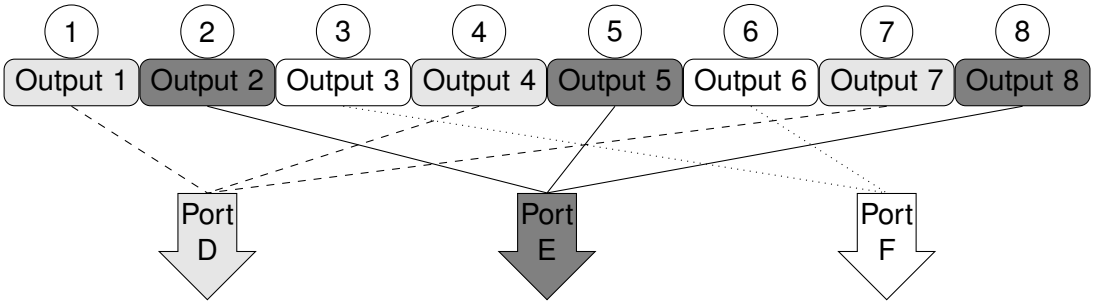


Figure 3.7: IO scheduling serialization overview

Fig. 3.7 shows an overview of the method and the algorithm is described in detail in Algorithm 1. The method can be decomposed into two main steps as follows.

Step 1: IO Serialization. The first step consists of assigning a single physical port to

Algorithm 1: Priority Input Output Scheduling Based Method (PIOS)

Input : BIP, P_{in}, P_{out}
$I = \{LS_{in1}, ..., LS_{inm}\}$, $O = \{LS_{out1}, ..., LS_{outn}\}$
// BIP: Behavioral IP in ANSI-C or SystemC
// P_{in}: Ports allocated as input
// P_{out}: Ports allocated as output
// I: Total logic inputs $\in$ BIP
// O: Total logic outputs $\in$ BIP
Output : $P_{in}(i) = \{LS_{inx}, ..., LS_{inz}\}$
$P_{out}(j) = \{LS_{outx}, ..., LS_{outz}\}$
// $P_{in}(i)$: Physical in port i with inputs' assignment
// $P_{out}(j)$: Physical out port j with outputs' assignment

```
1  P_out ← assign_max_ports(n);
2  P_in ← assign_single_port(1);
   // repeat for inputs and outputs separately
3  for I and O do
       // Step 1: IO port serialization
4      QOR(A, L, IO_schedule) = hls(P_in, P_out);
       // Step 2: Sequential Port Mapping
5      foreach P_in do
6          for (PL_i = IO_schedule; i < m; i++) do
7              P_in(i%P_in) ← PL_i;
8          end
9      end
       // Update ports and re-do for outputs
10     P_out ← assign_single_port(1);
11     P_in ← assign_max_ports(m);
12 end
13 return P_in(i) = {LS_inx, ..., LS_inz}, P_out(j) = {LS_outx, ..., LS_outz}
```

the inputs, hence P_{in}=1 and as many physical output ports as logic signals (line 1-2), thus $P_{out} = LS_{out}$ and synthesizing (HLS) this configuration (line 4). The HLS tool is therefore forced to schedule the inputs sequentially, serializing how data is being read into the design as there is only a single input port (only one signal can be read in a clock cycle). The key to this method is to rely on the HLS tool to schedule reading the inputs based on their usage priority. As mentioned before this has the additional benefit of making our proposed method HLS tool agnostic. The same operation is repeated for the outputs setting with P_{out}=1 and $P_{in} = LS_{in}$ (line 10-11). Thus, the result of this step is two sequences of IO accesses (*IO_schedule*), one for the inputs and one for the outputs. Fig. 3.7 shows an exemplary access sequence of output signals, where each output is being accessed in a different state (clock cycle).

Step 2: Port Mapping. Based on the results obtained in step 1, the method continues by directly assigning the logic signals sequentially as shown in Fig. 3.7. In this case, 8 logic output signals are assigned to 3 physical ports. Because the results of step 1 reported the output scheduling order as Output{1,2,3,4,5,6,7,8}, this also indicates the priority in which the outputs have to be written out and hence triplets are formed greedily (lines 6-8). In this case {(1,4,7),(2,5,8),(3,6)}, meaning that Output{1,4,7} are mapped onto the same physical port, Output{2,5,8} to another and Output{3,6} to the last one. This ensures that logic signals 1, 2 and 3 are written out first as these have also been scheduled first in the priority scheduling list from step 1.

The advantage of this method is that it is extremely fast as it only requires the generation of two configurations independently of the number of signals or ports. Hence the order of complexity in terms of synthesis iterations is $\mathcal{O}(1)$.

3.6 Experiments

This section performs multiple experiments using the proposed method to study its robustness compared to other methods. The experimental setup will be described first, continued by the presentation of the experimental results. These results are split into two parts. The first analyzes the effectiveness of our method for individual, isolated BIP, while the second presents a case study when two BIPs are connected directly together. This case study highlights the impact of pin multiplexing on the wire delay and extends the proposed method to take into consideration the wire delay minimization.

3.6.1 Experiment Setup

Five benchmarks taken from the open source Synthesizable SystemC Benchmark Suite (S2CBench) [101] were used to measure the effectiveness of our proposed method. For example, as shown in Table 3.1, *idct* has 22 input pins and 15 output pins. To study the scalability of our method, two larger complex benchmarks were created by grouping together several of these benchmarks as shown in Table 3.1, e.g., S1 is composed of idct, kasumi, snow3G, and sobel, and each of them is instantiated as a function of a top module. Hence, all of the benchmarks were synthesized and optimized

together.

Table 3.1: IO information

	#in	#out	#total	S1	S2
idct	22	15	37	1	
interpolation	25	17	42		1
kasumi	64	64	128	1	
snow3G	128	128	256	1	1
sobel	24	8	32	1	1
S1	238	215	453		
S2	177	153	330		

The size of these complex benchmarks is larger than the sum of the individual benchmarks as often these complex benchmarks are composed of multiple concurrent processes. In the single benchmark case, only the largest process is used, while for the complex benchmarks all processes are considered.

The experiments were run on an Intel Pentium 4 running at a 3.20GHz machine with 8 GBytes of RAM running Linux SUSE version 3.0.13-0.27. The HLS tool used in this work is CyberWorkBench (CWB) v5.5 from NEC [78] and the target HLS frequency set to 100MHz. The target FPGA is a Xilinx XC5VLX330 Virtex 5 FPGA.

To measure the qualitative and quantitative effectiveness of our proposed method, it is compared against an Exhaustive Search (ES) method, a random method (*RAND*) and a Simulated Annealing (SA) method. *ES* can guarantee the optimal global result as it explores the entire searching space. *RAND* performs random selections of both port allocation and port binding. This is the fastest method as it does not require the computation of the assignment effect on the resultant design. Thus, it could be considered the naïve default method. *SA* based methods have shown to lead to good results in multi-objective optimization problems like this one and hence should lead to a baseline method with good trade-offs between running time and quality of results. Due to the probabilistic nature of the *RAND* and *SA* methods, these are executed five times with different seeds, and the best results are reported.

Because of the conflicting minimizing objectives of reducing area, while also minimizing the latency penalty, the experiments were executed twice. Once with the target

cost function weights were set to α=1.0 and β=0 to find the very smallest configuration (optimized for area) and once setting α=0 and β=1.0, optimizing for latency. 3 different P_{max} values were used to test our methods. P_{max}=25%, 50% and 75% of the total number of logic signals (LS_{in} + LS_{out}).

3.6.2 Experimental Results

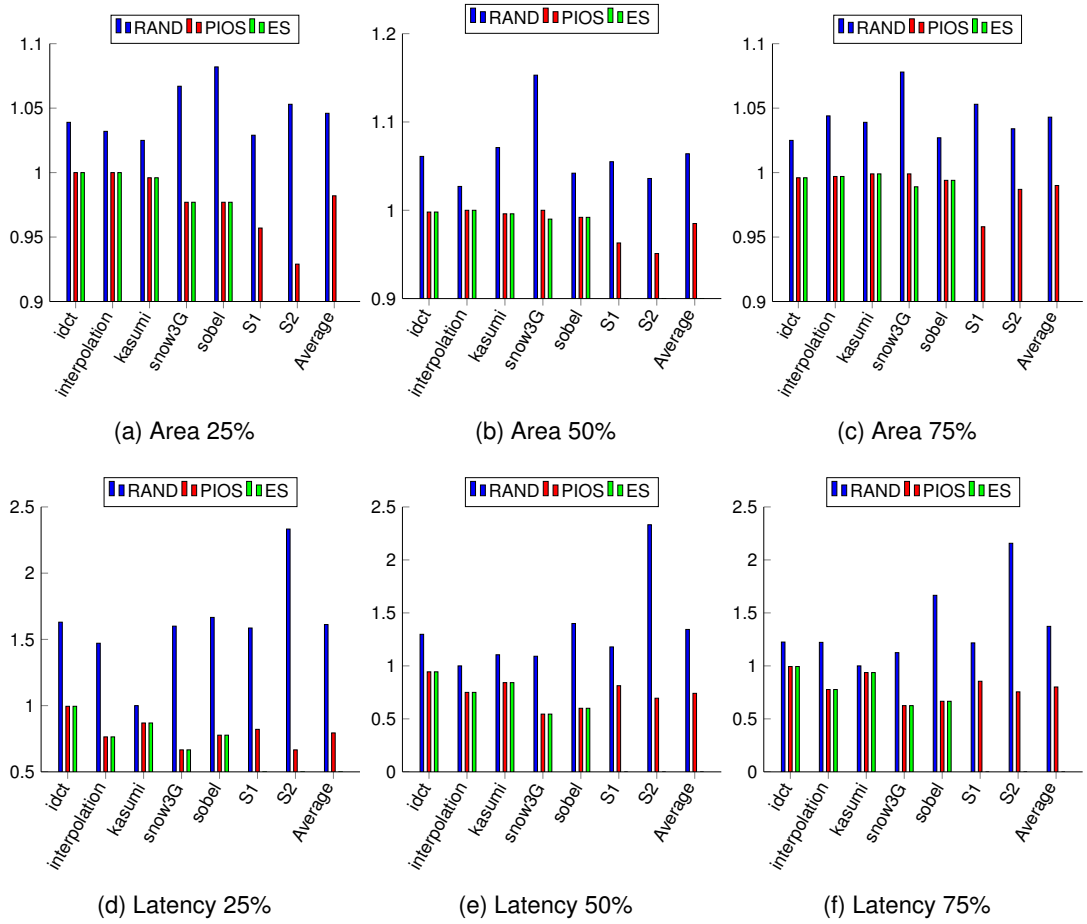
Individual BIP

Fig. 3.8 shows the normalized results of the four methods for the cases which target the optimization of area (α=1.0, β=0) and latency (α=0, β=1.0). In both cases, 25%, 50% and 75% of the original number of ports are used. The vertical axis indicates the normalized area or latency, hence the smaller, the better. Since *SA* is set as a reference, its value is "1" in the figures, and thus only *ES*, *RAND* and *PIOS* are presented. Besides, some of the results of *ES* (e.g., S1 in Fig. 3.8(a)) are not presented since their running time exceeds seven days. Therefore, the average values of *ES* are not shown either.

For the area driven optimization (Fig. 3.8(a), 3.8(b), 3.8(c)), our proposed method (*PIOS*) performs very close to *ES* except for the snow3G case shown in Fig. 3.8(b) and 3.8(c). In both cases, the results of *PIOS* are about 1% bigger than those of *ES*. When comparing against *SA*, our method is up to 2.3% better and on average 0.5% better for the five basic benchmarks, but much better for the larger cases due to the increase of the searching space. For these two complex benchmarks, our method is up to 7.1% better and on average 4.3% better.

Due to the simplistic nature of the *RAND* method, the results as expected are the worst of all the methods. In this respect, our method is up to 15.3% better and on average 6.3% better.

When optimizing for latency (Fig. 3.8(d), 3.8(e) and 3.8(f)), the cost function is set to α=0, β=1.0, and our method also shows good results. The latency improvements introduced by *PIOS* are up to 45.5% and 71.5%, and on average 22.1% and 41.8% compared with *SA* and *RAND*, respectively. Moreover, our method shows the same behavior as the optimal solution (*ES*) because *PIOS* leverages the scheduling data from the HLS tool to minimize the latency penalty.



(a) Area 25% (b) Area 50% (c) Area 75%

(d) Latency 25% (e) Latency 50% (f) Latency 75%

Figure 3.8: The results of area and latency in different cases

In terms of running time, Table 3.2 shows the comparison of the four methods for the three cases as mentioned earlier. The time measurement unit is second. The running time of *ES* for benchmarks S1 and S2 took more than seven days, and thus those results are not presented. Comparing *PIOS vs* both *SA* and *ES* shows that our method is much faster. To account for the size differences between different benchmarks, the geometric mean is given as an average indicator; however, the geometric means of *ES* are not calculated to maintain the equity as the two sets of longest running time are not included. An average speedup of 20, 13 and 15 is achieved by our method compared with *SA* for the 25%, 50%, and 75% cases, respectively. Obviously, *RAND* is the fastest among the four methods since only two synthesis iterations are needed, one for I/O data collection and one for the random assignment.

It should be noted that the proposed method is based on a *pre-characterization* stage. This stage only needs to be executed once to characterize each BIP fully, and

Table 3.2: Running time [s]

	ES			*RAND*			*SA*			*PIOS*		
	25%	50%	75%	25%	50%	75%	25%	50%	75%	25%	50%	75%
idct	134	276	159	16	15	15	114	191	145	82	148	110
interpolation	163	230	181	7	7	7	201	231	184	24	45	32
kasumi	163076	196458	103855	20	20	20	939	1200	1030	46	96	71
snow3G	120384	182519	166831	6	6	6	1071	1136	950	23	47	29
sobel	111	218	132	5	5	5	185	197	156	16	19	19
S1				58	57	57	81615	76114	77846	1101	1814	1214
S2				18	18	18	28976	27363	24888	224	362	250
Geomean				13	9	13	1391	1581	1343	70	121	88

thus our proposed method can be much more effective when re-using the BIP in future projects. Table 3.3 shows the running time of our proposed method excluding the *pre-characterization* phase. Compared to the results in Table 3.2 in terms of *Geomean*, our proposed method is at least 10$\times$ faster, which means that our two-phase approach is very efficient in such practical situations.

Table 3.3: Running time without pre-characterization [s]

	25%	50%	75%
idct	8	7	7
interpolation	2	2	2
kasumi	9	9	9
snow3G	2	2	2
sobel	3	3	3
S1	52	50	50
S2	16	32	17
Geomean	6.9	7.4	6.7

Based on the experimental results obtained it is safe to conclude that our method works well, and it is a good compromise between quality of results and running time.

Case Study: Multiple BIPs

To further demonstrate the effectiveness of our proposed method, a case study when two BIPs are connected is presented. The main purpose of pin multiplexing, in this case, is to reduce the congestion and wire delay, while again minimizing the area and/or

latency degradation. In particular, the system of this case study is composed of a *sobel* filter which performs an edge-detection algorithm on an image, directly connected to a 128-bit AES encryption block for secure transmission as shown in Fig 3.9. The total input signal bit width of AES is 128, which means that there are 128 wires connecting *sobel* and *AES* for data communication. These larger number of connections can lead to congestion and especially wire delay problems. Thus, our proposed method was applied to this case study.

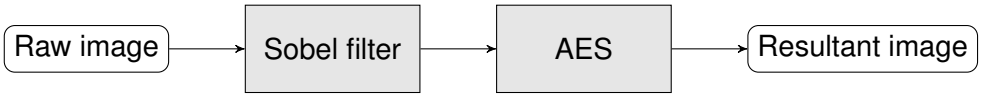


Figure 3.9: A system for case study

The main problem, not treated by our proposed method in cases like this, is to determine the total number of physical interconnections allowed, where interconnections are the same as pins in this case. Once this is determined, our method can be applied by fixing the number of inputs and outputs of one component. This fixes the ports of the other one automatically, as these are directly connected. In this case, the number of ports assigned as outputs to the sobel component has to be equal to the number of input ports of the AES component, $P_{out}(sobel) = P_{in}(AES)$.

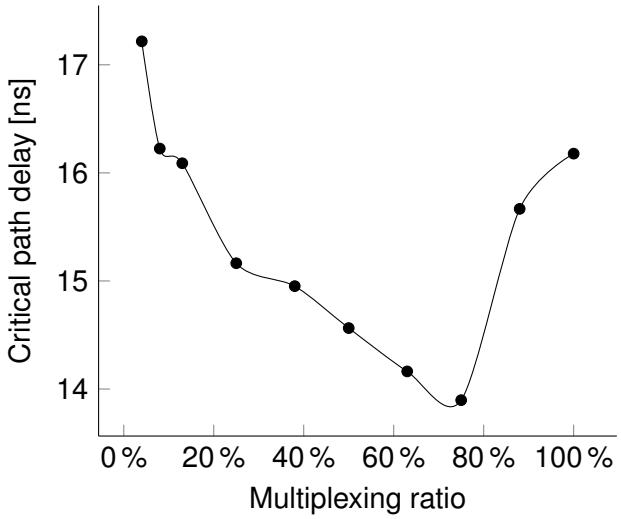


Figure 3.10: Critical path delay vs multiplexing ratio of the case study

To solve this problem, we perform some preliminary experiments on the effect of different level of pin multiplexing on the total circuit delay. Fig. 3.10 shows the results

obtained. It can be observed that the initial delay is largely due to the wire delays. The delay then falls until a global minimum value and then grows again when the multiplexing ratio falls (more pins are multiplexed, as the multiplexing ratio is defined as used pins over the total number of ports). This can be explained as initially, the wire delay dominates the total delay. Then, once the pins started to be multiplexed, the wire delay becomes smaller until the delay of the muxes, which are required to multiplex the pins, starts to outweigh the delay savings of the wire. Thus, there is a pin multiplexing factor which leads to the smallest overall delay. To find this multiplexing ratio, a binary search method is introduced. Algorithm 2 summarizes it. It is based on a well-known binary search. Because the wire delay needs to be extracted, the search has to perform a full place and route (*P&R*) on the entire system, which can be costly in terms of running time. On the other hand, the advantage of this method is that it is extremely fast, with an order to the complexity of $\mathcal{O}(log(n))$, where n is the number of possible pin multiplexing ratios.

Algorithm 2: Binary search to find optimal number of pins for directly connected components

Input : S, TechLib ,f_{target}
// S: Source code of System in synthesizable C
// TechLib: Source code ANN training in SystemC
// f_{target}: Target HLS frequency
Output : Pin multiplex ratio
// *Pins*: Total number of pins/wires which lead to smallest system delay

```
1  while (dly_new < dly_prev) do
2  |  rtl_new = HLS(C, Techlib, f_target);
3  |  dly_new = LS_PAR(rtl_new);
4  |  if (dly_new < dly_prev) then
5  |  |  incr_mux_ratio();
6  |  end
7  |  else
8  |  |  decr_mux_ratio();
9  |  end
10 end
11 return optimal_mux_ratio
```

The results of applying the search on this case study are shown in Table 3.4. The complexity of the routing was reduced when comparing the reported average fanout

from an average of 5.02 to 4.85 (3.4%). Additionally, the two largest critical-path (CP) delays of each case associated with the detailed logic and route delay were compared. An improvement of 9% could be observed.

Table 3.4: Results of case study

	Slices	Average fanout	CP delay [ns]	Logic [ns]	Route [ns]
Before	1139	5.02	16.178	1.511	14.667
			16.175	1.511	14.664
After	1057	4.93	13.439	1.354	12.085
			13.431	1.357	12.074

These results further demonstrate that our proposed method is extremely efficient not only when considering single isolated BIP, but multiple BIPs connected, taken into account not only the area and latency changes when multiplexing pins but also the interconnect delay.

3.7 Summary

In this work, we have studied the effect of pin multiplexing on single component's micro-architectures in HLS and presented a method to optimize the port allocation and port binding problems. It has been observed that the area and latency of the synthesized circuit change when assigning a different number of physical ports to the inputs or outputs. The proposed method is based on two main phases. The first one performs a *pre-characterization* stage which assigns the optimal number of physical ports to input and output ports. The second one determines which physical port is shared by certain logic signals. The proposed optimization method makes use of the scheduling results of the HLS process by *serializing* the IO timing and extracting the order in which the IOs are accessed. The proposed method has shown to produce outstanding results compared with an exhaustive search method, a simulated annealing method and a naïve random selection approach. Finally, the effect of pin multiplexing on delay and congestion of a system composed of multiple components connected directly was studied, and a search method to determine the optimal number of pins allowed was

introduced.

Chapter 4

Interconnect-aware Dataflow Implementation on FPGAs

The interconnect is the bottleneck of FPGAs. It has significant effects on the area, delay, and power consumption. Meanwhile, the synthesis may fail due to routing congestion, which is caused by excessive inter- or intra-connections on the limited hardware resources. Therefore it is essential to investigate design strategies that facilitate the routability of FPGAs. This chapter considers this case, especially in the context of complex dataflow systems mapped onto an FPGA.

One important feature of dataflow systems is the inter-module connections that may affect the area and performance as shown in the previous chapter. This chapter aims to optimize these interconnections regarding the conflicting objectives of area, latency, and delay. The results are thus, a set of optimal configurations that can be plotted as a trade-off curve.

The pin multiplexing technique described in the previous chapter will be used to generate these optimal configurations. To accelerate the optimization, this work also uses machine learning to skip the unnecessary synthesis processes. Experimental results show that the proposed method is efficient and accurate compared to an exhaustive search and the other state-of-the-art approaches.

4.1 Introduction

FPGAs continue to benefit from Moore's law by increasing their logic densities to a point where complete systems can now be integrated into a single device with minimal

off-chip resources. They also benefit from the breakdown of Dennard's scaling, as computationally intensive applications with inherent parallelism now have to be offloaded to dedicated hardware modules to accelerate their computation, while reducing the dynamic power by reducing the overall operating frequency. Moreover, they continue to benefit from their flexibility to be reconfigured in the field. Nevertheless, this flexibility also carries a heavy penalty regarding area overhead and performance degradation due to the flexible interconnect. The interconnect is also currently the culprit for most of the dynamic power consumption.

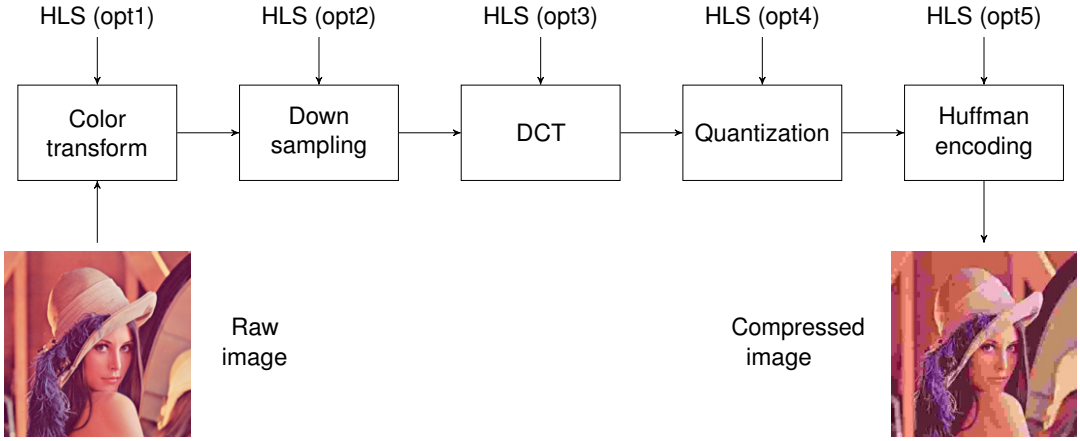


Figure 4.1: JPEG encoder block diagram

With the increase as mentioned above in logic density, a single FPGA can now hold complete systems. A typical system is shown in Fig. 4.1, which shows the block diagram of a JPEG encoder. These systems typically connect multiple components directly to take benefits of the underlying parallelism of these blocks and thus further speed up the computation, instead of traditional shared bus systems found in SoCs (at least for the accelerated hardware part of the system). This type of configuration is also referred to as *data-flow* computation or *stream computing*.

FPGA vendors have also embraced HLS as a methodology to program their FPGAs as this enables engineers with limited hardware development skills to be able to program FPGAs. Thus, HLS seems a natural design methodology. HLS takes as input a behavioral description written in C/SystemC/OpenCL and generates *efficient* RTL code which can execute it. HLS is also a single process design method, where each block in the system is synthesized, and thus, optimized separately. As shown in Fig. 4.1, each

individual block has to be synthesized separately with its own set of synthesis options ($\{opt1, opt2, ..., opt5\}$). This style leads to inevitable global inefficiencies. In contrast, one significant advantage of HLS is that state-of-the-art HLS tools allow to automatically pin-multiplex their ports. Typically, this involves specifying a Pin Count Constraint (PCC) that defines the maximum number of input and output pins allowed. The HLS tool then automatically assigns the logic IO ports to specific pins by time-multiplexing these. An additional advantage is that the synthesis process will, in turn, optimize the internal micro-architecture based on these constraints. For instance, a loop is fully unrolled if each IO port is assigned to its pin, but only partially unrolled if logic ports have to share physical pins, as it now takes multiple clock cycles to read the inputs and write the outputs. The first case will lead to a fast micro-architecture which requires more resources as the loop is fully unrolled, while the second micro-architecture will be smaller, using fewer resources but also slower. Other pin configurations lead to intermediate results. Thus micro-architectures with different area vs. performance can be obtained automatically. When combined with all the various components forming a data-flow system, this leads to configurations with various area vs. performance trade-offs, which this work aims at exploring.

In summary, the challenges that this work addresses and its main contributions are:

- Study the impact of pin-multiplexing on the area and throughput of FPGA designs composed of multiple components directly connected, also called data-flow computation or stream computing.
- Introduces a learning-based method to explore the search space, taking the overall interconnect overhead into consideration, by modifying the pin multiplexing ratios between the different components, given only the behavioral descriptions for each component in the system.
- Proposes a method to reduce the number of samples required to create the machine learning method called Adaptive Samples Selection Filter (*ASSF*).

4.2 Motivation

The following motivational example illustrates the challenges that this work addresses. In this case, a two-component system directly connected through 128 wires is presented (16 ports of 8 bits). In particular, a stream cipher (snow3G) connected to a post-processing cipher used in mobile communication systems (kasumi) is shown in Fig. 4.2(a). The following metrics are defined first:

Multiplexing Ratio (MuxR): Number of physical pins (*Pin*) available compared to the total number of logical ports (*Ports*). A larger *MuxR* implies more *Pins* and thus, more interconnect wires, with $MuxR \in (0, 1]$

Critical Path (CP): Longest combinational logic delay, which determines the maximum operating frequency ($f_{max} = 1/CP$) of the circuit.

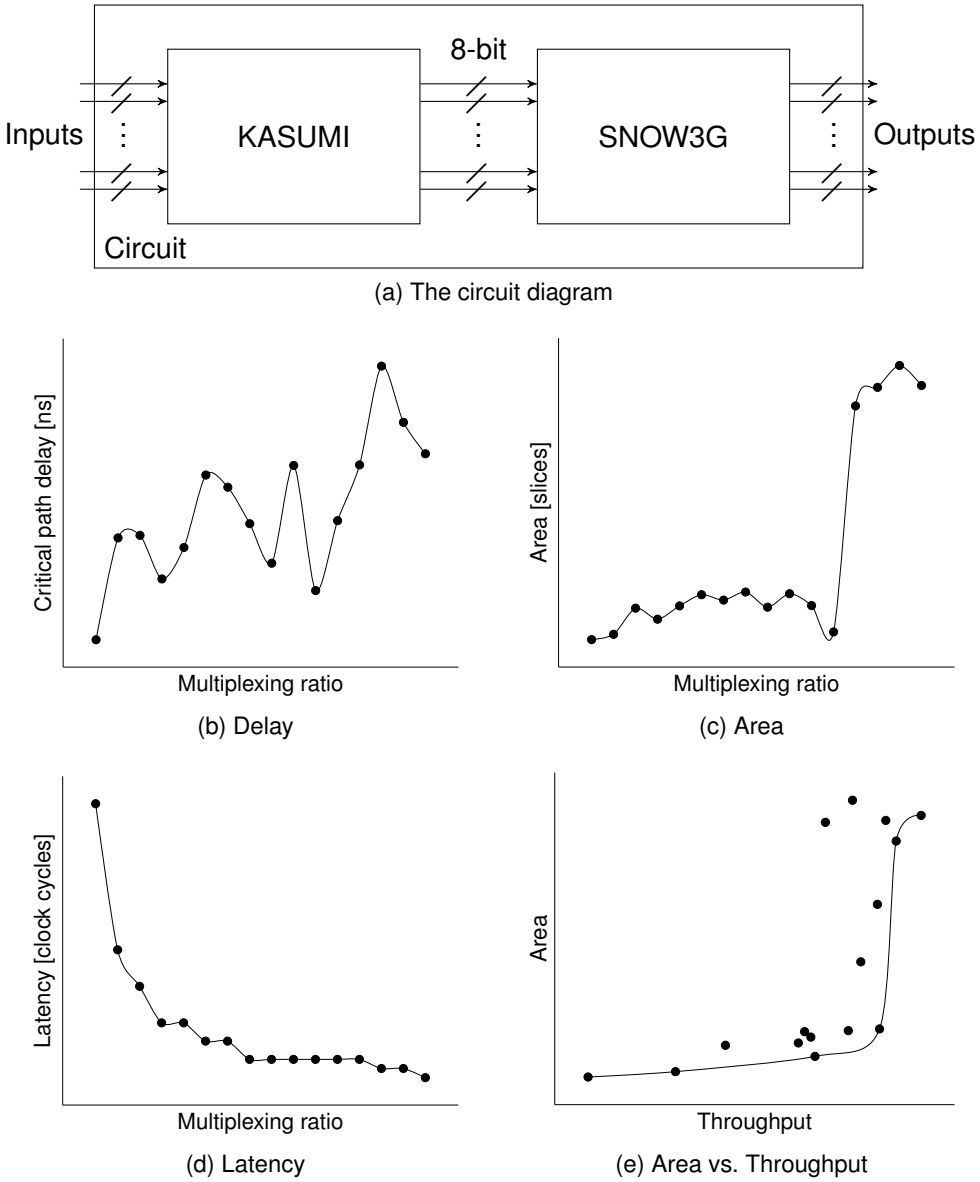
Latency (L): Number of clock cycles required by a circuit to read input and generate the corresponding output.

Throughput (T): Performance metric used in this work, $T = f_{max}/L$ (in case of multiple latencies the largest is used to measure the overall system's throughput)

Figs. 4.2(b), 4.2(c), 4.2(d) shows how the critical path (*CP*), Area (*A*) and latency (*L*) behave as a function of the multiplexing ratio (*MuxR*), where the *MuxR* is modified during HLS by setting different Pin Count Constraints (PCC). The results shown are for all cases after place and route (P&R) and hence, take into consideration the interconnect delay and area. The following observations can be made based on these results, which we have also observed for other systems:

Observation 1: From Fig. 4.2(b) it can be observed that in general critical path delay decreases with the multiplexing ratio (although not monotone). This is mainly because as the interconnects (wires) between components get reduced, the interconnect delay also gets smaller. At the same time, multiplexers are needed to multiplex the physical pins, which in turn contribute to the increase in delay. This explains the non-monotone behavior.

Observation 2: Fig. 4.2(c) shows that the area gets smaller when the multiplexing ratio decreases. This phenomenon is mainly due to the HLS processes, which create a micro-architecture that is adapted to the physical pins available. The fewer pins, the



(a) The circuit diagram

(b) Delay

(c) Area

(d) Latency

(e) Area vs. Throughput

Figure 4.2: Motivational example: KASUMI-SNOW3G, and the changes of delay, area, and latency when tuning the interconnections

slower the circuit becomes (larger latency) as shown in Fig. 4.2(d) and hence fewer hardware resources (e.g., less functional units are required).

Finally, Fig 4.2(e) shows the resultant trade-off curve that is obtained by modifying the number of physical pins between the two components leading to unique area vs. throughput configurations. The configurations with a larger number of pins typically lead to a higher area, but also higher throughputs, while reducing the number of pins, reduces the throughput as now data takes longer to be processed, but also lowers the system's area. According to the above observations, the problem that this paper

addresses can be formulated as:

Problem Formulation

Given a system consisting of multiple blocks[1] (b), $System = \{b_1, b_2, ..., b_n\}$, with each b_i specified in synthesizable C/SystemC ($Code$), thus, $b_i \rightarrow Code_i$, and with these blocks directly connected, find the optimal multiplexing ratios ($MuxR$) between every two adjacent blocks (b_i and b_{i+1}), which lead to the best area (A) vs. throughput (T) trade-off. Because the problem to be solved is a multi-objective optimization problem, and as shown in Fig 4.2(e), there is no unique optimal solution, but a set of optimal solutions called Pareto-optimal configurations (PO). Thus, the result of this work is a set of optimal configurations leading to a trade-off curve $PO = \{P_1(MuxR_1) = (A_1, T_1), P_2(MuxR_2) = (A_2, T_2), ..., P_i(MuxR_i) = (A_i, T_i), ...\}$, with P_i being a single Pareto-optimal configuration.

4.3 Related Work

Most previous work related to interconnect-aware synthesis mainly focuses on improving the performance/reduce the delay or reduce the power consumption by proposing different optimizations directly at the physical level, where the interconnect is fully visible (design has been fully P&R). For instance in [103], the authors proposed a routing architecture by mixing buffers and pass transistors, and claimed that the routing delay can be on average reduced by 50%. Besides, in [61], the authors reduce the critical path delay by 12.3% when taking the interconnect delay effects and cell congestion into account between mapping and placement stages.

Closer to this work is previous work in the area of physical-aware behavioral synthesis. The primary purpose so far has been to achieve timing closure. Recently the authors in [125] presented an iterative physical-aware HLS flow. This work back-annotates the delays after P&R into the HLS process to improve the quality of the synthesis. Initial efforts to integrate physical information to the behavioral synthesis process were made in [120], where the authors modeled the physical information through analytical models to speed the process up, which leads to inaccuracies. Physical-aware HLS process

[1] This work makes indistinguishable use of the terms block, component or process to refer to a single component in the dataflow hardware accelerator.

was also presented in [21], where placement-driven scheduling and binding algorithms were presented for multi-cycle communication architectures. This work nevertheless focuses on communication optimization. Other work uses the placement of block-level macros to guide the scheduling and binding refinement [98]. The main problem with this approach is that it allows optimizations neither within macros nor across. Moreover, it does not provide detailed routing information. In the ASIC domain, the tight integration between behavioral synthesis and physical design has been more widely studied. Several ASIC HLS flows have integrated floorplanning to estimate wire delays to improve scheduling and binding operations [95, 123] iteratively and more recently in [64], including variation-aware synthesis. The main problem with all these previous works is that, as mentioned before, HLS is a single process synthesis method. Thus all previous works ignore global intra-process optimizations, and in particular, the effect of the interconnect across modules. This work is therefore very different from these previous works and is completely orthogonal to it. To the best of our knowledge, it is the first work that aims at studying the effect of pin-multiplexing on the area and throughput of data-flow hardware systems mapped onto FPGAs.

4.4 Methodology

The proposed method is composed of two main phases. The first phase is a *pre-characterization* phase that fully characterizes the area, latency, and delay of each component in the system. An initial predictive model to estimate the total area and throughput of the complete system is created based on these preliminary results. Thus, the results of this first phase are two predictive models (one for the area and one for the throughput) which are not very accurate as they do not consider the interconnect delay between the components. The second phase then generates a set of complete system configurations to fine-tune the predictive models generated in the previous phase. This phase iterates through the model generation by intelligently adding samples that improve the quality of the predictive model. We call this method Adaptive Samples Selection Filter (*ASSF*). The model refinement loop exits when the method does not identify any new sample that could potentially lead to any new *PO* configuration. Details

are shown below.

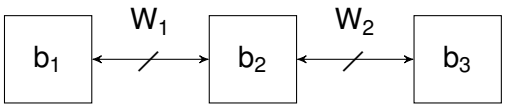


Figure 4.3: A circuit consists of three blocks

4.4.1 Phase 1: Pre-characterization

This first phase fully characterizes the effect of pin multiplexing on each component and creates an initial predictive model to estimate the area and throughput of the complete system. This phase can be further subdivided into two steps:

Step 1–Individual Component Characterization: This first step characterizes the area and throughput of individual components/blocks in the system, where each component can be defined as a process in the system, which is synthesized (HLS) separately with its own set of constraints. A block, with m input and n output ports, leads to the pin configuration of (m, n) without multiplexing. Thus, different configurations are created by either reducing the number of input or output pins sequentially, creating $C(m, n)$ configurations, which implies that $|C(m, n)| = m + n - 1$ configurations are required to characterize this component, as follows:

$$\begin{aligned} C(m,n) = \{&(m,n),(m-1,n),\dots,(1,n),\\ &(m,n-1),(m,n-2),\dots,(m,1)\} \end{aligned} \tag{4.1}$$

Therefore, for the circuit (shown in Fig. 4.3) which consists of three components (b_1, b_2, b_3), the *pre-characterization* for b_2 requires the following full synthesis and P&R:

$$\begin{aligned} b2(C(w_1,w_2)) = \{&\\ &b2(w_1,w_2), b2(w_1-1,w_2),\dots,b2(1,w_2),\\ &b2(w_1,w_2-1), b2(w_1,w_2-2),\dots,b2(w_1,1)\} \end{aligned} \tag{4.2}$$

In the $b2(w_1, w_2)$, w_1 is the maximum number of input pins and w_2 is the maximum

number of output pins of block b_2 required to map each logic IO port to its own IO pin. For the primary input and output components (i.e., b_1 and b_3), only the different configurations for the output ports and input pins, respectively, need to be generated to characterize them fully.

Thus, the running time required to characterize a system of 3 components is given by:

$$Cost_{pre} = t_{b1} * w_1 + t_{b2} * (w_1 + w_2 - 1) + t_{b3} * (w_2) \tag{4.3}$$

where t_{bi} ($i \in \{1, 2, 3\}$) is the synthesis time of $\{b_1, b_2, b_3\}$, with synthesis time implying HLS+Logic Synthesis+P&R, in order to get accurate results. Thus, $t_{bi} = t_{bi,HLS} + t_{bi,LS} + t_{bi,P\&R}$.

Although the number of configurations might seem large, the running time for each component pre-characterization is relatively short, compared to having to fully P&R the full system, which for each new configuration requires t_c. Thus, the running time of an exhaustive search method for the complete system is given by the following equation:

$$Cost_{ES} = t_c * w_1 * w_2 \tag{4.4}$$

with $t_c >> t_{bi}$ (synthesis of complete system vs. synthesis of a single component). Once each component is synthesized, the area, delay, and latency are extracted and annotated. The method then continues by creating an initial predictive model of the total area and throughput of the complete system in the next step of this first phase. It should be noted that in case that the latencies of the different component pairs do not match, the data is synchronized through a simple handshake protocol. No hand-shake is needed if the latencies match.

Step 2–Predictive Model Generation: With the information obtained from the previous step, the proposed method creates two initial predictive models. One to estimate the area and the other for the throughput of the entire system. As the aim of this work is to create unique configurations with different area vs. throughput characteristics, hence both metrics need to be predicted.

To create the predictive model, the proposed method arranges the synthesis results obtained during the pre-characterization phase in a matrix as shown in Fig. 4.4. In this

input matrix X, each row is the results of an individual configuration, also called sample (s_i, $i \in \{1, 2, ..., m\}$), where each column represents a predictor, p_j ($j \in \{1, 2, ..., n\}$. For instance, for area prediction, p_1 would be the area of module 1 with a certain number of inputs and a maximum number of outputs. While p_2 would be the area of moudle 1 with a maximum number of inputs and a certain number of outputs. The vector y are values that need to be predicted, and in this work are area (A) and throughput (T) of the complete dataflow. Thus, two matrices are generated for each of the values to be predicted. The objective of this step is to create a predictive model h by training X so that $h(X)$ is as close as possible to y.

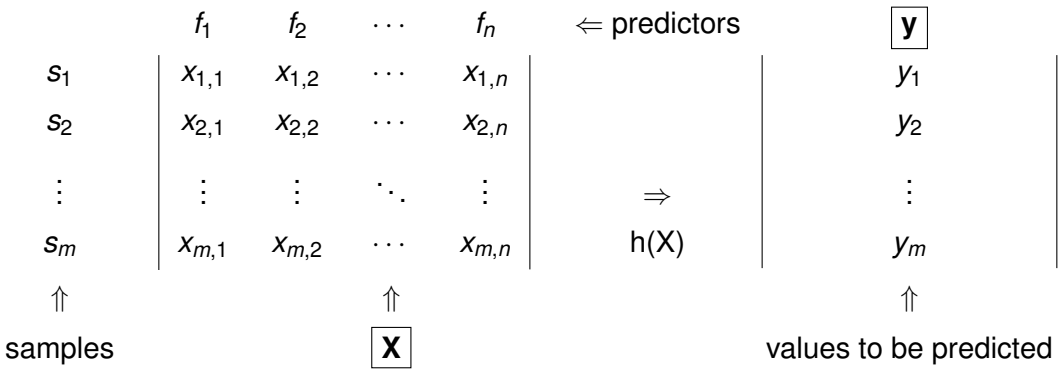


Figure 4.4: Sample data organization overview

Two predictive models are used in this work: a simple multivariable linear regression model and a more complex random forest model. In the experimental result section, the accuracy vs. running time of these two models is presented. To create these models a Python-based open source machine learning package is used (scikit [94]). The method then continues with the model refinement phase to take into consideration the effect of the overall interconnect.

4.4.2 Phase 2: Model Refinement and Dominating Configurations Search

This second phase consists of two additional steps. The first step characterizes a set of important samples that are known a priori to be important for the model refinement. The second step then continuously adds new samples to the model generator and re-calibrates the model taking into account the topology of the complete system after P&R. Because the P&R of the complete system can take a long time to finish, the goal

at this stage is to reduce the number of samples required to calibrate the predictive models. Fig. 4.5 shows the flow chart of the two steps. In particular:

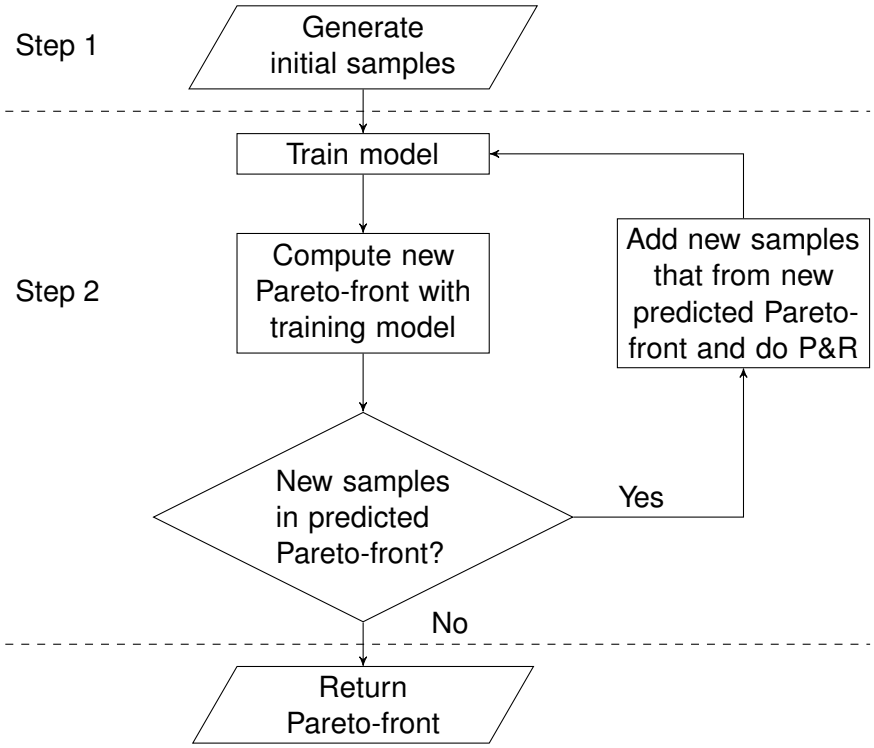


Figure 4.5: Adaptive Samples Selection Filter (ASSF) overview

Step 1–Generate Initial Samples: Important samples that are known to help to refine the predictive model are first automatically generated by default in this first initialization step. Intuitively, the sample with the maximum number of interconnects yields the highest throughput, but also the largest area; while the sample with the least amount of interconnects yields the lowest throughput, but also the smallest area. Each sample is fully P&R, and the area and throughput are annotated and added to the pre-characterization results obtained in the previous phase.

Step 2–Iterative Model Refinement: In this step, the proposed method continues by adding individual samples to the model refinement process, re-calibrating the model to consider the effect of the overall interconnect.

Most previous works on predictive models create the model and then compute the error of the model by measuring the difference between the predicted values and the actual values. A standard way of doing is using a 10-fold stratified cross-validation method (data evaluated until this point are divided randomly into ten parts, where each

part is held out in turn, and the learning scheme is trained on the remaining nine-tenths [116]). The model is considered stable when the error is smaller than a given threshold value.

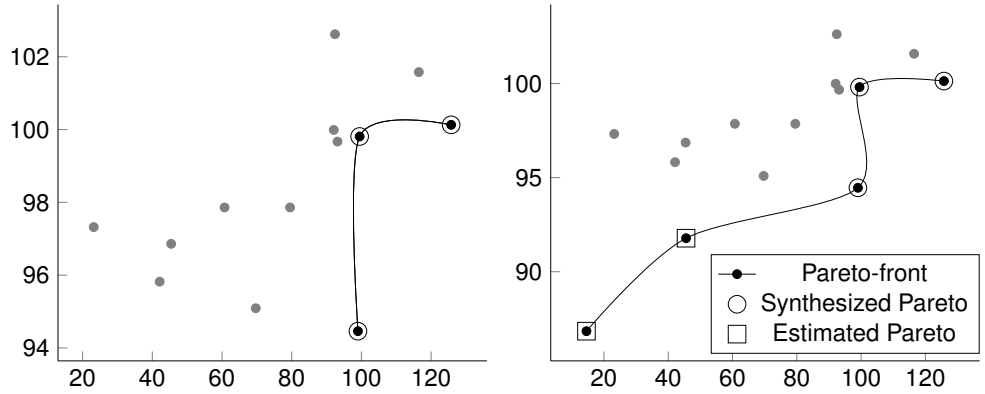
In this work, we introduce a unique approach. Often, the error based model creation does not converge or takes an extremely long time to reach the specified minimum error. Also, it is often non-trivial to set the error threshold value. Thus, often heuristics lead to better results and are faster. Therefore, this work proposes a different approach. As mentioned before, the goal in this step is to refine the predictive model, while requiring the smallest number of samples as these have to be fully P&R, which for large systems can be extremely time-consuming. Thus, we develop a method called Adaptive Samples Selection Filter (*ASSF*).

Fig. 4.6 shows an example of how *ASSF* works. The x-axis and y-axis represent normalized throughput and area, respectively. Based on the current fully P&R circuits, an initial Pareto-front is obtained, as shown in Fig. 4.6(a). Each point represent a sample (S_i) with a unique pin multiplexing ratio $MuxR_i$ between all the components, which results in a given area and throughput

$$S_i(MuxR_i) = \{A_i, T_i\}.$$

The predictive model is then used to find new configurations that lead to potential new Pareto-optimal designs. Fig. 4.6(b) highlights these in squares. These are only predicted optimal configurations. Thus, these configurations are added as new samples to the model's training set and are fully synthesized and P&R. The learning model is in turn re-calibrated based on the results. The process is repeated, by evaluating new potential dominating designs based on the continuously refined model, adding new candidates to the samples to be synthesized to continue with the model refinement.

This step continues until no further optimal configurations are found and returns the final trade-off curve with optimal pin multiplexing configurations for each block pair



(a) Syntheized samples with Pareto (b) Predicted new Pareto

Figure 4.6: Adaptive samples selection filter (ASSF) example

composing the system, i.e.,

$$PO = \{P_1(MuxR_1) = (A_1, T_1),$$
$$P_2(MuxR_2) = (A_2, T_2), ...$$
$$P_i(MuxR_i) = (A_i, T_i), ... \}.$$

4.5 Experimental Results

This section first describes the experimental settings. It then continues presenting the results of the proposed method as well as comparing it with other methods, and discussing these results.

4.5.1 Experiment Settings

To fully characterize the proposed method, instead of using a single data-flow engine as a case study, which would not allow drawing any major conclusions, a multitude of synthetic systems of different sizes and interconnect complexities are created. These designs were taken from the open source S2Cbench benchmark suite [101]. Table 4.1 highlights the details of the different designs created, their number of inputs and outputs and their bit widths. Table 4.2 shows the various synthetic systems generated ranging from 2 block systems to 3 and 4 blocks directly connected. To further study the stability of the proposed method, different constraints of maximum fanout in the P&R process

are used in each complex benchmark, as this constraint can severely impact the overall delay. Thus, in total 21 different system configurations are created. For example, AES-KASUMI-10 is a circuit consisting of the AES and KASUMI benchmarks, and the max fanout constraint is set to 10. The numbering in Table 4.2 indicates the order in which the different components are connected and the last row shows the total number of components in the given benchmark.

Table 4.1: Single Benchmarks Details

	#Inputs	I-bit	#Outpus	O-bit
AES	33	8	16	8
ADPCM	16	8	4	8
KASUMI	16	8	16	8
SNOW3G	16	8	16	8

Table 4.2: Complex benchmarks formation

Benchmark	S1	S2	S3	S4	S5	S6	S7
AES	1				1	1	1
ADPCM		1			3		4
KASUMI	2	2	1	2	2	2	2
SNOW3G			2	1		3	3
Designs	2	2	2	2	3	3	4

To measure the quality of the proposed method, and considering that the result is a trade-off curve of dominating solutions, this work follows the suggestions of [99] and uses the Average Distance from Reference Set (ADRS) as the quality measure. ADRS measures the distances between the actual Pareto-optimal front obtained by the exhaustive search method, which guarantees that the exact optimal solutions are found (PF_a), and the front obtained by our method (PF_m). ADRS is calculated as follows

$$ADRS(PF_a, PF_m) = \frac{1}{|PF_a|} \sum_{p_a \in PF_a} (\min_{p_p \in PF_m} \{dist(p_a, p_m)\}) \tag{4.5}$$

where $dist(p_a, p_m)$ is the distance of each solution between the two fronts.

The value of ADRS is inversely proportional to the degree of similarity between the two Pareto sets. A high ADRS value indicates that a significant part of the reference Pareto-front is missing in the approximate Pareto set.

The running time of the different methods is used to measure the effectiveness of the proposed method quantitatively.

The experiments were run on an Intel Pentium 4 running at the 3.20GHz machine with 8 GBytes of RAM running Linux SUSE version 3.0.13-0.27. The HLS tool used in this work is CyberWorkBench v5.5 from NEC [78] and the target HLS frequency set to 100MHz. The target FPGA is a Xilinx XC5VLX330 Virtex 5 FPGA, and the design tool is ISE v14.3.

To measure the qualitative and quantitative effects of the proposed method, *ASSF* is compared against a state-of-the-art method called clustering similarity measurement method (CSM) [113] and an ES, which guarantees to find the optimal solution, albeit taking extremely long running times. As mentioned in the previous section, two predictive models are used in our method: linear regression (ASSF-LR) and random forest (ASSF-RF). Results of both methods are presented.

4.5.2 Results

The experimental results are shown in Table 4.3, where the results of the exhaustive search are used as the reference. The first two columns indicate the benchmark name and their maximum fanout constraints. The next three columns present the ADRS result (qualitative result), and the last four the running time of each method (quantitative results). The results are split into three parts: 2-block systems, 3-block, and 4-block systems for easier interpretation of the results.

ADRS

As shown in the table, our proposed method leads to outstanding results, with average ADRS of 7.4% and 5.8%, 9.0%, and 8.8%, and 13.8% and 6.3% for the linear regression model and the random forest model for the 2-block, 3-block, and 4-block systems, respectively. The random forest leads to better results because it is a much

Table 4.3: Experimental Results (ADRS and Runtime)

Benchmark	Fanout	ADRS			Running time [s]			
		ASSF-LR	ASSF-RF	CSM	ASSF-LR	ASSF-RF	CSM	ES
AES-KASUMI	10	0.0	0.0	5.2	16246	19080	16246	22672
	100	0.0	0.0	0.0	10407	11724	19626	21072
	1000	13.0	8.0	13.0	17088	18399	22332	20976
KASUMI-ADPCM	10	1.6	2.7	1.6	11677	11677	13774	11184
	100	3.3	0.0	10.8	8622	12450	9260	10208
	1000	5.0	2.2	5.0	8761	11866	13108	9936
KASUMI-SNOW3G	10	10.6	4.7	10.7	8088	9336	9648	4992
	100	9.8	6.4	32.2	9688	11692	14364	10688
	1000	19.9	19.9	19.9	9702	9029	13067	10768
SNOW3G-KASUMI	10	6.0	6.0	6.0	16880	14370	19390	20080
	100	0.0	0.0	0.0	11425	14872	16021	18384
	1000	19.1	19.1	19.1	12576	14880	16032	18432
2-Block average	**Avg**	**7.4**	**5.8**	**10.3**				
	Geomean				**11,763**	**13,281**	**15,239**	**14,949**
AES-KASUMI-ADPCM	10	9.4	6.4	9.6	43920	62290	53105	470272
	100	8.3	8.8	6.0	25888	53888	104638	448000
	1000	5.6	5.7	69.4	42958	51988	124228	462336
AES-KASUMI-SNOW3G	10	3.6	2.8	5.1	41380	47122	72004	489984
	100	15.7	16.6	25.7	33904	41160	91952	464384
	1000	11.3	12.5	17.0	53802	66444	117012	462336
3-Block average	**Avg**	**9.0**	**8.8**	**22.1**				
	Geomean				**40,309**	**53,815**	**93,823**	**466,219**
AES-KASUMI-SNOW3G-ADPCM	10	26.2	10.4	15.5	84308	120440	199363	NA
	100	8.9	3.2	30.9	75024	125040	212536	NA
	1000	6.4	5.3	36.2	77558	134235	173014	NA
4-Block average	**Avg**	**13.8**	**6.3**	**27.5**				
	Geomean				**78,963**	**126,571**	**194,971**	**NA**
Total Average		**8.7**	**6.7**	**16.1**				
Total Geomean					**21,378**	**25,260**	**35,771**	**61,489**

more complex model. Regardless, a simple multi-predictor linear regression model also leads to good results, and as shown in the runtime comparison it is faster than the random forest. In comparison with the CSM method, our proposed method is for 2-block systems 28% and 44% better, for the three-block systems on average 59% and 69% better, and finally for the four block system 50% and 77% for the linear regression and random forest models, respectively.

Running Time

To measure the effectiveness of the proposed method, we measure the running time required to find the trade-off curve. For 2-block circuits, *ASSF-LR* is on average 35.6%

and 15.9% faster than *CSM* and *ASSF-RF*.

Compared with *ES*, *ASSF-LR* and *ASSF-RF* are 35.2% and 16.6% faster. Although not much faster than the *ES*, it can be observed that the improvement in runtime is more significant for 3-block than 2-block benchmarks, which indicates that our method has a high initial overhead due to the pre-characterization process of individual components, but scales well. On average, *ASSF-LR* is 157.7% and 47.8% faster than *CSM* and *ASSF-RF*. The *ES* could not find a solution within five days for the 4-component system. To compute the geometric mean, and compare all the methods fairly, a five-day runtime is given for the *ES* in those cases that it could not finish within the maximum time frame allowed (5 days). This further highlights the complexity of the problem to be solved.

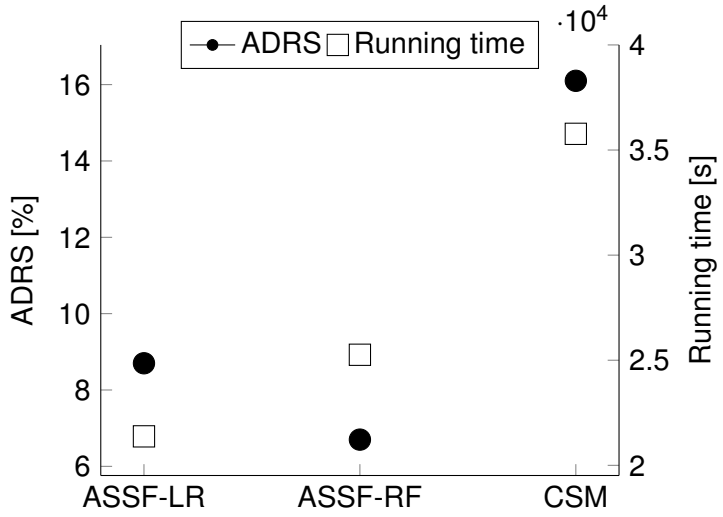


Figure 4.7: Comparison of the three methods

Finally, Fig. 4.7 gives a quick overview of the ADRS and runtime of the proposed method compared to the CSM method on a single graph, where only the averages are shown. This graph indicates that our method is superior to the CSM method in both dimensions, while a clear runtime vs. quality of results can be observed between the linear regression and the random forest predictive method versions.

4.6 Summary

FPGAs play a major role in the semiconductor world, especially due to their reconfigurability. However, this flexibility comes at a cost: in particular, larger area, higher power

consumption, and longer critical path delay compared to dedicated solutions. This work addresses this last issue, but considering the impact of the interconnect on the area and throughput of complex systems, composed of multiple directly connected parts. These system configurations, also called data-flow engines or stream computing are extremely popular in FPGA design. This work has proposed a learning-based method to find the Pareto-optimal system configurations with unique area and throughput by controlling the pin multiplexing ratio between the different components of a data-flow engine. Experimental results show that the method is fast and accurate compared to other reference methods.

Chapter 5

Design Space Exploration Prediction from ASIC to FPGA

FPGA and ASIC are the two major design platforms of modern VLSI systems. FPGAs are favored when flexibility and quick time-to-market are the two dominant concerns. ASICs are preferred for mass production, or when the goals are low unit cost, low power consumption, and fast speed. In practice, however, designs are often converted from one platform to another. In terms of the conversion from FPGA to ASIC, a typical situation would be that a functional block initially targeting FPGAs may be later converted to ASICs for mass production. In terms of ASIC to FPGA conversion, there are two common cases. The first case is FPGA-based verification in ASIC design flows such as prototyping and emulation. The second case is the migration of ASIC designs onto FPGA platforms. This chapter discusses the conversion from ASIC to FPGA. In particular, it analyzes the issues of the conversion and introduces an efficient predictive approach to facilitate the conversion.

5.1 Introduction

The move to heterogeneous computing systems based around complex hardware accelerators, combined with shorter and shorter required TAT has led many VLSI design companies to embrace HLS finally. Raising the level of abstraction from the RT-level to the behavioral level has multiple advantages. One of the most important ones is the ability to quickly re-target a circuit from one technology to another. This could be from one ASIC technology node to another ASIC technology, from one FPGA to another

FPGA with the same or different underlying structure (*e.g.* LUT size and number, and types of hard macros) or from ASIC to FPGA [110, 97, 100]. This work mainly deals with this latter one.

Note that the RTL codes designed for FPGA and ASIC platforms are usually different, since designers have to explicitly instantiate the existing IPs which are built for different platforms, and the design team also has to manually change the coding style to meet the design constraints such as timing constraint on different platforms. This situation can be simplified in HLS. The source code is written in a higher language like C or C++, thus without any platform-dependent information. The HLS tools take as inputs the source code, the targeting clock frequency, and the platform-specific libraries, then transfer them to the RTL code that is specific to the targeting platform. The most important advantage of this transition is that it requires no manual modification of the source code. Thus it is fast and less error-prone. Another important feature of designing in higher language is that the synthesis process can be re-tuned for the target technology by inserting a set of different synthesis directives in the form of pragmas. These pragmas typically allow to control how to synthesize arrays (*i.e.* registers or RAM), loops (*i.e.* unroll all, partially unroll or pipeline) and functions (*i.e.* inline or not) and thus, based on the target technology, can re-optimize the micro-architecture quickly, without the need to modify (re-write) the behavioral description.

This work makes use of this capability to quickly, without the need of any synthesis, find the Pareto-optimal designs for a behavioral description when an FPGA is targeted, given the previous results of an ASIC HLS design space exploration. In summary, the main contributions of this work are:

- Investigate if the synthesis directives that lead to Pareto-optimal designs when an ASIC is targeted also lead to Pareto-optimal designs when an FPGA is targeted.
- Introduce a predictive model-based approach to quickly, without the need for re-exploration, find the Pareto-optimal designs for an FPGA, given the exploration results for an ASIC.
- Present extensive experimental results that prove the effectiveness of our proposed method.

5.2 Motivational Example

This section presents a motivational example for this work and answers the first question of the contribution list. Fig. 5.1(a) shows the HLS DSE result of a five-stage decimation filter when an ASIC is targeted, generated using a multi-objective genetic algorithm (MO-GA), which has shown to work very well for these types of multi-objective optimization problems [31]. To search the design space, synthesis directives in the form of pragmas are used to generate micro-architectures with different area vs. performance trade-offs. Fig. 5.1(a) shows all the micro-architectures explored and highlights the most important ones, the Pareto-optimal designs that form the dominating trade-off curve. Out of all the generated designs, these are the most important ones.

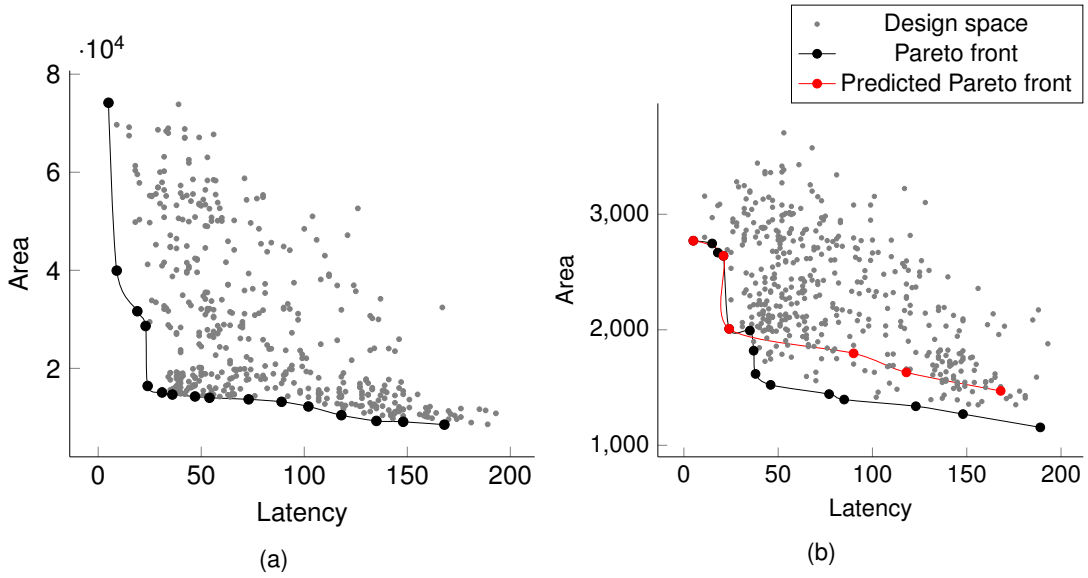


Figure 5.1: Design space exploration result of a five stage decimation filter, (a) DSE results for ASIC, (b) DSE result for FPGA highlighting Pareto-optimal designs obtained from FPGA synthesis and trade-off curve when Pareto-optimal designs from ASIC explorations are used.

Intuitively, one would think that when re-synthesizing (HLS) the same combination of synthesis directives that lead to the Pareto-optimal designs in the ASIC case, this would also lead to the Pareto-optimal designs when an FPGA is targeted. Fig. 5.1(b) shows that this is not the case. This figure shows that only one combination leads to a Pareto-optimal design in the ASIC and FPGA case. For the rest of the cases, other sets of synthesis options lead to better results. This observation has been consistently

seen across all of the behavioral benchmarks, from different domains (*e.g.* DSP and encryption), used in this work.

This observation is important because modern FPGAs embrace cutting-edge technology nodes like 16nm from Xilinx [34] and 10nm from Intel [81], and they are also big enough to hold complex systems on a single FPGA. According to [6] posted on Synopsys website in 2015, a LUT of Xilinx FPGAs equals the size of six 2-input NAND gates. So an UtraScale FPGA which contains 4400K logic cells is equivalent to 26.4 million gates. Besides, this type of FPGA often has built-in processors and memories, where a complete SoC can be fit and shipped to customers. Given this situation, some companies will prefer FPGAs to achieve fast implementation and to meet the time-to-market requirement, especially when they possess behavioral IPs which initially targeting ASICs.

Another situation is that currently most complex ASIC designs are prototyped or emulated on FPGAs. Fig. 5.2 shows an overview of a typical SoC flow and highlights how the use of emulation and prototyping significantly reduces the TAT as the embedded software can now be thoroughly tested before the actual SoC has been taped out. Logic simulation is too slow to allow for HW/SW co-simulation as logic simulators achieve simulation speeds in the order of kHz, while FPGA prototypes have been shown to work in the order of lower MHz ranges [28]. In the emulation and prototyping case, it is therefore important to re-tune the FPGA to enable and accelerate the verification process. So far this has been addressed by completely re-exploring the behavioral descriptions for the target emulation/prototype FPGA to obtain optimized designs for the FPGA, which for complex designs can easily take multiple days.

Based on the above situations, we can define the goal of this work as:

Problem Definition: Given a behavioral description (D) and its exploration results targeting an ASIC, $D^A = \{D^A_1, D^A_2, ..., D^A_n\}$, which include the Pareto-optimal designs

$$D^A_{opt} = \{D^A_{1,opt}, D^A_{2,opt}, ..., D^A_{m,opt}\},$$

where $D^A_{opt} \subset D^A$ and with each design (D^A_i) generated from a unique set of synthesis directives that we call attributes, $D^A_i = \{attr_1, attr_2, ..., attr_p\}$, generate a predictive

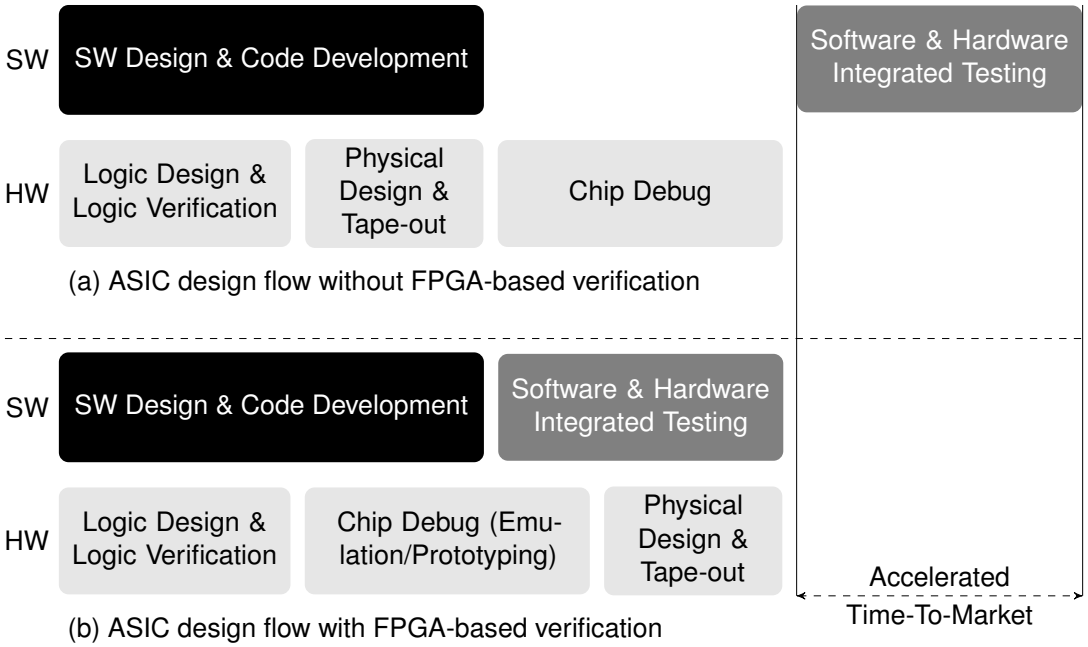


Figure 5.2: Complete SoC development flow overview (a) without FPGA (b) with FPGA emulation/prototyping.

model (PM) that takes as input the results from the ASIC HLS DSE (D^A) and quickly, without the need to perform a HLS DSE targeting a FPGA, returns the Pareto optimal trade-off curve for a FPGA (D^F_{opt}), where $D^F_{opt} = PM(D^A)$.

5.3 Related Work

Predictive model based DSE has been widely studied in two related domains, namely processors design and HLS, mainly because they share some common features. (i) The search space is huge (number of parameters to be explored). (ii) Each parameter or combination of parameters affect the results differently. (iii) There are several conflicting objectives that have to be balanced.

In the domain of processors design, Dubach [30] proposed an architecture-centric approach for DSE. The method uses a pre-trained (off-line) model to predict the performance of the rest unseen benchmarks. The model is a linear combination of five program-specific predictors which are trained using an Artificial Neural Network. In [60], the authors, leveraged the technique of transfer knowledge to predict the optimal configurations of the unseen benchmarks with a minimal number of simulations. The authors

train the prediction model using AdaBoost. Ozisikyilmaz [88] was able to accurately predict the performance of programs with a large number of design configurations using machine learning. They randomly sampled 2% to 10% configurations which are used to train three predict models: 1) linear regression, 2) neural network simple, 3) neural network complex.

In the domain of HLS, Chen et al. [17] introduced a ranking-based explorer called ArchRanker. It learns and predicts the rankings of the configurations instead of estimating the exact performance. Meng et al. [70] proposed a machine-learning-based method for DSE by eliminating non-Pareto configurations rather than searching the Pareto optimal. The idea is to estimate the risk of losing important points. In [62], the authors leveraged transductive experimental design to sample the search space and based on this used random-forest to find the Pareto-optimal designs. However, the limitations are 1) the design space is restricted to 242 knob settings and 2) only one benchmark is presented for method evaluation. Zhao et al. [124] introduced COMBA, a model-based framework to analyze the effects of pragmas on the generated architectures without actual high level synthesis using Vivado, thus exploring the design space with on average 100$\times$ speed-up compared with traditional approaches. Another important application of machine learning in HLS is to improve the Quality of Results (QoR) as the modern HLS tools are less capable of producing an accurate estimation of hardware resource utilization when targeting FPGAs. Several authors [29, 127, 26] proposed learning-based methods which take as input the pre-generated results after implementing on FPGAs, to make predictions of unseen design configurations using the inaccurate HLS reports, and to help calibrating the expected results. In particular, [29] utilized an adaptive windowing method to classify which designs need to be synthesized to find the true Pareto-optimal designs, [127] proposed Lin-Analyzer to analyze the performance of loop intensive applications, and [26] proposed an approach to study the relative importance among features and applications of different domains so as to guide the estimator to generate accurate QoR estimations.

The main problem with these methods is that they all require sampling the search space, which can involve considerable time. Our proposed method is universal and does not require the re-training of the model. It only requires a re-balancing of model

weights for each new design, which is extremely fast and does not require any re-synthesis at all.

5.4 Rapid ASIC HLS DSE To FPGA DSE Translation Method

The initial goal of this work was to find a single universal predictive model that could be used to estimate the logic resources count (*e.g.* slices, ALMs, ALUTs) for an unseen benchmark using the result of the ASIC HLS DSE. Unfortunately, this was not possible, mainly due to the hard-macros of the FPGAs, which introduce high non-linearity to the predictive models, thus, leading to sub-optimal results (see experimental results section for details). A different approach was therefore investigated, based on characterizing a training set composed of a variety of designs from multiple domains (DT_1, DT_2, ..., DT_n) and adjusting the predictive model for every new design (D_{new}) to be converted. This gives our proposed method the flexibility to adjust the predictive model for every new design, while at the same time does not require any time-consuming training process, which would involve having to re-synthesize some designs. We call our proposed method ASIC to FPGA (A2F).

The complete predictive model based translation framework consists of four main steps, shown in Fig. 5.3. **Step 1** performs design points sampling on several behavioral descriptions targeting both ASIC and FPGA technologies. These behavioral descriptions are considered as the training set from which the predictive models will be derived. A detailed description of how this exactly works is given in the next subsections. Although this training step takes a long time, it only needs to be executed once for the target technologies to generate the predictive models. **Step 2** continues by generating predictive models for the ASIC area (A_{ASIC}) and the FPGA area (A_{FPGA}) using the training data generated in step 1. **Step 3** continues by using the data generated during a HLS DSE and the results from step 2, and by generating a predictive model for the area (A_{ASIC}) of a new design targeting an ASIC technology (D_{new}). This is accomplished by building a weighted-regression model, which leads to a set of weights for each regression term indicating the importance that each training benchmark has on A_{ASIC} of D_{new}. **Step 4** finally estimates A_{FPGA} in *Slices* of D_{new} by leveraging the models generated in

step 2 for A_{FPGA} and the weights in Step 3. Also, the proposed translation framework also reports the number of DSP macros and Block-RAM used. The next subsections describe these four steps in detail.

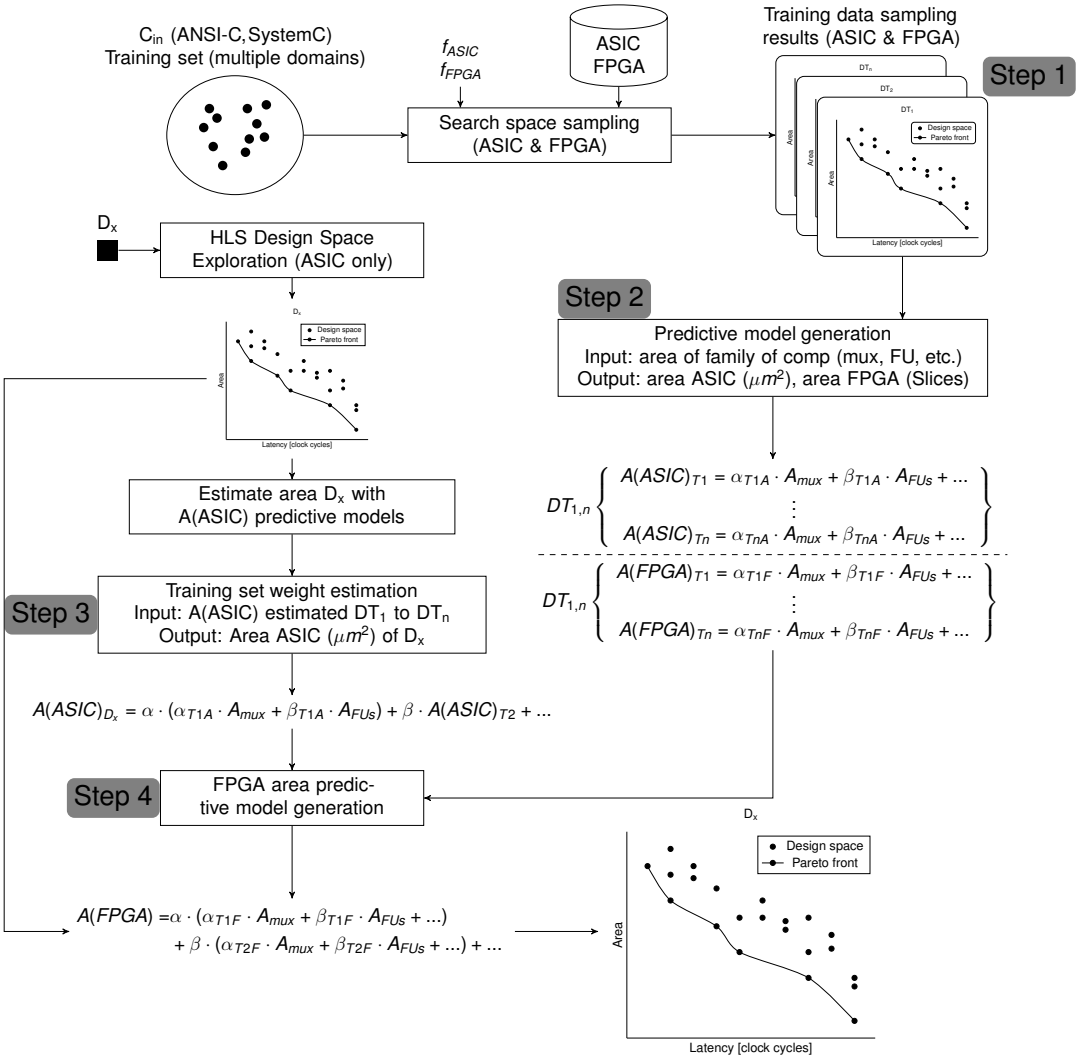


Figure 5.3: Complete FPGA slices predictive model generation flow.

Step 1: HLS Design Space Exploration (DSE)

This first step samples the entire search space for all the different design technologies (DT_1, DT_2), i.e., ASIC and different FPGAs for a set of training benchmarks. This sampling step enables the generation of the predictive model in the latter steps and only requires to be executed once. One of the main problems in this step is the exponentially growing number of design configurations with each explorable operation (mainly loops, arrays and functions). To make things worse, each new configuration needs to

be synthesized (HLS) to retrieve the effect of the synthesis attributes on the area and performance. This implies that a scalable sample mechanism is needed.

One simple method would be to randomly generate configurations that cover a given percentage of the entire search space. Although simple, this method might ignore some important combinations that often lead to Pareto-optimal designs. Some examples include loop unrolling factors that have matching register bank read/write ports, or matching Data Initiation Intervals (DIIs) with RAM ports. Other more complex approaches involve the use of Transduction Experimental Designs (TED) that aim at generating hard to predict settings [122]. The salient idea in TED is to select configurations that contribute the most to predictions on unlabeled test data that are given beforehand.

TED nevertheless also suffers from scalability issues. To address this, the authors in [62] proposed a randomized TED method, where in each sampling iteration they draw a random subset sample size, then add previously selected samples and continue following the same criterion as in the sequential TED algorithm to choose the best residuals combinations. Thus, in this work, we make use of this approach but enhance it by judiciously creating the initial seed configurations to combinations that have shown to lead to good results, but that are difficult to obtain randomly. These configurations include register and RAM port matching with DII and loop unrolling factors, fully unrolled loops combinations, loops with same unrolling factors and DIIs to enable loop fusions, etc... Thus, these configurations are set as initial sampling configurations from which to the complete search space is sampled.

This sampling is performed on all of the training benchmarks for the target ASIC and FPGA technologies under consideration. Because the training benchmarks are independent of one another, this step can be fully parallelized, by running each training design on separate machines/threats.

One significant difference between the ASIC and the FPGA exploration is the target HLS frequency. If kept identical, the HLS process might lead to very different micro-architectures for the ASIC and FPGA technologies when the same set of pragmas are specified. This is mainly because the delays in the FPGA are larger than that at the ASIC so that less logic can be scheduled in each HLS control step. Thus a micro-

architecture with different latency and resources will be generated. This mismatch of the delays leads to high non-linearities that are virtually impossible to predict and therefore, based on initial investigations, prevents to create accurate predictive models.

To address this, and to eliminate the delay effect in the HLS process, the target HLS frequency for the FPGA exploration (f_{FPGA}) has to be adjusted, such that the resultant circuit has the same latency (in clock cycles) as that of ASIC. For this, a binary search is performed each time a new chromosome (list of attributes) is generated. Once the latencies match, our method compares the number of functional units (FUs) in both versions (ASIC and FPGA). Having the same latency and FUs ensures that the only differences between the FPGA and ASIC micro-architectures are due to the synthesis directives and not the HLS target frequency. One additional benefit of this is that we do not need to predict the latency of the FPGA circuit as it will always match the ASIC latency (in clock cycles). Although time-consuming, this exploration only requires being executed once. It should be noted that much previous work has been done in the area of HLS DSE [62, 14, 124] and that this work does not try to improve on the previous work. Hence it is out of scope to prove the optimality of the results by comparing our proposed method with the previous work. In this work, we only require these exploration results to train our model and hence, could use any of the previously referenced methods for this purpose.

It should be noted that this work assumes that the ASIC HLS tool is also used for FPGA HLS. All of the commercial ASIC HLS tools support FPGAs [9, 71, 78] and hence, this does not pose any restrictions to our approach. One alternative solution would be to use the ASIC HLS tool for the SoC design and an FPGA HLS tool provided by the FPGA vendors to prototype the design on the FPGA. Although our proposed method would also work in this case, this training phase would require using both tools. One serious drawback with this approach is that HLS tool vendors, especially ASIC HLS tools, make extensive use of tool dependent hardware extensions, including their own data types. These constructs are vendor specific and would imply that the behavioral description would need to be re-written and re-verified for the FPGA HLS tool. Moreover, only Xilinx has a stable commercial HLS tool [119]. Intel has recently introduced a new HLS tool [42], but so far the tool does not seem to be competitive enough in terms

of quality of results. Thus, Intel FPGAs could not be targeted. Finally, HLS tools take different input languages, *e.g.* Intel's HLS tool does not support SystemC, which is popular in ASIC designs. Based on this, it is more practical for the ASIC designers to use the same HLS tool, which outputs RTL code which in turn is synthesized and placed and routed by the FPGA vendors' tools.

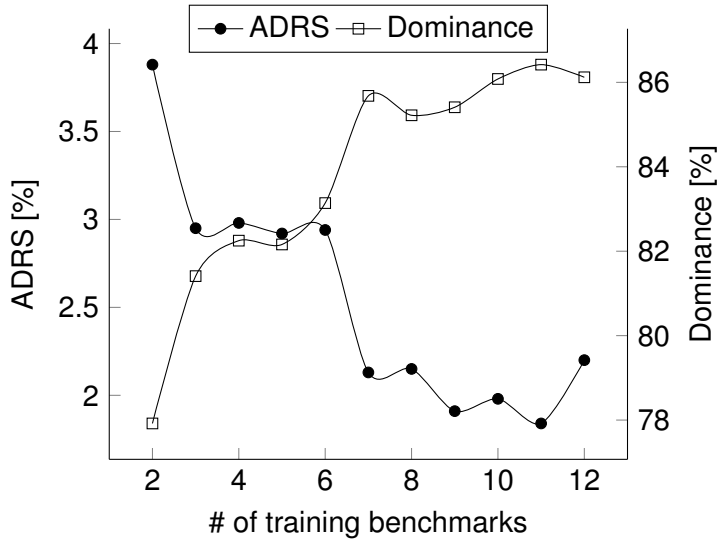


Figure 5.4: Quality of prediction model (ADRS and Dominance) vs. number of training designs (*DT*) used.

One obvious question is to decide on the number of training benchmarks to be characterized to build a stable model. For this, we iterated through the complete flow (steps 1 to 4) to predict the quality of the ASIC to FPGA area conversion framework by comparing the quality of the trade-off curves using a different number of benchmarks. The quality of the exploration result was measured using the ADRS, which is a widely used metric to compare multi-objective optimization problems like this one. ADRS indicates the average distance between the reference Pareto-front and the approximate Pareto-set *i.e.*, tells how close a Pareto front is to the reference front, where the reference front is obtained from the actual HLS DSE for the targeted FPGA (FPGA synthesis is performed). In particular, given a reference Pareto-front Γ and an approximate Pareto set Ω, ADRS is computed as shown in (5.1),

$$ADRS(\Gamma, \Omega) = \frac{1}{|\Gamma|} \sum_{\gamma \in \Gamma} \min_{\omega \in \Omega} f(\gamma, \omega) \tag{5.1}$$

where *f* computes the Euclidean distance between γ and ω. Another measuring metric is *Dominance*, which indicates the ratio of the number of correctly predicted Pareto optimal designs to the total number of the exact Pareto set.

From Fig. 5.4 it can be observed that after 11 test cases used to create the predictive model, the ADRS and dominance do not improve anymore. These results were consistently observed across multiple designs. Thus, it was decided to use 11 benchmarks from different domains, from the freely available SystemC S2CBench benchmarks suite [101] as training data for our proposed framework.

Step 2: ASIC and FPGA Predictive Model Generation

This second step takes as input the HLS DSE results obtained in step 1 and creates predictive models (*PM*) for the ASIC area and the FPGA area for each of the training designs used. For example, assumed there are three training benchmarks (DT_1, DT_2, DT_3) and a new unseen design (D_{new}) for which we want to estimate the logic resources count (slices, ALM or LABs depending on the target FPGA). The data representing the ASIC HLS DSE result for DT_1 is denoted as DT_1^A and DT_1^F for the FPGA DSE. Based on this notation, Table 5.1 shows the results after step 1. The goal is to estimate D_{new}^F (FPGA logic resources) as accurate as possible. For this purpose, a predictive model *PM* for A_{ASIC} for every training benchmark is first built. Taking DT_1 as an example, the training process takes as input DT_1^A, and create the model PM_1^A as shown in (5.2).

$$Training(DT_1^A) \Rightarrow PM_1^A \tag{5.2}$$

The generated predictive models (for ASIC area and FPGA area) are both based on linear regression. For any predictive model, the selection of the predictors is significant to the training process. In this work, the HLS tool reports the area for the different family of components for each design, *e.g.*, the area of multiplexers, decoders and functional units. The predictors with the highest impact on the model accuracy are finally used, which are reported by the predictive model generator [94]. Thus, PM_1^A and PM_1^F are two predictive model generated using DT_1^A, and DT_1^F (HLS DSE results targeting ASIC and FPGA). The corresponding models for DT_2 and DT_3 are obtained by performing the above operations for the other two test cases. Therefore, the outputs of this step

are: PM_i^A and PM_i^F, where $i \in \{1, 2, 3\}$.

Table 5.1: Example with three training designs of ASIC and FPGA model generation.

Benchmark	DT_1	DT_2	DT_3	D_{new}
ASIC	DT_1^A	DT_2^A	DT_3^A	D_{new}^A
FPGA	DT_1^F	DT_2^F	DT_3^F	

Step 3: Training Set Weight Estimation

Step 3 takes as input the HLS DSE for a new design (D_{new}) targeting only an ASIC and for which we want to obtain the FPGA Pareto-optimal designs. It also takes as input the predictive model for the ASIC area obtained in step 2. The output of this step is a new linear regression predictive model for the ASIC area of the new design, such that the regression coefficients of this new model indicate the weight that each training design has on the current design. We call the new model *Weighted-Regression*, which is denoted as *WR*. In the case of the 3 test designs introduced previously, this leads to the model shown in (5.3), where $PM_1^A(D_{new}^A)$ is the predictions using PM_1^A as the model and D_{new}^A as the input data.

$$WR(PM_1^A(D_{new}^A),\ PM_2^A(D_{new}^A),\ PM_3^A(D_{new}^A)) \Rightarrow D_{new}^A \tag{5.3}$$

The coefficients of *WR* indicate the weight that each training design has on the new design. It was observed that the higher the similarity between the designs is, the larger the coefficient is, *e.g.* if both designs make heavy use of hard-macros.

Step 4: FPGA Area Prediction

This last step estimates the *Slices* count of D_{new}, for which only the ASIC HLS DSE results are available, by using the FPGA predictive models (PM_i^F) from step 2 and the weighting regression model *WR* created in step 3. This operation is shown in (5.4), where D_{new}^F is the predicted values of the FPGA DSE results for the new design.

$$D_{new}^F = WR(PM_1^F(D_{new}^A),\ PM_2^F(D_{new}^A),\ PM_3^F(D_{new}^A)) \tag{5.4}$$

Using this predictive model, our proposed framework evaluates the effect of the different synthesis attributes on the FPGA area given in logic resources, which could be slices, ALM or LABs, depending on the logic source used when the predictive model is created and as mentioned before retrieves the latency (L) (in clock cycles) directly from the ASIC synthesis results. Thus, the Pareto-optimal configurations with unique *Logic resources* vs. *Latency* are quickly obtained. This work can make use of previous work, fully orthogonal to this one, to estimate the delay, based on the results obtained, and hence, the maximum frequency [26]. This would allow our method to plot logic resources vs. throughput (T), where throughput is given as $T = f_{max}/L$.

5.5 Experimental Results

To prove the effectiveness of the proposed framework, seven benchmarks from the open source Synthesizable SystemC benchmark suite S2CBench v.2.2 [101] are taken. The training set benchmarks are also taken from this benchmark suite but are different. The HLS tool used is CyberWorkBench from NEC [78]. The target ASIC technology is Nangate 45nm, and the target FPGAs are Xilinx Virtex 5 and 7. Two Virtex FPGA families with different underlying architectures are targeted to study the robustness of our proposed method. The Virtex 5 family is based on a 6-input/1-output or dual 5-input/2-output configurations, while the Virtex 7 family is based on 6-input LUTs. Scikit [94] is used to create predictive models.

We could not find any published method(s) to compare our approach against. Hence, to measure the quality of the results, two other methods were implemented. The first is called Direct Method from ASIC (DM), which basically considers the Pareto-optimal designs obtained from the ASIC HLS DSE and directly assumes that these configurations are also the Pareto-optimal designs for the FPGA. The second method (LR), is based on creating a single, static fixed linear regression model to estimate the logic resources of the FPGA based on the training benchmarks.

To improve the accuracy, the area of the multipliers are directly set to zero, as they will be mapped to DSP macro. The area of the adders in LUT is set to *# adders* $\times$ *bit-width*, as each full adder typically occupies one LUT. Finally, we also execute

a full FPGA HLS DSE for the target benchmarks (*Synth*) to get accurate results for verification.

Fig. 5.5 shows graphically the trade-off curves obtained for each of the four methods. It can be seen that our proposed method leads to better results when compared to the other two methods. Table 5.2 and 5.3 report these differences quantitatively using ADRS and Pareto dominance for the two FPGA families (Virtex 5 and Virtex 7) as described in the previous section.

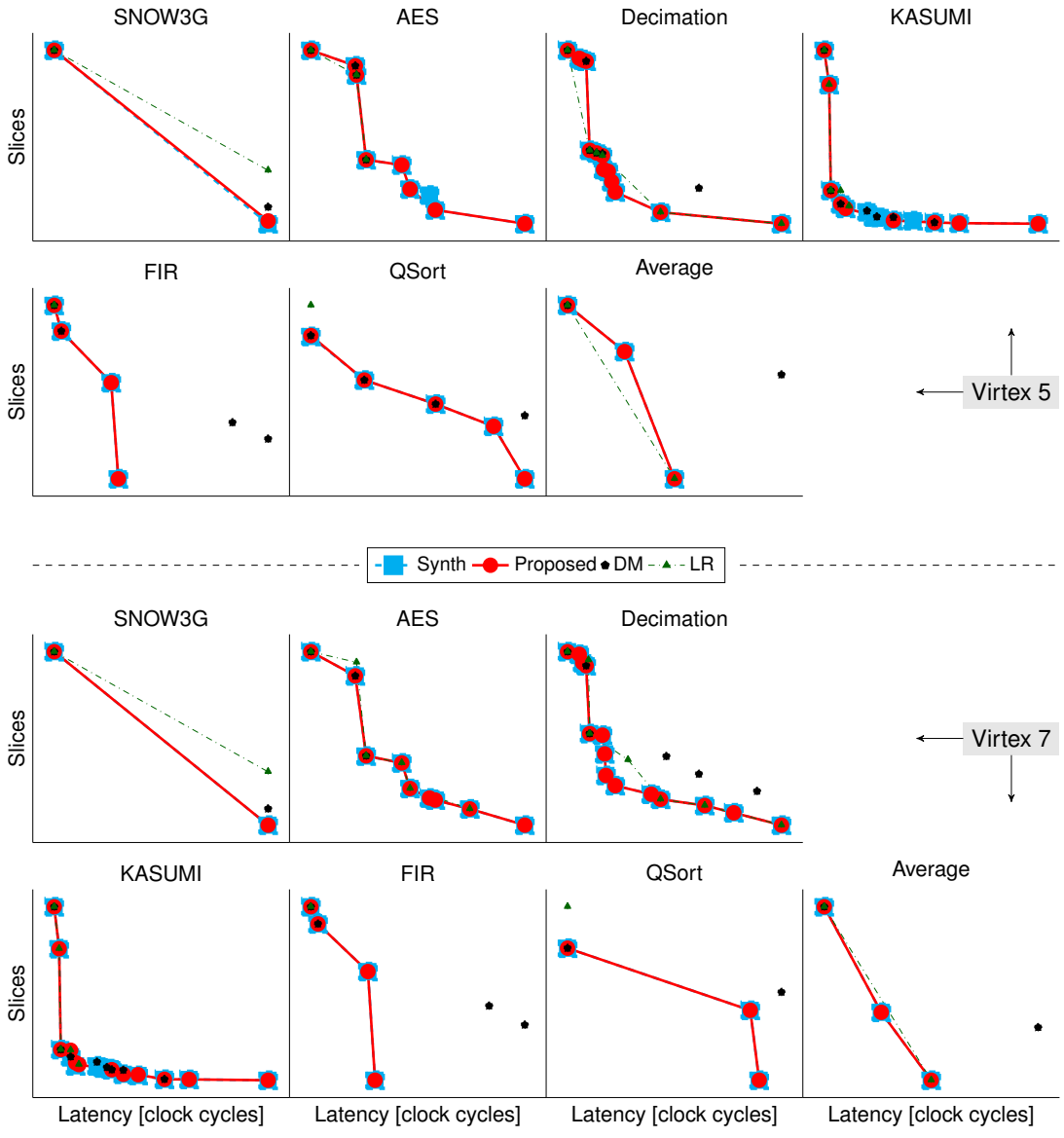


Figure 5.5: The trade-off curves of the methods for seven benchmarks when Virtex 5 and Virtex 7 FPGAs are targeted

These two quality indexes are used to better understand the quality of the proposed

method. A high ADRS value implies that the method is missing completely a portion of the reference curve, while a low Pareto dominance indicates that the designs found by the method do not lay on the actual Pareto-optimal trade-off curve.

From Table 5.2 and 5.3 it can be observed that our proposed method outperforms the other two methods in both quality metrics. Note that, in the two tables, the values for *qsort* in the last column are **INF**, because the dominance obtained by *LR* is 0 for both Virtex 5 and 7 cases. In terms of ADRS, our proposed method (*Proposed*) is on average 90.65% and 96.08% smaller than *DM* and *LR* and regarding Pareto dominance, on average 107.89% and 145.76% larger for the Virtex 5 case. For the Virtex 7 case, the results are consistent. Our proposed method produces on average 97.65% and 98.26% smaller ADRS and on average 163.67% and 165.84% larger Pareto dominance. For some benchmarks, the improvement is 100% which means the proposed method can find the exact entire set of Pareto optimal designs.

Table 5.2: Comparison of Methods using ADRS and Dominance for Virtex 5

	Values						Improvement			
	ADRS [%]			Dominance			Δ ADRS [%]		Δ Dominance [%]	
	Proposed	DM	LR	Proposed	DM	LR	Proposed vs DM	Proposed vs LR	Proposed vs DM	Proposed vs LR
aes_cipher	0.29	9.74	9.93	0.89	0.44	0.33	97.02	97.08	102.27	169.70
average	0.00	13.21	2.34	1.00	0.33	0.67	100.00	100.00	203.03	49.25
decimation	0.02	10.40	5.06	0.92	0.31	0.38	99.81	99.60	196.77	142.11
fir	0.00	24.23	31.87	1.00	0.50	0.25	100.00	100.00	100.00	300.00
kasumi	4.88	8.71	85.89	0.69	0.31	0.23	43.97	94.32	122.58	200.00
qsort	0.03	2.97	16.23	0.80	0.40	0.00	98.99	99.82	100.00	INF
snow3g	2.87	17.23	55.07	0.50	0.50	0.50	83.34	94.79	0.00	0.00
Avg	1.16	12.36	29.48	0.83	0.40	0.34	90.65	96.08	107.89	145.76

Table 5.3: Comparison of Methods using ADRS and Dominance for Virtex 7

	Values						Improvement			
	ADRS [%]			Dominance			Δ ADRS [%]		Δ Dominance [%]	
	Proposed	DM	LR	Proposed	DM	LR	Proposed vs DM	Proposed vs LR	Proposed vs DM	Proposed vs LR
aes_cipher	0.01	13.99	2.61	0.89	0.22	0.44	99.93	99.62	304.55	102.27
average	0.00	30.10	9.93	1.00	0.33	0.67	100.00	100.00	203.03	49.25
decimation	0.00	13.57	6.04	1.00	0.21	0.36	100.00	100.00	376.19	177.78
fir	0.00	41.94	50.60	1.00	0.50	0.25	100.00	100.00	100.00	300.00
kasumi	3.03	8.37	39.27	0.57	0.36	0.21	63.80	92.28	58.33	171.43
qsort	0.00	5.54	13.88	1.00	0.33	0.00	100.00	100.00	203.03	INF
snow3g	0.00	16.10	52.40	1.00	0.50	0.50	100.00	100.00	100.00	100.00
Avg	0.43	18.52	24.96	0.92	0.35	0.35	97.65	98.26	163.67	165.84

These results indicate that the proposed method works well and that it leads to

better results than the other two predictive methods. The additional advantage of our proposed method is that it does not require any new training phase for each new design to be converted, and thus, is very fast. The model generation phase takes less than 1 second, while the actual re-exploration of the design, takes on average less than 1-minute making use of the new predictive method. In contrast, the average running time of the GA-based HLS DSE, having to fully re-synthesize each new design configuration targeting the desired FPGA is 7.7 hours. This exploration running time is a function of the complexity of the benchmark, *e.g.* the longest HLS DSE took 12 hours.

In summary, we can conclude that our proposed method is effective in automatically converting HLS DSE results obtained for ASICs to FPGA.

5.6 Conclusion

In this work, we have introduced a fast predictive model-based method to quickly convert the HLS design space exploration results obtained for an ASIC to those of an FPGA, and thus, to find the synthesis directives in the FPGA case that lead to the Pareto-optimal designs. We have also shown that the synthesis directives that lead to Pareto-optimal micro-architectures when an ASIC is targeted do not lead to Pareto-optimal designs for the FPGA case. This implies that a completely new design space exploration is needed. Thus, a method to quickly find the Pareto-optimal designs for FPGAs is important to facilitate and accelerate the prototyping and emulation of ASICs.

Chapter 6

Re-optimize Complex Dataflow from ASIC to FPGA

This chapter introduces an automatic stream computing re-optimization flow from ASICs to FPGAs. Complex VLSI designs need to be prototyped and/or emulated on FPGAs. Also chances are that designs initially targeting ASIC might port to FPGA platforms. These situations involve the conversion from ASIC to FPGA. The main problem that we address in this work is that architecture configurations targeting ASICs are often, as we will show in this work, highly un-optimal when mapped onto an FPGA. Thus, this work proposed a method to first generate a variety of dataflow configurations (a set of synthesis directives) targeting an ASIC given multiple behavioral descriptions for HLS and then based on a compositional predictive model automatically re-optimize the data flow when mapped onto an FPGA. Meanwhile, the predictive model utilizes the conversion approach discussed in chapter 5. Experimental results show that our proposed method works well.

6.1 Introduction

Complex SoCs include multiple embedded processors, memories, interfaces and numerous dedicated hardware accelerators, all interconnected through a shared bus, bus hierarchy or Network on Chip (NoC). These heterogeneous SoCs are required to be taped out at increasingly shorter time frames. These shorter TAT combined with the increased complexity of these SoCs have forced VLSI design companies to prototype and/or emulate these systems before the SoC is taped out. This helps reducing de-

sign faults avoiding costly and time-consuming re-spins, and also allows to engage the embedded software development team early on, before the silicon is ready.

The increasing speed and capacity makes the modern FPGAs not only as a good platform for verification, but also as competitive final products. Nurvitadhi [86, 85] at Intel implemented neural networks on FPGA, CPU, GPU, and ASIC, and compared their performances. FPGA outperforms both CPU and GPU, but it is not as efficient as ASIC. However, FPGA is a decent alternative if NRE cost could not be amortized by the mass production. In this case, the designs initially targeting ASICs would be ported to FPGAs.

Both prototyping/emulation and platform porting involve the conversion from ASIC to FPGA. In the verification case, designers intend to verify the functionality implemented in the same micro-architecture on ASIC and FPGA. This involves the manual modification of the RTL code since ASIC and FPGA differ from the basic elements and the IP libraries. In high-level synthesis (HLS), the designs are described using high-level languages like C/C++, thus no platform-dependent information is included. After HLS, latency (clock cycles) is one of the metrics that could prove the equivalence between ASIC and FPGA. Because when allocation (functional units type and count decision) and scheduling (functional units placement) are consistent, the resulting micro-architecture (the main datapath) would not change, thus the latency would be maintained. In the case of platform porting as products, the goal is to implement the same functionality on FPGAs while meeting the constraints such as area, latency, and power. Initially, a design space exploration targeting ASIC is performed to figure out the interesting design points. However, the optimal design points on ASIC would not directly leading to the ones on FPGA, which will be discussed in the motivation section. Therefore, a DSE targeting FPGA is required to figure out the new interesting design points.

One of the differentiation factors between different SoC vendors are the dedicated hardware accelerators. These dedicated hardware accelerators execute a fixed function one to two orders of magnitude faster than general purpose solutions, while being also much more energy efficient [18]. One example includes Apple's A11 Application processor, which includes a dedicated image processing unit and the neural network

accelerator (Neural Engine). To benefit from hardware acceleration, these functions need two main attributes: First, they should have large amounts of parallelism. Second, it should be processed using short word-length arithmetic. Some applications include neural networks, DSP applications, and multi-media application [67]. Very often, these applications are configured following the stream-computing paradigm, also called data-flow computing, where each module is connected to the next one and data is pipelined through all the modules in the data-flow. This leads to additional speed-ups as pipelining is a well known parallelization technique while reducing costly external memory accesses [41].

In order to further reduce the TAT, VLSI design companies have started raising the level of VLSI design abstraction by using HLS. Raising the level of abstraction from RT-level to the behavioral level has multiple advantages. One of the most important ones is the ability to quickly re-target a circuit from one technology to another, such as from ASIC to FPGA. One of the main enablers for this is that the synthesis process can be re-tuned for the target technology by inserting a set of different synthesis directives in the form of pragmas. These pragmas typically allow to control how to synthesize arrays (i.e. registers or RAM), loops (i.e. unroll all, partially unroll or pipeline) and functions (i.e. inline or not) and thus, based on the target technology, can re-optimize the micro-architecture quickly, without the need to modify (re-write) the behavioral description.

This work makes use of this capability to quickly, without the need of any synthesis, find the Pareto-optimal dataflow configurations with unique area vs. performance trade-offs when an FPGA is targeted, given the previous result of a dataflow optimized for ASIC. In summary, the main contributions of this work are:

- Propose a method to optimize data flow hardware accelerators specified as *N* behavioral descriptions.
- Introduce a method to re-optimize a data flow optimized for ASICs to FPGAs.
- Present extensive experimental results to validate the proposed method.

6.2 Motivational Example

This section presents an example to illustrate the importance of our contributions in this work. To re-target the design from ASIC to FPGA for emulation, the most straightforward approach is to perform a multi-objective heuristic method (e.g., genetic algorithm) on the entire data-flow system targeting FPGA platforms. Performing the heuristic approach gives a set of Pareto optimal configurations regarding the design objectives such as Area and Latency. However, two major issues make the approach impractical:

1. HLS is a single-process synthesis method, which means it cannot accept multiple modules and transform them into a connected top module.
2. Even by connecting multiple modules manually, the running time of logic synthesis and Place & Route is large as the data-flow system grows. Meanwhile, the design space grows exponentially while increasing the number of modules.

Therefore, alternative methods are necessary.

There are three methods, and their work flows are differed by using three types of lines. Referring to Fig. 6.2, every block is identified by the number at the top left corner. The first method is represented by the dotted line, it is similar to the above approach, except that it performs the heuristic on individual modules and utilizes a compositional approach to obtain the final Pareto optimal configurations. This workaround avoid the need of synthesizing multi-process modules and shrink the exploration space as well. This is the baseline in this work. The second method is indicated by the dashed line, it assumes that the Pareto optimal designs on ASIC will lead to the Pareto optimal designs on FPGA. Fig 6.1 shows that the assumption is not true. Only a small portion or none of configurations lead to Pareto optimal on both ASIC and FPGA platforms. For the rest of the cases, other configurations lead to better results which are missed by the method. This observation has appeared consistently across all benchmarks used in this work.

It is important to minimize the mismatches since current complex ASIC designs are prototyped or emulated on FPGA for verification. For prototyping, the engineers are trying to keep the micro-architecture identical on both platforms, because they have

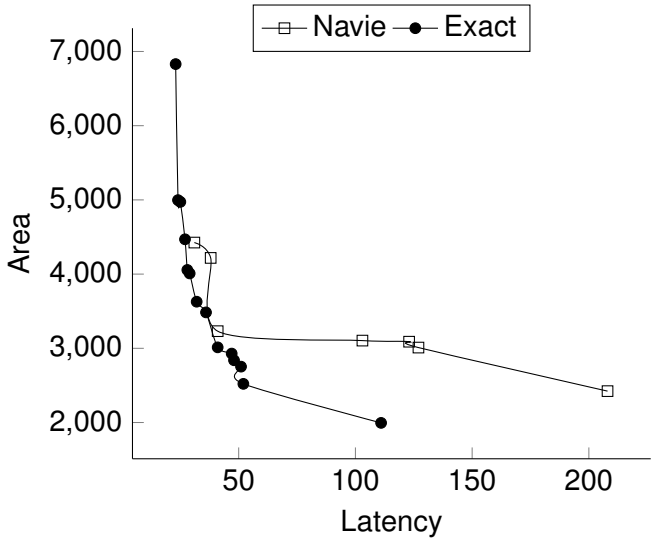


Figure 6.1: Motivation: the mismatch of the naive method

to guarantee that the designs they are verifying will be the same as those that will be taped out. However, for the case of emulation or hardware-assisted simulation, the goal is to accelerate the emulation process as much as possible. Therefore, it is possible to adjust the circuit mapped on FPGA to find the most optimal micro-architecture to further accelerate the verification process. In this case, the main problem would be having to completely re-explore the behavioral descriptions for the target FPGA, which happens in the situation of platform porting as well, and this process, for complex designs, can easily take multiple days. To deal with this issue, we define the objective of this work as the following.

Problem Definition: Given a dataflow system consist of n behavioral descriptions ($D = \{D_1, D_2, ..., D_n\}$) and their exploration results targeting an ASIC, $D^A = \{D_1^A, D_2^A, ..., D_n^A\}$, generate a predictive model (*PM*) and a compositional method (*Compo*) to obtain, without the need to perform any FPGA synthesis, the Pareto optimal trade-off curve for the dataflow system (D_{opt}^F), where $D_{opt}^F = Compo(PM(D^A))$.

6.3 Previous Work

Data-flow (streaming) computing has been extensively studied for its importance in multiple application domains such as images and videos processing, and signal processing. This technology utilizes the parallelism of underlying platforms (CPUs, GPUs,

and FPGAs) to improve the performance of streaming applications. Normally, three objectives are widely investigated: area, latency/throughput, and power.

For CPUs, [57] speeds up the stream programs by maximizing the pipeline stages on multi-core platforms. [104] reduces the overall energy consumption of streaming applications on the multicore (StrongARM processor) systems using both off-line and on-line Dynamic Voltage and Frequency Scaling (DVFS). [36] introduces a stream compiler to exploit the parallelism in the level of task, data, and pipeline on multicore platforms, so as to accelerate the stream programs. On GPU-based platforms, by leveraging the large array of parallel processing units, [48, 47] proposes a framework for mapping stream programs onto multi-GPU platforms under shared memory constraints, so that the speedup is maximized. [83] presents the heuristic partitioning and ILP mapping algorithms to optimize the performance of streaming applications in the systems of up to 4 GPUs. For stream programs mapped on FPGAs, [22] utilizes DSE as actor selection combined with actor replication to meet the throughput requirement while trying to minimize the area. [23] proposes an approach to further balance the occupied hardware resources and throughput of streaming applications when mapped onto an FPGA. This approach is able to figure out the profiles of actor implementation & replication, buffer size for communications, and pipeline scheduling. [39] proposes a stream folding algorithm which first replicates filters and then reduce the unnecessary replicas in order to optimize the throughput under the constraints of area and latency. [44] introduces the Optimus compiler which maps the streaming applications on FPGAs in an efficient way. [45] optimizes the energy of stream computing on FPGAs using power gating given that the stream program executes in a repetitive way. [115] implements the streaming applications on CPU-FPGA heterogeneous systems and improves the throughput by balancing the workload on CPUs and FPGAs so that the efficiency on each platform is maximized.

These works mainly focus on stream programs given in synchronous data flow graph (SDFG) [59], and the strategies can be grouped into three categories: 1) workload balance/actor selection, 2) communication overhead minimization (latency matching), and 3) pipeline stages maximization.

Other researches have been focused on the cross-platform performance predic-

tion. O'Neal [87] proposes HALWPE to predict the performance, using cycle accurate simulation, of future devices, particularly the GPUs from one generation to the next generation. Zheng [126] introduces LACross which predicts the performance and power on ARM-based platforms given the existing performance data on Intel and AMD based platforms. A part of this work shares the idea that the model learns the relation between different platforms but using different strategy.

In this work, the data-flow applications are given as multiple modules (actors) connected sequentially, and each module has been explored in terms of the design space for ASICs. We propose a method to 1) predict the DSE results on FPGAs using machine learning techniques, which is equal to actor selection, 2) fine-tune the inter-module connections to minimize and match the communication latencies, and 3) fix the clock frequency of the modules and ensure the data-flow computing executes in a repetitive way so that the pipeline stages are maximized.

6.4 Proposed ASIC to FPGA Optimization Flow

Fig. 6.2 shows an overview of the complete flow which is composed of 3 main phases. Note that the other two comparative approaches are also presented in this figure, because the three methods share some common steps. The input to our flow is a list of n synthesizable (HLS) behavioral descriptions $C = \{C_1, C_2, ..., C_n\}$ and the outputs are two trade-off curves of Pareto-optimal dataflow configurations – one targeting the ASIC technology where the SoC will be taped out and another targeting the FPGA where the dataflow will be emulated or prototyped. The first phase pre-characterizes each behavioral description (C_i) performing a HLS Design Space Exploration (DSE) on it. This leads to a set of unique micro-architectures for each description with unique area (A) vs. latency (L) trade-offs, thus, $C_i = \{D_1(A, L), D_2(A, L), ..., D_{p_i}(A, L)\}$, where p_i is the number of Pareto optimal designs of C_i.

The proposed flow is represented by the solid arrows, and the method includes seven steps (1, 2, 3, 5, 8, 11, 7, 10). The next subsections describe the four phases in detail.

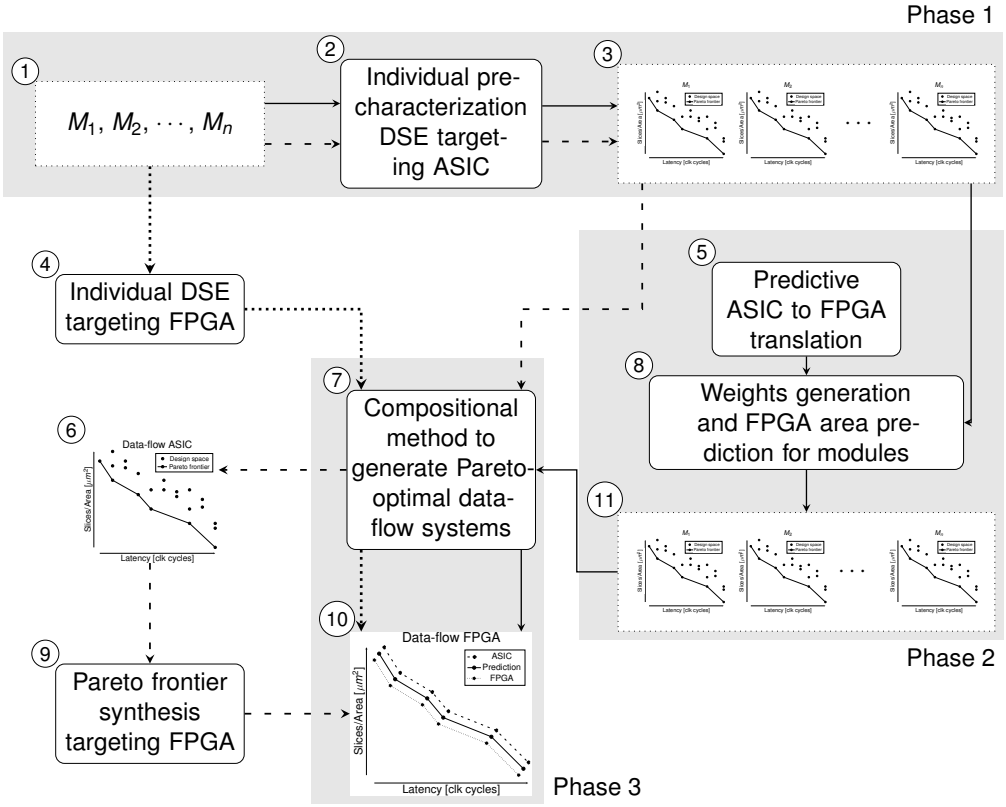


Figure 6.2: Complete dataflow system generation method overview.

6.4.1 Phase 1: Individual Micro-architectural Design Space Exploration (Steps 1, 2, 3)

This first phase pre-characterizes each behavioral description composing the dataflow individual by creating a set of Pareto-optimal micro-architectures. HLS is a single process synthesis method and hence can only synthesize each description separately generating a hardware module for that description, which in turn is *stitched* to the next module either directly or through standard interfaces, *e.g.* AHB, AXI buses or FIFOs for which commercial HLS tool vendors provide synthesizable APIs.

A brief description is given of the exploration method based on a genetic algorithm meta-heuristic as this is not part of the main contribution of this work. Much work in the area of single process micro-architecture DSE has been done and it is out of scope of this work to proof the optimality of the results [62, 66, 121]. In this work we make use of a genetic algorithm search that has shown to produce very good results in a reasonable time, as the extremely large search space makes exhaustive searches

unfeasible [33].

Each explorable operation, *EOP*, is represented as a gene to which a synthesis directive is assigned, and the list of all genes builds a chromosome *CM*. Therefore a gene, *EOP* is equivalent to a *CM* . In this work $EOP = \{arrays, loops, func, pins\}$, where $arrays = \{register, expand, logic, RAM, ROM\}$, $loops = \{no, partial, all, folding\}$ and $func = \{goto, inline\}$. Pins represent the number of pins assigned to either inputs or outputs and have to be time-multiplexed, $pins = \{input, output\}$.

The first step of the genetic algorithm is the random generation of an initial population of *M* chromosomes. Then, each member of the population is paired with another randomly selected member. The crossover operator selects a cut-point at random and combines the left half of one parent with the right half of the other to produce an offspring. The crossover operator is performed based on the crossover rate r_c specified by the user. The offsprings are then mutated. This implies randomly selecting one gene within the offspring's chromosome and changing it to another random value. Only a certain percentage of the offsprings produced are mutated. The probability is also determined by the mutation rate passed by the user as an exploration parameter, r_m. In this work r_c and r_m are set to 0.8 and 0.1 respectively.

At this point, the mutated offspring is synthesized calling the HLS tool and the area, and latency of the synthesized design back-annotated, leading to a new design $D_{new} = \{A_{new}, L_{new}\}$. The newly generated offspring will replace one of the parents if one of the following conditions is met: (i) The offspring improves one or more of the best-so-far values. The parent to be replaced is randomly chosen. Moreover, if an identical copy of the offspring already exists within the population, the offspring is discarded. (ii) One of the parents is dominated by the offspring (i.e., is inferior to the offspring across all of the objectives, in this case, area or latency).

6.4.2 Phase 2: Predictive ASIC to FPGA Micro-architectural Exploration Translation (Steps 5, 8, 11)

This translation is based on the previously method proposed in [63]. This phase is done offline and only needs to be done once independently of the dataflow being optimized. The details have been discussed in Section 5.4, and Fig. 6.3 shows an overview of the

approach which contains three steps.

Step 1 samples the search space of several behavioral descriptions used as training set targeting both ASIC and FPGA technologies by generating different HLS configuration settings (synthesis directives in the form of pragmas) and synthesizing each new configuration. These behavioral descriptions are considered as the training set from which the predictive models will be derived. A detailed description of how this exactly works is given in the next subsections. Although this training step takes a long time, it only needs to be executed once for every target technology. **Step 2** continues by generating predictive models for the ASIC area (A_{ASIC}) and the FPGA area (A_{FPGA}) using the training data generated in step 1. **Step 3** continues by using the data generated during step1 and the results from step 2, and by generating a predictive model for the area (A_{ASIC}) of a new design targeting an ASIC technology (D_{new}). This is accomplished by building a weighted-regression model, which leads to a set of weights for each regression term indicating the importance that each training benchmark has on A_{ASIC} of D_{new}.

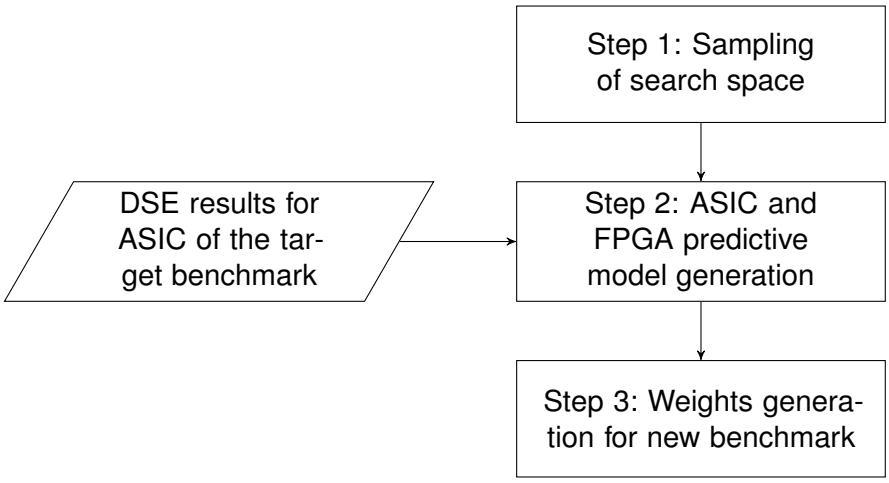


Figure 6.3: Predictive translation from ASIC to FPGA overview.

6.4.3 Phase 3: Compositional Method for Optimal ASIC Dataflows (Steps: 7, 10)

This last phase makes use of the results of phase 1 and phase 2 to find optimal configuration of the complete dataflow when targeting an FPGA mainly for emulation or prototyping given the optimal results obtained for an ASIC, which are obtained in phase

1. The proposed method is based on a compositional method to quickly converge to the optimal Pareto-front using FPGA area (in term of slice count) and dataflow performance and is itself composed of two main steps. **Step 1** sorts all the Pareto-optimal micro-architectures based on their latency. **Step 2** iteratively builds dataflows by selecting the most optimal micro-architecture for each module based on the number of limited pins for each module for the FPGA based on three facts: 1) the overall latency of the dataflow system is decided by the module which has the largest latency, 2) the total area of the dataflow system equals the summation of the area of all individual modules, 3) the I/O ports count of two adjacent modules must be the same. Algorithm 3 summarizes the compositional approach.

Algorithm 3: Compositional method algorithm

Input : $D = \{D_1, D_2, ..., D_n\}$
$D^{io} = \{D_1^{io}, D_2^{io}, ..., D_n^{io}\}$
// D_i: Pareto front of module i
// D_i^{in}: Pareto front of module i given a certain number of inputs
// D_i^{out}: Pareto front of module i given a certain number of outputs
Output : $C = \{C_1, C_2, ...\}$
// C_i: Configurations leading to one Pareto optimal dataflow

1 /* **Step 1**: Sorting micro-architecture by Latency */
$SortedLatencyList \Leftarrow Sort(D_1(L) \cup D_2(L) \cup ... D_n(L))$;
2 /* **Step 2**: Building Dataflows */ **for** $d \in SortedLatencyList$ **do**
3 Add d to C_i;
4 $I = d(in), O = d(out), L = d(latency)$;
5 Find that d is from module m;
6 **for** $m_b \in$ *modules before* m **do**
7 $I = max(\{I | I \in D_{m_b}^{out=I}, I \leq L\}$;
8 Find d_b corresponding to I and add d_b to C_i;
9 $I \Leftarrow d_b(in)$;
10 **end**
11 **for** $m_a \in$ *modules after* m **do**
12 $I = max(\{I | I \in D_{m_a}^{in=O}, I \leq L\}$;
13 Find d_a corresponding to I and add d_a to C_i;
14 $O \Leftarrow d_a(out)$;
15 **end**
16 Add C_i to C;
17 **end**

The inputs to this phase are the results for each module in the dataflow when the FPGA is targeted and which are obtained in step 2 of phase 2 (D^F). Each design $D_i^F\{area, latency, in, out\}$ is fully characterized by its area, its latency in clock cycles

and the numbers of inputs and outputs. The result of this phase is a trade-off curve composed of Pareto-optimal dataflow configurations. Each configuration composed of different mixes of micro-architectures for each module in the dataflow.

Step 1: Sorting Micro-architectures based on Latency: The main intuition behind this step is that the dataflows throughput is limited by the module with the longest latency. As mentioned previously, we only consider dataflows which directly connect the different modules. Hence, our method has to ensure that the latencies, which act as data ignition intervals (DIIs) between the modules, match. In this particular case, and considering that we also multiplex the pins, a given design $D_i = \{A, L, IN, OUT\}$ will also have a specific number of IO pins. This reduces the search space as this module can only be connected with a previous module that has the same number of input pins $D_i(IN) = D_{previ}(OUT)$ and to another module with the same number of output pints $D_i(Out) = D_{nexti}(IN)$. The output of this phase is $SortedLatencyList = \{d_1, d_2, ...\}$.

Step 2: Building Dataflows: This second step takes iteratively the design with the smallest latency from the sorted list generated in step 1 (*SortedLatencyList*) and finds the smallest micro-architectures of the neighboring modules in the dataflow with latency equal or smaller than that micro-architecture in the sorted list. Smaller latencies are allowed as it is easy to add delays at the outputs in the form of *buffers*. As mentioned previously the micro-architectures' candidates are limited by the interconnect between each module pair in the dataflow. Thus, only those micro-architectures with the same number of pins are considered. For each new configuration, the total area is computed by adding up the area of all the micro-architectures in the dataflow and the latency by adding these up.

This step is repeated until all the micro-architectures from the sorted latency list are considered. The results is a list of predicted Pareto-optimal dataflow configurations C for an FPGA, where each design is characterized by a unique set of pragmas and number of IOs, which in turn lead to a specific area and latency.

6.5 The Comparative Methods

This section discusses the two comparative methods which are also presented in Fig. 6.2.

6.5.1 Baseline Method

This method (1 - 4 - 7 - 10) performs full DSE targeting FPGAs for every module and sends the trade-off curves to the compositional method. Therefore, it is the most accurate but takes the longest running time in this work. This will be the reference method in this paper, and it is denoted as *Base*.

6.5.2 Direct Mapping

This method (1 - 2 - 3 - 7 - 6 - 9 - 10) uses the existing DSE results for ASIC platforms and feeds them to the compositional method to obtain the trade-off curve for the data-flow system targeting an ASIC. Next, it extracts the configurations of the designs on the trade-off curve and re-synthesize them targeting FPGA platforms. This method is based on the assumption that the Pareto optimal designs for an AISC would also yield the trade-off curve on FPGAs, which is not true and is the problem we are trying to solve. We denote this method as *DM*. Note that *DM* is faster but will be less accurate than *Base*. As shown in Fig. 6.2, *DM* also shares most of the blocks with the proposed method except blocks 5 and 8. The two blocks perform the prediction of individual modules. Therefore, given a pre-trained model, the running times of blocks 5 and 8 are trivial compared to the running time of block 2. Thus the proposed method would run as fast as *DM*.

6.6 Experimental Results

To prove the accuracy and efficiency of the proposed method, this section firstly describes the experimental environment used in this work. Then it presents the extensive experiment results produced by the three methods.

Table 6.1: Modules of Data Flow Applications

	S1	S2	S3	S4	S5	S6	S7	S8	S9
average	1		1	1	2		4	1	1
adpcm_encoder	2	2	2		1	1	1	2	3
decimation				3		2		5	5
fir		1	3			3	5	3	4
interpolation				2	4	4	3	4	6
sobel					3		2		2

6.6.1 Experimental Setup

To prove the effectiveness of the proposed framework, six benchmarks from the open source Synthesizable SystemC benchmark suite S2CBench v.2.2 [101] are taken. Combining some of them produces nine data-flow applications as shown in Table 6.1. For example, S3 has three modules, *average* is the first module which is connected to *adpcm_encoder*, and *fir* is the last module. The HLS tool used is CyberWorkBench from NEC [78], the target ASIC technology is Nangate 45nm, and the target FPGAs are Xilinx Virtex 7.

6.6.2 Results

The quality of the exploration result is measured using the ADRS, which is a widely used metric to compare multi-objective optimization problems like this one. ADRS indicates the average distance between the reference Pareto-front and the approximate Pareto-set *i.e.*, tells how close a Pareto front is to the reference front, where the reference front is obtained from the actual HLS DSE for the targeted FPGA (FPGA synthesis is performed). In particular, given a reference Pareto-front Γ and an approximate Pareto set Ω, ADRS is computed as shown in (6.1) (also defined in (5.1)).

$$ADRS(\Gamma, \Omega) = \frac{1}{|\Gamma|} \sum_{\gamma \in \Gamma} \min_{\omega \in \Omega} f(\gamma, \omega) \tag{6.1}$$

Another measuring metric is *Dominance*, which indicates the ratio of the number of correctly predicted Pareto optimal designs to the total number of the exact Pareto set.

Fig. 6.4 presents the Pareto optimal trade-off curves found by the three methods

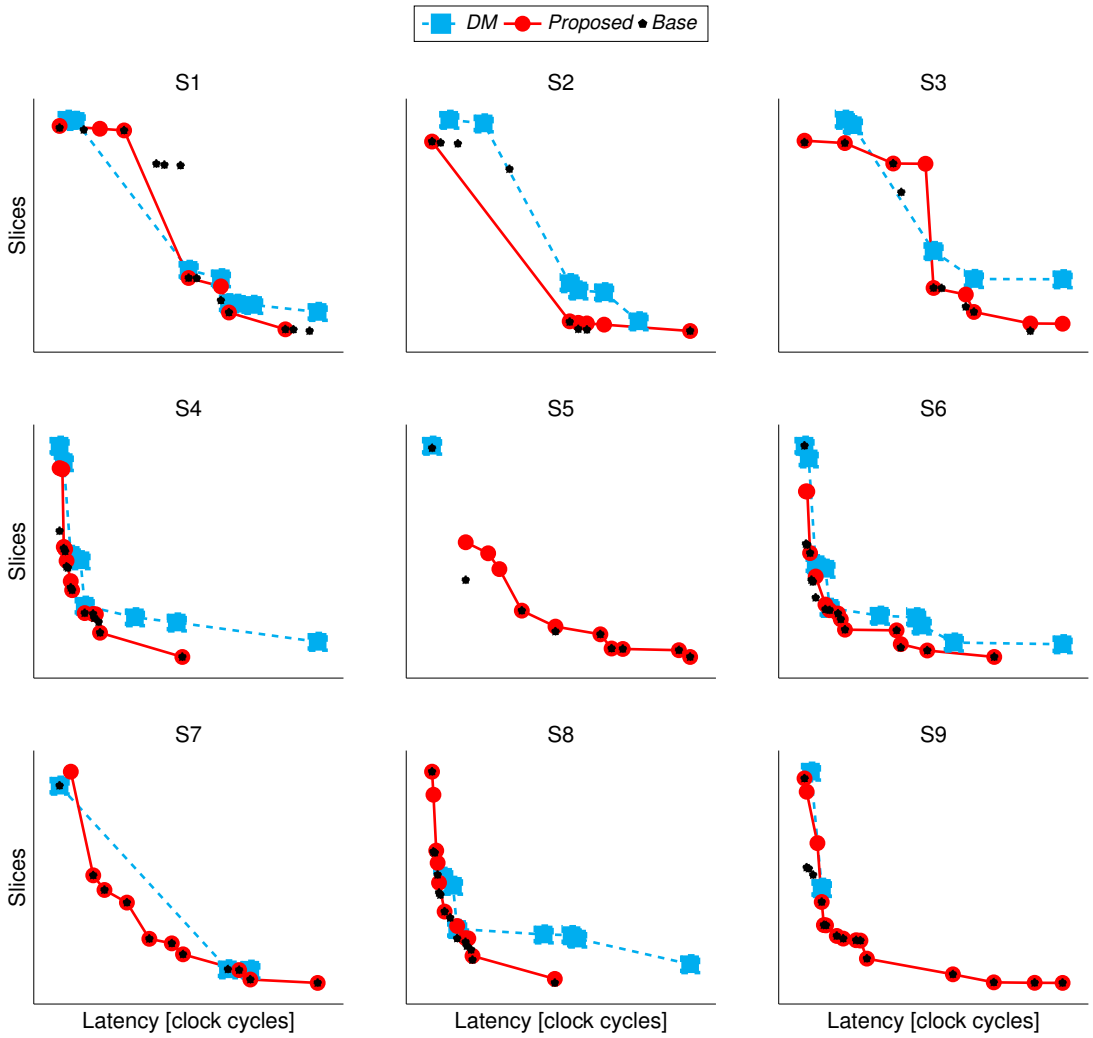


Figure 6.4: The trade-off curves of different methods for the nine dataflow systems

for different data-flow systems. For example, regarding S8, the black dots (FPGA) represent the accurate Pareto optimal designs that S8 could produce in this work, so this set is the reference set. The red curve (Prediction) indicates the Pareto optimal designs found by the proposed method, and the more overlapping with the black curve (FPGA), the better the red curve (Prediction) is. Similarly, the blue curve (ASIC) is the result produced by Direct Mapping, which is able to find some exact Pareto optimal designs but performs not as well as the proposed method.

Table 6.2 shows the comparison of the three methods in detail. The baseline method produces the exact Pareto optimal designs, thus it yields 0% ADRS and 100% dominance, and its results will not be shown. It can be observed that the proposed method outperforms the naïve method in both quality metrics. On average, the

proposed method improves ADRS by 92.51%, and the dominance is increased by 1674.17%.

Table 6.2: Methods Measurement [%]

	ADRS		Dominance	
	ASIC	Proposed	ASIC	Proposed
S1	7.84	2.49	0.00	30.77
S2	14.57	1.98	0.00	25.00
S3	23.36	2.17	0.00	44.44
S4	19.08	2.44	0.00	38.46
S5	185.22	4.39	0.00	66.66
S6	14.55	4.10	6.25	50.00
S7	22.51	0.86	18.18	81.81
S8	20.30	4.41	0.00	14.29
S9	39.57	3.20	0.00	81.25
Ave	38.56	2.89	2.71	48.08

These results indicate that the proposed method works well and that it leads to better results than the other method. The additional advantage of our proposed method is that it does not require any new training phase for each new design to be converted, and thus, is very fast (as shown in Table 6.3). The model generation phase takes less than 1 second, while the actual re-exploration of the design takes on average less than 1-minute making use of the new predictive method. In contrast, the average running time of the GA-based HLS DSE, having to fully re-synthesize each new design configuration targeting the desired FPGA, is 26.5 hours. This exploration running time is obviously a function of the complexity of the benchmark, *e.g.* the longest dataflow exploration took 84.1 hours for S9.

6.6.3 Case Study: JPEG Encoder

In this case study, the proposed method is also applied on JPEG encoder, which is a well-known streaming application in image processing. Fig. 6.5 presents the Pareto-optimal trade-off curves obtained by the three methods. We can see that the proposed method is able to find the trade-off curve which is very close to the exact one. As a result, the metric values of ADRS and dominance are 1.16% and 57.14% for the proposed method, while they are 62.93% and 14.28% for the direct mapping method.

Table 6.3: Running Time Comparison

	DM [in seconds]	Proposed [in seconds]	Base [in hours]
S1	<1	18.8	3.0
S2	<1	19.4	4.1
S3	<1	19.2	4.4
S4	<1	25.8	56.6
S5	<1	29.8	81.6
S6	<1	14.3	60.4
S7	<1	18.8	83.0
S8	<1	38.3	60.7
S9	<1	38.6	84.1
Geomean	<1	23.5	26.5

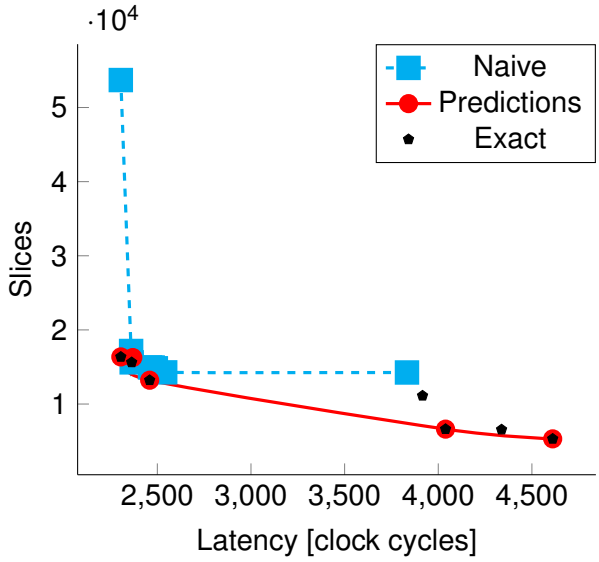


Figure 6.5: JPEG encoder case study

6.7 Summary and Conclusion

In this work he have proposed a method based on compositional predictive models to quickly and effectively map complex data flows optimized for ASICs to FPGAs of different underlying structure. This enables the faster verification of complex SoCs that make heavy use of hardware accelerators often organized in data flow structure. Experimental results have shown that our proposed method is very effective vs. having to fully re-explore the extremely large search space to find Pareto optimal data flow configurations at a fraction of the total running time.

Chapter 7

Time-multiplexed Reconfigurable Dataflow

As we have shown throughout this thesis, any computationally intensive applications are implemented on hardware following the stream computing, also called dataflow computing, paradigm. This entails that data is streamed through different components of a given application in wide deep pipelines to maximize throughput. One of the main drawbacks of this computing paradigm is that it consumes a large number of hardware resources. Thus, in this chapter, we propose an exploit the capabilities of state of the art FPGAs to perform partial runtime reconfigurable. Meanwhile, we propose a hardware overlay onto which to map any computationally intensive application given as a behavioral description for HLS composed of multiple stages. These designs would typically fit the stream computing paradigm. This overlay uses the internal's FPGA BlockRAM to store the intermediate results of each stage to speed up the computation and time-multiplexes the different stages by reconfiguring the computational part.

This work also includes a design methodology to optimize the implementation of each stage to balance the dataflow architecture as well as generating systems with unique area vs. performance trade-offs. The proposed architecture and methodology has been prototyped on a Xilinx Zedboard mounting a Zynq FPGA, and a case study of a JPEG encoder is presented highlighting the benefits of it. The overlay will be made public and open source after the publication of this work.

7.1 Introduction

FPGAs have regained much attention recently, mainly because they are starting to be extensively deployed in data centers operated by traditionally software companies [27, 76]. The main idea is to offload portions of applications, with a high amount of parallelism, from the CPU onto the FPGA. This increases the overall performance, while significantly reducing the power consumption. The acquisition of Altera by Intel is also mainly motivated by this, and Intel has recently announced that it is targeting the production of around 30% of the servers with FPGAs in data centers by 2020 [7]. At the same time, FPGAs are being used for High Performance Computing (HPC) to accelerate numerically intensive applications. Several startups have also appeared in this field, *e.g.* Falcon Computing [32] and Maxelar Technologies [68]. Both of these companies have in common the use of high-level languages as inputs to facilitate the use of their FPGA-based HPC systems. Concurrently, FPGA vendors are attempting to raise the level of abstraction to allow software programmers with limited hardware knowledge to program these devices, mainly using HLS. Hardware skills are still very much needed to program these devices and hardware skills are rare. The US Bureau of labor statistic reported in 2017 that the ratio of hardware engineers to software engineers had almost doubled since 2010 [2].

Most of the applications to be accelerated are mapped as stream computing, also called dataflow computing, architectures onto the FPGA, where data is *streamed* onto the FPGA and passed from one module to another through wide deep pipelines, reducing the need to access external memory [41]. Some applications that benefit from dataflow computing can be found at [67] and they include Monte Carlo simulations, matrix multiplications, and blockchain miners. This computing paradigm leads to extremely high throughput at the cost of large area usage. This implies that because companies try to maximize profits, they mostly use the smallest (cheapest) FPGA device that fits their design. Future updates (in the field) might not fit that device.

One additional problem is that the dataflow architectures have typically to be optimized manually. If the DII of the pipelines match, then each module can be connected directly together, and if not, FIFOs are typically used between the modules. This work

tries to address this by proposing a partial runtime reconfigurable overlay, specifically designed for dataflow applications that time-multiplex the execution of each of the modules, thus, reducing the total area required, and making use of the FPGA's internal BlockRAM to store the partial results of each stage. We also present a methodology to optimize the micro-architecture of each dataflow stage by generating the best micro-architectures for each dataflow stage that balance the area vs. performance. In summary, the main contributions of this work are:

- Propose a static overlay architecture onto which to map behavioral dataflow applications efficiently using BlockRAM to store intermediate stage results.
- Introduce a methodology to find the best mixes of micro-architectures for each module in the dataflow to generate systems with different area vs. performance trade-offs.
- Present extensive experimental results, prototype the proposed overlay architecture onto a Xilinx Zynq FPGA and compare it against a static implementation and another overlay-based implementation.

7.2 Motivational Example

Fig. 7.1, shows a motivational example for this work. It shows the four main steps in the JPEG encoding process (DCT, quantization, RLE and Huffman encoding) laid out spatially as a dataflow, where each step is described as a behavioral description given in *e.g.* C or C++. Because of this, and considering that HLS is a single process synthesis method, each description is synthesized and optimized individually leading to a unique micro-architecture. In this case, the input is an image to be encoded, and the output is the compressed image. As shown in the figure, the data is streamed from one stage to another.

Also, shown in Fig. 7.1, it can be observed that HLS allows the generation of a variety of micro-architectures with unique area vs. performance and/or power overheads. This is done by specifying different synthesis options, typically in the form of synthesis directives (pragmas) that control how to synthesize arrays (*e.g.* as registers or RAM), loops (unroll, not-unroll, partially unroll or pipeline) and functions (inline or not). Out of

all the micro-architectures, the most important ones are the Pareto-optimal ones that form the area vs. latency trade-off curve in Fig. 7.1.

Also, when mapping this application on a runtime reconfigurable FPGA which time-multiplexes each stage, the area of the final circuit is determined by the largest micro-architecture of four stages.

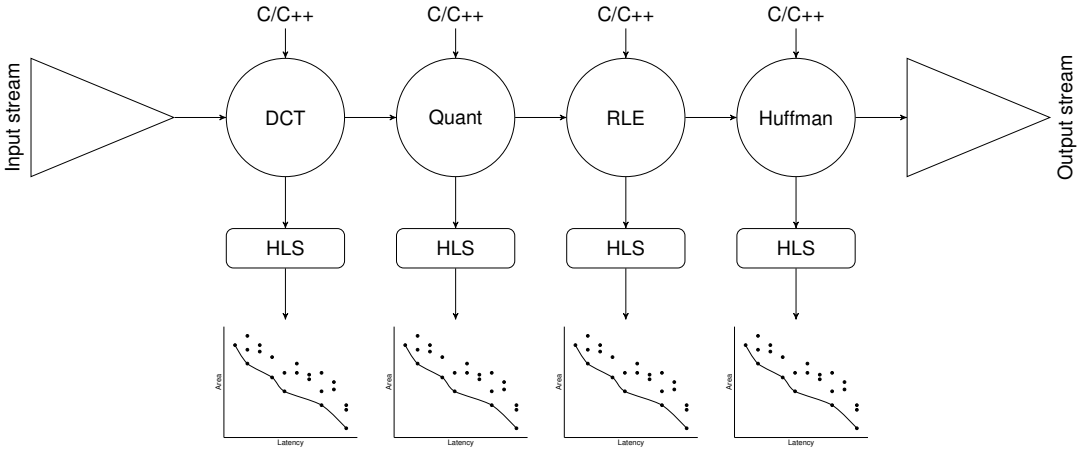


Figure 7.1: JPEG encoder steps laid out spatially as dataflow and individual description of optimization using HLS.

Based on this, the objectives of this work are two-fold: First, we create a runtime reconfigurable hardware overlay that facilitates the execution of dataflow applications, like the JPEG encoder, to reduce the area utilization, while minimizing the performance degradation due to the reconfiguration. Secondly, we develop a methodology to find the best mix of micro-architectures that leads to Pareto-optimal overall system configurations with unique area vs. performance trade-offs.

7.3 Related Work

FPGA overlays have been proposed in the past, mainly to hide the intricacies of the FPGA, while allowing to benefit from the massive amount of parallelism that can be leveraged when mapping applications on them. In [12] the authors present a Virtual Dynamically Reconfigurable (VDR) overlay that consists of an array of functional units interconnected by programmable switches and present a Just-in-time (JIT) compilation flow to map applications onto this overlay. A soft process overlay tightly coupled with hardware accelerators is presented in [82]. In [11], the authors introduce an overlay for

data flow graphs (DFGs) mapping these onto a mesh of functional units.

Closer to our work, the authors in [15] make use of partial reconfiguration to map dataflow applications onto an FPGA. The authors also present a flow to partition applications and map them onto the FPGA. In their work, no fix overlay is used.

It should be noted that our work assumes that the application to be mapped onto the FPGA has already been partitioned into individual modules and that each module is complex enough that it requires being synthesized (HLS) individually. Much work has been done in the past in the area of automatic partitioning of applications to be mapped onto FPGAs [96, 105, 74].

Our proposed work is different from previous work in multiple dimensions. First, our work is based on an overlay onto which to map any stream computing application specified as a set of behavioral descriptions (one description for each module). Secondly, the overlay is optimized to avoid external memory accesses by using the FPGA's BlockRAM to store the intermediate results. Finally, we propose a methodology to refine the system by generating a variety of different micro-architectures for each module in the dataflow. This method, combines with an analytical performance and area model, allows performing a complete design space exploration of the application mapped onto the overlay.

7.4 Proposed Partial Runtime Reconfigurable Overlay

Fig. 7.2 shows the block diagram of the proposed overlay architecture and the JPEG encoder used as a case study. The overlay has been implemented on a Programmable SoC FPGA (Xilinx Zynq) which contains a dual-core ARM Cortex-A9 processor and reconfigurable fabric onto which the overlay is mapped. In the case of the JPEG encoder, this application can be divided into four distinct PEs, with PE_1=*DCT*, PE_2=*Quantization*, PE_3=*Run-Length Encoding (RLE)*, and PE_4=*Huffman encoding*. The *PEs* are time-multiplexed on the dynamic region of the overlay. In particular, the overlay consists of four main parts:

1. The dynamic place and route region (*P-Block*), onto which the dataflow applica-

tion is mapped.

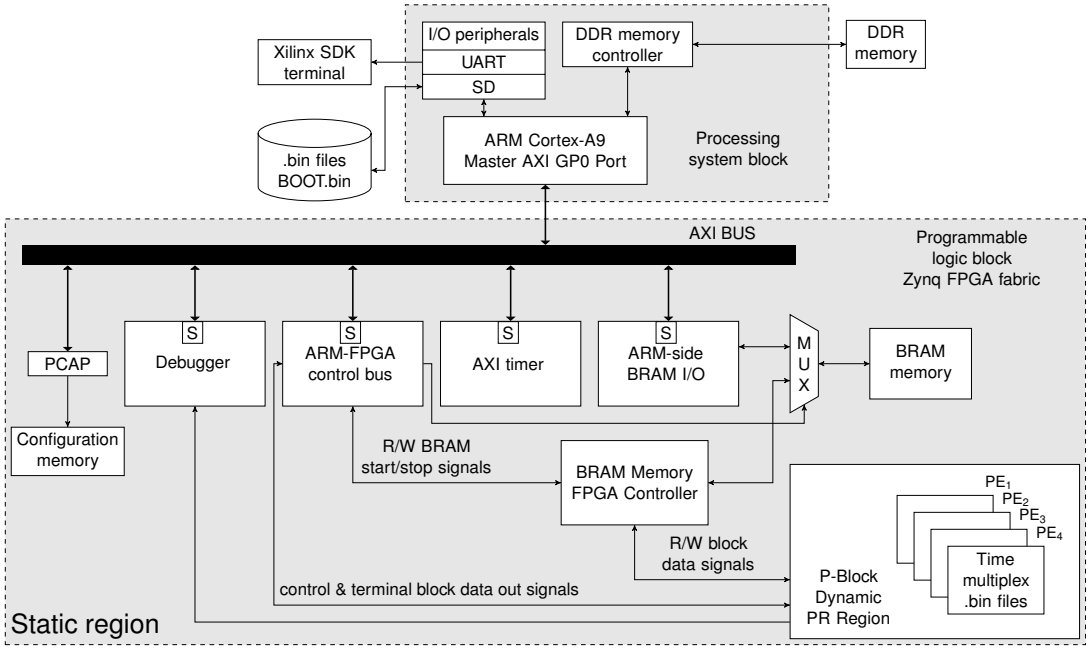
2. The BlockRAM (BRAM) memory which stores the input data as well as the intermediate results generated at each dataflow execution stage.
3. The BRAM controller which controls if the ARM processor or the application accesses the BRAM and is responsible for writing and reading data into the BRAM.
4. The processor configuration access port (PCAP) and configuration memory, which stores all the partial bitstreams of the dataflow application and necessary logic to reconfigure the P-Block

The sequence of operations follows three main steps.

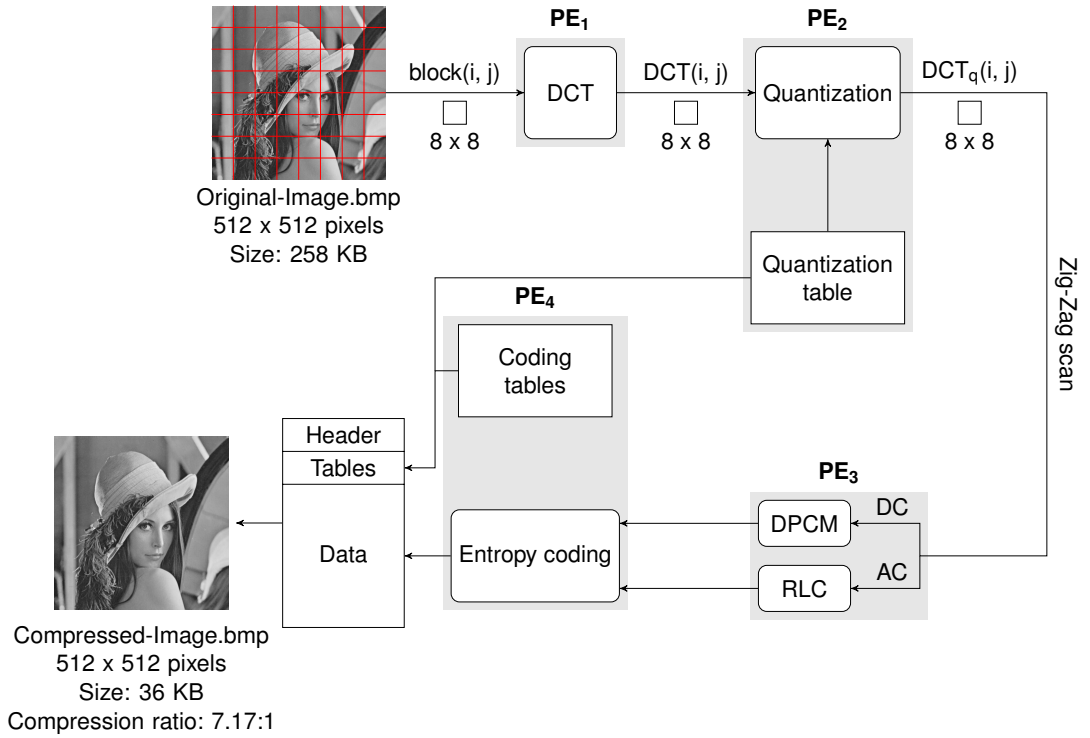
Step 1: Load configuration bits onto FPGA: The ARM processor loads the bitstreams for each partition (.bin files) from the SD card to the DDR memory of the FPGA board and from there through the PCAP interface to the reconfigurable fabric.

Step 2: Initial configuration and data loading: The ARM processor configures the *P-Block* with the very first stage of the application and transfers the input data from main memory (DDR) to the internal BRAM to be processed. In the case of the JPEG encoder, the input data is the image to be encoded. It should be noted that due to the BRAM size limitation, the image might have to be partitioned into blocks of data that fit the BRAM, which is, in turn, a function of the FPGA selected. In the experimental results section, we present extensive results of how the partition of the data impacts the throughput.

Step 3: Execution of application: Once the data is loaded onto the BRAM, the ARM processor, which acts as a controller, sets the start signal to start the computation and the BRAM controller starts loading the data from the BRAM to the processing units. Initially, the *P-Block* is configured with the functionality of PE_1, which in the JPEG case is the DCT. The results generated by the DCT are stored as they are generated into the BRAM. This step continues until all of the input data is processed and the outputs generated. A finish signal is sent to the processor once the DCT stage finishes, which in turn reconfigures the *P-Block* with the next stage in the dataflow, in this case, PE_2, which performs the quantization of the results generated by the DCT stage.



(a)



(b)

Figure 7.2: Overlay architecture and JPEG partition mapped onto it.

This process continues by reconfiguring the *P-Block* with PE_3 (RLE) and finally PE_4 (Huffman encoding), after which the processor reads the data stored in the BRAM as the final result. In the case that the BRAM is not large enough, this process has to

be repeated multiple times. This impacts the running time and hence the throughput, but at the same time allows to control the latency, where the latency can be defined as the time it takes to generate new output. By splitting the application into smaller pieces, the total running time will increase, but the latency decreases. Fig. 7.3 shows some experimental results when the input image is partitioned into 2, 4, 8, 16, and 32 blocks, highlighted in the number of samples graph, where each sample is an 8×8 block. Thus, a large number of samples implies more data being processed in each iteration. This impacts the total number of reconfigurations required to encode the full image as shown in a number of reconfiguration graph from 8 to 16, 32, 64, 128 and 256 respectively. Fig. 7.3 also shows how this affects the total running time. The more reconfigurations, the longer the running time, which doubles with the number of reconfigurations.

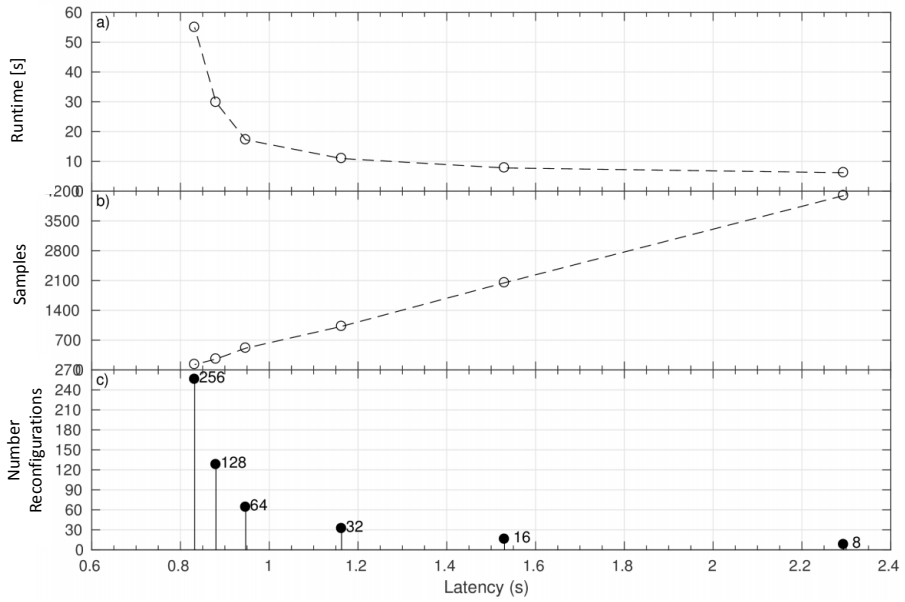


Figure 7.3: Impact on runtime on samples size (partitions from image) and number of reconfigurations.

A simple predictive model can be created based on these results to predict the area (A_{total}) and the execution time (T_{run}) of any dataflow computation mapped onto the overlay. In particular:

$$A_{total} = A_{static} + A_{dynamic}\{max(PE_1, ..., PE_i)\} \tag{7.1}$$

where, A_{static} is the static portion of the overlay, and $A_{dyanmic}$ is the part mapped on the reconfigurable module, which is determined by the largest module mapped on it. Moreover,

$$T_{run} = T_{application} + T_{overhead} + T_{reconf} \times N_{reconf} \tag{7.2}$$

where $T_{application}$ is the time it takes to run the application when the dataflow is laid out spatially. Note that the two predictive equations would also apply to other dataflow applications, because the reconfigurable implementation steps apply to other applications, which leading to similar approaches of area and running time measurements. In the case of JPEG encoding for the 512×512 image, $T_{application}$=2.99s. $T_{overhead}$ is the time it takes to load all the partial reconfigurable bit files (.bin files) from the SD card to the FPGA and can be ignored for applications that execute repeatedly. For the JPEG encoder case we measure $T_{overhead}$=1.68s. Finally T_{reconf} and N_{reconf} are, respectively, the time it takes to reconfigure the *P-Block* and the number of required reconfigurations, which in this case is measured to T_{reconf}=0.20s and as shown in Fig. 7.3 with N_{reconf} depending on the input data partition.

One observation that can be quickly made is that the total running time depends on the time it takes to reconfigure the *P-Block*, which is, in turn, a function of the size of the *P-Block*. The larger it is, the longer it takes to reconfigure. Thus, it is imperative to reduce the P-Block size to the smallest possible size, which in turns depends on the largest module in the dataflow.

7.5 Dataflow Design Space Exploration

As shown in Fig. 7.1, one of the advantages of using behavioral descriptions to create dataflow applications is that a large number of unique micro-architectures for each stage in the dataflow can be automatically generated. This, in turn, implies that by mixing different micro-architectures, complete dataflow systems with unique characteristics can be created.

Based on this, this work proposed a method to find a set of Pareto-optimal dataflow configurations given a behavioral description for each of the dataflow stages. This flow is based on two stages. The first performs a detailed HLS design space exploration

for each individual stage in the dataflow, while the second makes use of compositional design techniques based on the predictive models of the total area and runtime introduced in the previous section (equations (7.1) and (7.2)) to find a trade-off curve of dataflow configurations with unique area vs. performance characteristics. In particular:

Step 1: Single Module HLS DSE: HLS is a single process synthesis method, thus, every module that composes the dataflow needs to be explored separately. Much work in the past has focused fast heuristics on performing HLS DSE. They can be coarsely classified into synthesis-based methods [121] and predictive-based methods [62]. The synthesis-based methods generate new configurations based on the available exploration *knobs*, typically synthesis directives in the form of pragmas, and synthesize each new configuration from which they extract the different design metrics. Based on the results, some of the exploration *knobs* are adjusted and the process repeated until a given exit condition is reached. In the second case, predictive-based methods, this same process is repeated for a specific training set, after which a predictive model is created. These methods, then continue with the exploration using the predictive model to quantify the impact of new exploration *knob* settings, and thus, avoiding having to synthesized each new configuration.

Because of the simplicity, and considering that the micro-architectural explorer is not part of the main contribution of this work, we adopt a synthesis-based exploration method based on a genetic algorithm, which has shown to lead to outstanding results in these types of multi-objective optimization problems [31].

In summary, each explorable operation (loop, arrays, and functions considered in this work) is represented as a gene to which a synthesis attribute (pragma) is assigned. The list of all genes builds a chromosome *Cr*. The genetic algorithm first step is the random generation of an initial population of *N* chromosomes. Then, each member of the population is paired with another randomly selected member from the population and produces an offspring using the crossover operator. The crossover operator selects a cut-point at random and combines the left half of one parent with the right half of the other. The crossover operates only a certain percentage of the time, according to the specified crossover rate r_c specified beforehand by the user. The offspring is then

mutated, which involves randomly selecting one gene within the offspring chromosome and changing it to another random value. Only a certain percentage of the offsprings produced are mutated based on the mutation rate, r_m. In this work r_c and r_m are set to 0.8 and 0.1 respectively.

At this point, the mutated offspring is synthesized calling the HLS tool and the area and latency of the synthesized design back-annotated, hence leading to a new design. The newly generated offspring will replace one of the parents if one of the several conditions is met: (i) one of the parents is dominated by the offspring (i.e., is inferior to the offspring across all of the objectives); (ii) the offspring improves on one or more of the best-so-far values (the parent to be replaced is randomly chosen). The algorithm will continue until N child designs do not improve any of the parents.

As indicated, much work has been done in this area in the past, and it is out of scope to *e.g.* prove the optimality of the solution. In this work, the main objective of the explorer is to find a trade-off curve of dominating designs as a proof of concept for the proposed hardware overlay.

Step 2: System-Level DSE: This second step takes as inputs the exploration results obtained from step 1 for each of the modules composing the dataflow. In the case of the JPEG encoder, this would be four trade-off curves (DCT, Quantz, RLE, and Huffman). It then continues by applying a compositional design technique based on an incremental greedy algorithm to find the overall system-level trade-off curve. Algorithm 4 summarizes this step.

The proposed method starts by sorting the designs/micro-architectures in each trade-off curve based on the area (line 1). It then goes through the list of designs; chooses the smallest micro-architecture between all the trade-offs and adds this design to a new dataflow configuration, DF_i (line 2-4). The search then continues by finding the micro-architecture with the smallest area difference compared to DF_i for all the other PEs (lines 6 to 8). Because the area of this initial micro-architecture is the smallest, the new ones should inevitably have a larger area. Once a micro-architecture for each PE stage has been found, our method continues by using the predictive models for area and execution time described in equations (7.1) and (7.2) to evaluate the effect

Algorithm 4: Proposed compositional design technique based on incremental greedy algorithm.

Input : $DList = \{PE_1 = \{D_1(A, L), ..., D_x\}, ..., PE_n\}$
// $DList$: Designs list for different processing element (PEs)
// PE_n: Different processing elements in dataflow
// D_x: Optimal design for PE
// A: Design area; L: Latency;
Output : $DFList_{opt} = \{DF_1(A_s, T_{run}), ..., DF_X\}$
// $DFList_{opt}$: Pareto-optimal system-level dataflow list
// A_s: System area; T_{run}: execution time;

```
1  sort_by_area(DList);
2  while (DList) do
3      D_PEi = extract_min_area_design(DList);
4      DF_i = add_to_DF(D_PEi);
       // choose the next smallest design for other modules
5      foreach PE_j ∈ DList , PE_j != PE_i do
6          D_PEj = extract_Δ_A_smallest(D_PEi(A));
7          DF_i = add_to_DF(D_PEj);
8      end
9      DF_i[A_s] = estimate_area(DF_i);
10     DF_i[T_run] = estimate_time(DF_i);
11     DFList = add_to_DFList(DF_i[A_s, T_run]);
12 end
13 DFList_opt = remove_non_optimal_DFs(DFList);
14 return DFList_opt
```

of this micro-architecture mix on the complete system (lines 9-10). This new dataflow configuration (DF_i) is then added to the dataflow configuration list ($DFList$) generated so far ($DFList$) (line 11) and the search continues by extracting the next smallest design from all the trade-off curves. This process continues until a configuration with the largest but fastest micro-architectures of each PE is generated.

The last step analyzes all the results obtained and deletes all non-Pareto-optimal configuration and returns only the dataflow configurations that are optimal, $DFList_{opt}$ (line 13-14).

7.6 Experimental Results - JPEG encoder Case Study

The analysis of the proposed overlay and dataflow design space exploration methodology is presented using the JPEG encoder discussed in the previous section as a case study. The design has been taken from the open source Synthesizable Sys-

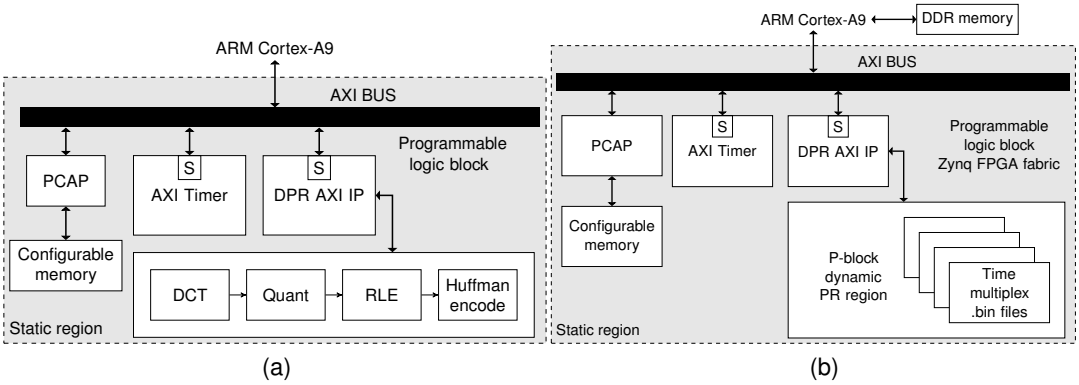


Figure 7.4: Additional two implementations with which we compare our proposed overlay shown in Fig. 7.2.(a) Static implementation of JPEG encoder following a traditional dataflow (b) Runtime reconfigurable alternative storing intermediate results onto the DDR

temC benchmark suite S2CBench v.2.2 [101], which is given as four separate modules (sc_modules). The HLS tool used is CyberWorkBench from NEC [78] and the FPGA a Xilinx Zynq 7000 SoC FPGA mounted on a Digilent Zedboard board. The experiments are conducted on an Intel i7-6700 3.50GHZ CPU and 16 GB memory, running CentOS 7.

Two reference implementations have been additionally created to fully characterize our proposed method. Fig.7.4 shows their block diagrams. Fig.7.4(a), shows the first reference implementation, which follows a traditional dataflow computation model where the JPEG encoder is fully laid out spatially on the reconfigurable fabric. We call this, spatial implementation. Fig.7.4(b) shows the second reference design, which we call PR_{DDR}. It is based on the same runtime reconfigurable overlay but uses the external DDR memory to store the intermediate results of each dataflow stage. Thus, no BlockRAM is used in this case. The first reference implementation should intuitively lead to a large area overhead while having the highest throughput, whereas the second reference design (PR_{DDR}) should lead to the slowest but the smallest configuration.

Fig. 7.5 shows the design space exploration results of each of the four JPEG stages, while Fig. 7.6 shows the result of the system-level exploration results. This last figure also compares the quality of the proposed iterative greedy method with a brute force approach, which guarantees to find the optimal solution. From the figure, it can be observed that the proposed method can find most of the optimal system configurations

while being 328× faster than the brute force method(1 second vs. 328 seconds). Note that Fig. 7.6 shows running time in seconds which involves the latencies as shown in Fig. 7.5.

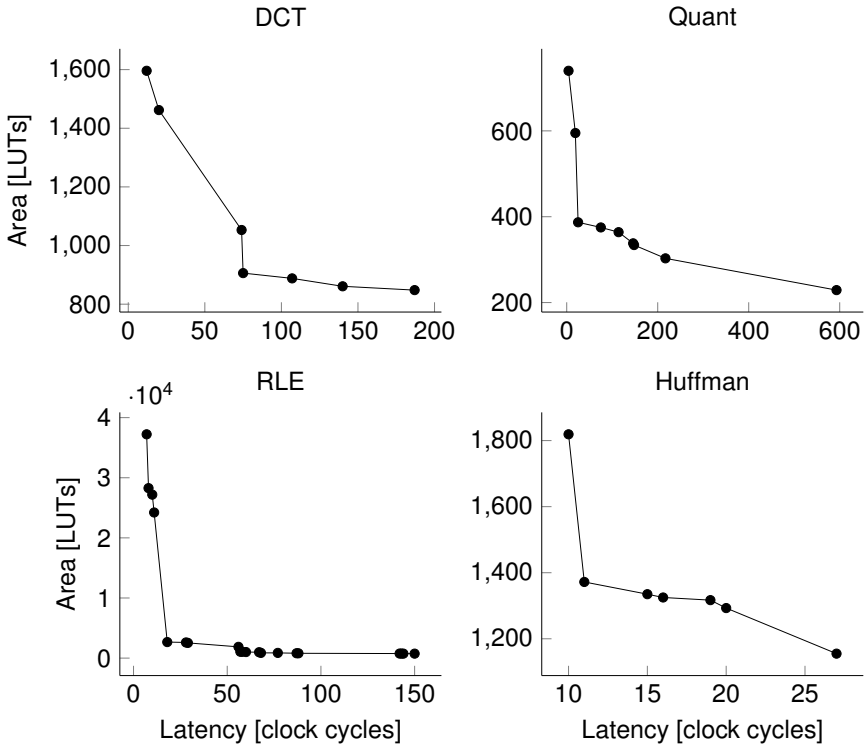


Figure 7.5: DSE exploration results for different JPEG modules.

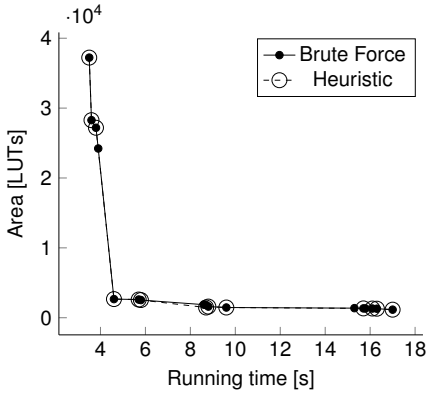


Figure 7.6: JPEG encoder system-level DSE result for proposed overlay

Table 7.1 shows the utilization report of the three implementations, where in this case the micro-architectures of largest area/smallest latencies for each JPEG stage are used. It can be observed that the results match the intuition. The spatial implementation uses 28.19% of the FPGA's LUTs, while the two implementations, based

on the proposed overlay, require only 20.32% and 23.25% for the P_{DDR} and P_{BRAM} respectively, which represents 27.9% and 17.52% area reduction. Although significant, it is far from the theoretically expected area reduction of 3/4 that we would expect by time-multiplexing the four modules. The main reasons for this are that as shown in (7.1), the total area is bounded by the largest module in the dataflow; also, we have to account for the fixed area overhead of circuitry that enables the runtime reconfiguration. This overhead is nevertheless constant and becomes, therefore, more or less important depending on the size of the application mapped to the overlay.

Table 7.1: Utilization report of 2 reference JPEG implementation (Spatial and PR_{DDR}) and proposed Overlay (PR_{BRAM})

Resource		Spatial Implementation		PR_{DDR} Implementation			PR_{BRAM} Implementation		
Type	Available	A_{static}	Utilization [%]	A_{static}	$A_{dynamic}$	Utilization [%]	A_{static}	$A_{dynamic}$	Utilization [%]
LUT	53200	14997	28.19	4119	6692	20.32	5681	6692	23.25
Flip-Flops	106400	24759	23.27	6018	9432	14.52	12967	9432	21.05
BRAM Tiles	140	2	1.43	-	3	2.14	128	3	93.57

Finally, Table 7.2 compares the three implementations based on execution time to encode different images. For the two runtime reconfigurable implementations based on the proposed overlay (PR_{DDR} and PR_{BRAM}), different blocks sizes (input image partitioning), which lead to a different number of reconfigurations are also investigated. The geomean value shows the average runtime values (geomean takes into account the benchmark size difference). Two main observations can be made from the results. The first is that the spatial implementation is the fastest. Compared to PR_{DDR}, on average, by 2, 2.5 and 3$\times$ and compare to PR_{BRAM} by 1.6, 1.77 and 2.55$\times$ for the three different block sizes (4, 8 and 16) respectively. When comparing the two runtime reconfigurable implementations, as expected the version that uses the BRAM to store the results of the intermediate stages is on average 30% faster.

Table 7.2: Execution time [s] of 3 JPEG implementations

	Spatial	PR_{DDR}			PR_{BRAM}		
Nreconfig	0	4	8	16	4	8	16
lena	1.82	3.75	4.40	5.68	2.30	3.08	4.62
pepper	1.84	3.86	4.71	5.71	2.38	3.28	4.73
goldhill	1.90	3.95	4.84	6.18	2.61	3.51	4.91
Geomean	1.85	3.85	4.65	5.85	2.98	3.29	4.75

Based on these results we can conclude that the introduced overlay proposes an interesting trade-off between area savings and performance degradation. It should be noted that the larger the dataflow is, the smaller the area overhead due to the fixed logic framework becomes and thus, larger area savings are expected.

7.7 Conclusion

In this work we have introduced an overlay for a configurable SoC FPGA that is optimized for behavioral dataflow applications. The overlay makes use of the FPGA's internal BlockRAM to store the intermediate results of every stage. We have shown, using a JPEG encoder as a case study, that the proposed overlay works well. The overlay has been fully characterized against a static dataflow implementation of the JPEG encoder as well as a simplified overlay version that stores the intermediate results on external memory. We have also introduced a design space exploration flow that finds Pareto-optimal configurations of the application mapped onto the proposed overlay by making use of one of the main advantages of C-based VLSI design over traditional RTL-based: the ability to generate a variety of different micro-architectures from a single behavioral description. Results show that our optimization method works well and that we can generate a variate of different systems with unique area vs. performance trade-offs.

Chapter 8

Conclusions and Suggestions for Future Research

8.1 Conclusion

This thesis discusses challenges in the design and optimizations of dataflow systems and in particular when these are specified at the behavioral level. HLS is a proven technology that raises the level of VLSI design abstraction and increases design productivity. The main focus of this work is to leverage the advantages of HLS to optimize the dataflows as mentioned earlier. We first investigate how pin multiplexing affects the micro-architecture of a behavioral description for HLS and propose an optimization method to map logic ports together. This method is in turn used to build complete dataflow systems to reduce the congestion and interconnect delay in FPGAs. We also propose a method to automatically convert optimized designs for ASICs to FPGA designs. Finally, a full ASIC to FPGA dataflow optimization is proposed.

8.2 Future Work

High-level synthesis is developing rapidly. Given the promising future of HLS, more and more hardware IPs will be described in the behavioral level, especially in C or C++. At the same time, machine learning can be used to complement existing hardware design methodologies to provide quick estimates of the resultant designs without having to execute time-consuming steps. The design methodologies in this thesis take advantages of the two facts and focus on a particular part of a SoC, which is dataflow

hardware accelerators. However, the concepts presented in the projects are generic, which means they apply to other design situations without or with little adjustment. For example, the conversion from ASIC to FPGA could also be applied to other cases: the prediction a) from older-generation ASIC to newer one (e.g., from 45nm to 32nm), b) from older FPGA to newer one (e.g., from Virtex-5 to Virtex-7).

The hardware accelerators are only a small portion of a complex SoC. The ultimate goal would be to allow the quick optimization of complete SoCs fully described at the behavioral level. This should include communications on the bus between masters and slaves (dataflow accelerators).

Bibliography

[1] S. Amellal and B. Kaminska. “Scheduling of a control data flow graph”. In: *1993 IEEE International Symposium on Circuits and Systems*. Chicago, IL, USA: IEEE, 1993, pp. 1666–1669. DOI: 10.1109/ISCAS.1993.394061.

[2] *Architecture and Engineering Occupations : Occupational Outlook Handbook: : U.S. Bureau of Labor Statistics*. en-us.

[3] J. Babb et al. “Logic emulation with virtual wires”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 16.6 (June 1997), pp. 609–626. DOI: 10.1109/43.640619.

[4] Vaughn Betz and Jonathan Rose. “FPGA routing architecture: segmentation and buffering to optimize speed and density”. en. In: *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays - FPGA ’99*. Monterey, California, United States: ACM Press, 1999, pp. 59–68. DOI: 10.1145/296399.296428.

[5] S. Borkar. “Design challenges of technology scaling”. In: *IEEE Micro* 19.4 (Aug. 1999), pp. 23–29. DOI: 10.1109/40.782564.

[6] *Breaking The Three Laws > How many ASIC Gates does it take to fill an FPGA?* en-US.

[7] Jeff Burt. *Intel Begins Shipping Xeon Chips With FPGA Accelerators*. Apr. 2016.

[8] Mike Butts. “Logic emulation in the megaLUT era — Moore’s Law beats Rent’s Rule”. In: *2014 International Conference on Field-Programmable Technology (FPT)*. Shanghai, China: IEEE, Dec. 2014, pp. 1–1. DOI: 10.1109/FPT.2014.7082742.

[9] Cadence. *Stratus High-Level Synthesis*. en.

[10] R. Camposano. “From behavior to structure: high-level synthesis”. In: *IEEE Design & Test of Computers* 7.5 (Oct. 1990), pp. 8–19. DOI: 10.1109/54.60603.

[11] Davor Capalija and Tarek S. Abdelrahman. “A high-performance overlay architecture for pipelined execution of data flow graphs”. In: *2013 23rd International Conference on Field programmable Logic and Applications*. Porto, Portugal: IEEE, Sept. 2013, pp. 1–8. DOI: 10.1109/FPL.2013.6645515.

[12] Davor Capalija and Tarek S. Abdelrahman. “Towards Synthesis-Free JIT Compilation to Commodity FPGAs”. In: *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*. Salt Lake City, UT, USA: IEEE, May 2011, pp. 202–205. DOI: 10.1109/FCCM.2011.25.

[13] B. Carrion Schafer and K. Wakabayashi. “Machine learning predictive modelling high-level synthesis design space exploration”. en. In: *IET Computers & Digital Techniques* 6.3 (2012), p. 153. DOI: 10.1049/iet-cdt.2011.0115.

[14] Benjamin Carrion Schafer. “Probabilistic Multiknob High-Level Synthesis Design Space Exploration Acceleration”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35.3 (Mar. 2016), pp. 394–406. DOI: 10.1109/TCAD.2015.2472007.

[15] Riccardo Cattaneo et al. “A framework for effective exploitation of partial reconfiguration in dataflow computing”. In: *2013 8th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*. Darmstadt, Germany: IEEE, July 2013, pp. 1–8. DOI: 10.1109/ReCoSoC.2013.6581535.

[16] S. Chaudhuri, R.A. Walker, and J.E. Mitchell. “Analyzing and exploiting the structure of the constraints in the ILP approach to the scheduling problem”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 2.4 (Dec. 1994), pp. 456–471. DOI: 10.1109/92.335014.

[17] Tianshi Chen et al. “ArchRanker: A ranking approach to design space exploration”. In: *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. Minneapolis, MN, USA: IEEE, June 2014, pp. 85–96. DOI: 10.1109/ISCA.2014.6853198.

[18] Young-kyu Choi et al. “A quantitative analysis on microarchitectures of modern CPU-FPGA platforms”. en. In: *Proceedings of the 53rd Annual Design Automation Conference on - DAC ’16*. Austin, Texas: ACM Press, 2016, pp. 1–6. DOI: 10.1145/2897937.2897972.

[19] Michael D. Ciletti. *Advanced digital design with the Verilog HDL*. 2nd ed. Prentice Hall Xilinx design series. OCLC: ocn213835490. Boston: Prentice Hall, 2011.

[20] LegUp Computing. *High-Level Synthesis For Any FPGA | LegUp Computing*.

[21] J. Cong et al. “Architecture and Synthesis for On-Chip Multicycle Communication”. en. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 23.4 (Apr. 2004), pp. 550–564. DOI: 10.1109/TCAD.2004.825872.

[22] J. Cong et al. “Combining module selection and replication for throughput-driven streaming programs”. In: *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. Dresden: IEEE, Mar. 2012, pp. 1018–1023. DOI: 10.1109/DATE.2012.6176645.

[23] Jason Cong, Muhuan Huang, and Peng Zhang. “Combining computation and communication optimizations in system synthesis for streaming applications”. en. In: *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays - FPGA ’14*. Monterey, California, USA: ACM Press, 2014, pp. 213–222. DOI: 10.1145/2554688.2554771.

[24] P. Coussy et al. “An Introduction to High-Level Synthesis”. In: *IEEE Design & Test of Computers* 26.4 (July 2009), pp. 8–17. DOI: 10.1109/MDT.2009.69.

[25] M. Dahl et al. “Emulation of the Sparcle microprocessor with the MIT Virtual Wires emulation system”. In: *Proceedings of IEEE Workshop on FPGA's for Custom Computing Machines*. Napa Valley, CA, USA: IEEE Comput. Soc. Press, 1994, pp. 14–22. DOI: 10.1109/FPGA.1994.315594.

[26] Steve Dai et al. “Fast and Accurate Estimation of Quality of Results in High-Level Synthesis with Machine Learning”. In: *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Boulder, CO, USA: IEEE, Apr. 2018, pp. 129–132. DOI: 10.1109/FCCM.2018.00029.

[27] BARB DARROW. *Why Microsoft Is Putting These Chips at the Front and Center of Its Cloud*. en. Oct. 2016.

[28] D. Dingee, D. Nenni, and M.R. Chene. *Prototypical: The Emergence of Fpga-based Prototyping for Soc Design*. CreateSpace Independent Publishing Platform, 2016.

[29] Dong Liu and Benjamin Carrion Schafer. “Efficient and reliable High-Level Synthesis Design Space Explorer for FPGAs”. In: *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. Lausanne, Switzerland: IEEE, Aug. 2016, pp. 1–8. DOI: 10.1109/FPL.2016.7577370.

[30] Christophe Dubach, Timothy Jones, and Michael O'Boyle. “Microarchitectural Design Space Exploration Using an Architecture-Centric Approach”. In: *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. Chicago, IL, USA: IEEE, 2007, pp. 262–271. DOI: 10.1109/MICRO.2007.12.

[31] A. E. Eiben, P. -E. Raue, and Zs. Ruttkay. “Genetic algorithms with multi-parent recombination”. In: *Parallel Problem Solving from Nature - PPSN III*. Ed. by Gerhard Goos et al. Vol. 866. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 78–87. DOI: 10.1007/3-540-58484-6_252.

[32] Falcon. *Falcon Computing*. en-US.

[33] Fabrizio Ferrandi et al. “A Multi-objective Genetic Algorithm for Design Space Exploration in High-Level Synthesis”. In: *2008 IEEE Computer Society Annual Symposium on VLSI*. Montpellier, France: IEEE, 2008, pp. 417–422. DOI: 10.1109/ISVLSI.2008.73.

[34] *FPGAs and 3D ICs*.

[35] C.H. Gebotys. “Optimal scheduling and allocation of embedded VLSI chips”. In: *[1992] Proceedings 29th ACM/IEEE Design Automation Conference*. Anaheim, CA, USA: IEEE Comput. Soc. Press, 1992, pp. 116–119. DOI: 10.1109/DAC.1992.227851.

[36] Michael I. Gordon, William Thies, and Saman Amarasinghe. “Exploiting coarse-grained task, data, and pipeline parallelism in stream programs”. en. In: *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems - ASPLOS-XII*. San Jose, California, USA: ACM Press, 2006, p. 151. DOI: 10.1145/1168857.1168877.

[37] H. W. Gould. “The q -Stirling numbers of first and second kinds”. en. In: *Duke Mathematical Journal* 28.2 (June 1961), pp. 281–289. DOI: 10.1215/S0012-7094-61-02826-5.

[38] L.J. Hafer and A.C. Parker. “A Formal Method for the Specification, Analysis, and Design of Register-Transfer Level Digital Logic”. en. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 2.1 (Jan. 1983), pp. 4–18. DOI: 10.1109/TCAD.1983.1270016.

[39] Andrei Hagiescu et al. “A computing origami: folding streams in FPGAs”. en. In: *Proceedings of the 46th Annual Design Automation Conference on ZZZ - DAC ’09*. San Francisco, California: ACM Press, 2009, p. 282. DOI: 10.1145/1629911.1629987.

[40] S. Hauck and G. Borriello. “Pin assignment for multi-FPGA systems”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 16.9 (Sept. 1997), pp. 956–964. DOI: 10.1109/43.658564.

[41] J. Herath et al. “Dataflow computing models, languages, and machines for intelligence computations”. In: *IEEE Transactions on Software Engineering* 14.12 (Dec. 1988), pp. 1805–1828. DOI: 10.1109/32.9065.

[42] *High-Level Synthesis Compiler - Intel HLS Compiler*. en.

[43] M. Holzer, B. Knerr, and M. Rupp. “Design Space Exploration with Evolutionary Multi-Objective Optimisation”. In: *2007 International Symposium on Industrial Embedded Systems*. Costa da Caparica, Portugal: IEEE, July 2007, pp. 126–133. DOI: 10.1109/SIES.2007.4297326.

[44] Amir Hormati et al. “Optimus: efficient realization of streaming applications on FPGAs”. en. In: *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems - CASES ’08*. Atlanta, GA, USA: ACM Press, 2008, p. 41. DOI: 10.1145/1450095.1450105.

[45] Mohammad Hosseinabady and Jose Luis Nunez-Yanez. “Energy optimization of FPGA-based stream-oriented computing with power gating”. In: *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*. London, United Kingdom: IEEE, Sept. 2015, pp. 1–6. DOI: 10.1109/FPL.2015.7293946.

[46] T. C. Hu. “Parallel Sequencing and Assembly Line Problems”. en. In: *Operations Research* 9.6 (Dec. 1961), pp. 841–848. DOI: 10.1287/opre.9.6.841.

[47] Huynh Phung Huynh et al. “Mapping Streaming Applications onto GPU Systems”. In: *IEEE Transactions on Parallel and Distributed Systems* 25.9 (Sept. 2014), pp. 2374–2385. DOI: 10.1109/TPDS.2013.195.

[48] Huynh Phung Huynh et al. “Scalable framework for mapping streaming applications onto multi-GPU systems”. en. In: *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming - PPoPP ’12*. New Orleans, Louisiana, USA: ACM Press, 2012, p. 1. DOI: 10.1145/2145816.2145818.

[49] C.-T. Hwang, J.-H. Lee, and Y.-C. Hsu. “A formal approach to the scheduling problem in high level synthesis”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 10.4 (Apr. 1991), pp. 464–475. DOI: 10.1109/43.75629.

[50] Intel. *Cyclone V SoC FPGAs- Intel SoC FPGA*. en.

[51] Intel. *FPGA Architecture White Paper*.

[52] Intel. *Nios II Processors for FPGAs - Intel FPGA*. en.

[53] G.G. de Jong. “Data flow graphs: system specification with the most unrestricted semantics”. In: *Proceedings of the European Conference on Design Automation.* Amsterdam, Netherlands: IEEE Comput. Soc. Press, 1991, pp. 401–405. DOI: 10.1109/EDAC.1991.206434.

[54] Eunsuk Kang, Ethan Jackson, and Wolfram Schulte. “An Approach for Effective Design Space Exploration”. In: *Foundations of Computer Software. Modeling, Development, and Verification of Adaptive Systems*. Ed. by Radu Calinescu and Ethan Jackson. Vol. 6662. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 33–54. DOI: 10.1007/978-3-642-21292-5_3.

[55] V. Krishnan and S. Katkoori. “A genetic algorithm for the design space exploration of datapaths during high-level synthesis”. In: *IEEE Transactions on Evolutionary Computation* 10.3 (June 2006), pp. 213–229. DOI: 10.1109/TEVC.2005.860764.

[56] H. Krupnova. “Mapping multi-million gate SoCs on FPGAs: industrial methodology and experience”. In: *Proceedings Design, Automation and Test in Europe Conference and Exhibition*. Paris, France: IEEE Comput. Soc, 2004, pp. 1236–1241. DOI: 10.1109/DATE.2004.1269065.

[57] Manjunath Kudlur and Scott Mahlke. “Orchestrating the execution of stream programs on multicore platforms”. en. In: *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation - PLDI ’08*. Tucson, AZ, USA: ACM Press, 2008, p. 114. DOI: 10.1145/1375581.1375596.

[58] Kung. “Why systolic architectures?” In: *Computer* 15.1 (Jan. 1982), pp. 37–46. DOI: 10.1109/MC.1982.1653825.

[59] E.A. Lee and D.G. Messerschmitt. “Synchronous data flow”. In: *Proceedings of the IEEE* 75.9 (1987), pp. 1235–1245. DOI: 10.1109/PROC.1987.13876.

[60] D. Li et al. “Efficient design space exploration by knowledge transfer”. In: *2016 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. 2016, pp. 1–10.

[61] Joey Y. Lin, Ashok Jagannathan, and Jason Cong. “Placement-driven technology mapping for LUT-based FPGAs”. en. In: *Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays - FPGA ’03*. Monterey, California, USA: ACM Press, 2003, p. 121. DOI: 10.1145/611817.611836.

[62] Hung-Yi Liu and Luca P. Carloni. “On learning-based methods for design-space exploration with high-level synthesis”. en. In: *Proceedings of the 50th Annual Design Automation Conference on - DAC ’13*. Austin, Texas: ACM Press, 2013, p. 1. DOI: 10.1145/2463209.2488795.

[63] Shuangnan Liu, Francis Lau, and Benjamin Carrion Schafer. “Accelerating FPGA Prototyping through Predictive Model-Based HLS Design Space Exploration”. In: *Proceedings of the 56th Annual Design Automation Conference 2019 on - DAC ’19*. Las Vegas, Nevada, United States: ACM Press, June 2019, pp. 1–6.

[64] Gregory Lucas, Scott Cromar, and Deming Chen. “FastYield: Variation-aware, layout-driven simultaneous binding and module selection for performance yield optimization”. In: *2009 Asia and South Pacific Design Automation Conference*. Yokohama, Japan: IEEE, Jan. 2009, pp. 61–66. DOI: 10.1109/ASPDAC.2009.4796442.

[65] Yufei Ma et al. “Optimizing Loop Operation and Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks”. en. In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA ’17*. Monterey, California, USA: ACM Press, 2017, pp. 45–54. DOI: 10.1145/3020078.3021736.

[66] Anushree Mahapatra and Benjamin Carrion Schafer. “Machine-learning based simulated annealer method for high level synthesis design space exploration”. In: *Proceedings of the 2014 Electronic System Level Synthesis Conference (ESLsyn)*. San Francisco, CA, USA: IEEE, May 2014, pp. 1–6. DOI: 10.1109/ESLsyn.2014.6850383.

[67] Maxeler. *Apps | Maxeler AppGallery*.

[68] *Maxeler Technologies*.

[69] Tilo Meister, Jens Lienig, and Gisbert Thomke. “Novel Pin Assignment Algorithms for Components with Very High Pin Counts”. In: *2008 Design, Automation and Test in Europe*. Munich, Germany: IEEE, Mar. 2008, pp. 837–842. DOI: 10.1109/DATE.2008.4484778.

[70] Pingfan Meng et al. “Adaptive Threshold Non-Pareto Elimination: Re-thinking Machine Learning for System Level Design Space Exploration on FPGAs”. en. In: *Proceedings of the 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. Research Publishing Services, 2016, pp. 918–923. DOI: 10.3850/9783981537079_0350.

[71] Mentor. *Catapult High-Level Synthesis*. en.

[72] Mentor. *NVIDIA - HLS*. en.

[73] Mentor. *STMicroelectronics*. en.

[74] Ashish Mishra, A. R. Asati, and Kota Solomon Raju. “Scheduling of dataflow graphs on partial reconfigurable hardware in Xilinx PR flow”. In: *2013 International Conference on Advanced Electronic Systems (ICAES)*. PILANI, India: IEEE, Sept. 2013, pp. 108–112. DOI: 10.1109/ICAES.2013.6659371.

[75] Gordon E. Moore. “Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff.” In: *IEEE Solid-State Circuits Society Newsletter* 11.3 (Sept. 2006), pp. 33–35. DOI: 10.1109/N-SSC.2006.4785860.

[76] John Morris. *Intel pushes FPGAs into the data center*. en. Oct. 2017.

[77] Steven S. Muchnick. *Advanced compiler design and implementation*. San Francisco, Calif: Morgan Kaufmann Publishers, 1997.

[78] NEC. *CyberWorkBench: Products | NEC*.

[79] George Nemhauser and Laurence Wolsey. *Integer and Combinatorial Optimization: Nemhauser/Integer and Combinatorial Optimization*. en. Hoboken, NJ, USA: John Wiley & Sons, Inc., June 1988. DOI: 10.1002/9781118627372.

[80] J. von Neumann. “First draft of a report on the EDVAC”. In: *IEEE Annals of the History of Computing* 15.4 (1993), pp. 27–75. DOI: 10.1109/85.238389.

[81] *Next Generation 10 nm FPGAs - Falcon Mesa - Intel FPGA*. en.

[82] Ho-Cheung Ng, Cheng Liu, and Hayden Kwok-Hay So. “A Soft Processor Overlay with Tightly-coupled FPGA Accelerator”. In: *CoRR* abs/1606.06483 (2016).

[83] Dong Nguyen and Jongeun Lee. “Communication-aware mapping of stream graphs for multi-GPU platforms”. en. In: *Proceedings of the 2016 International Symposium on Code Generation and Optimization - CGO 2016*. Barcelona, Spain: ACM Press, 2016, pp. 94–104. DOI: 10.1145/2854038.2854055.

[84] Alexandru Nicolau and Roni Potasmann. “Incremental tree height reduction for high level synthesis”. en. In: *Proceedings of the 28th conference on ACM/IEEE design automation conference - DAC ’91*. San Francisco, California, United States: ACM Press, 1991, pp. 770–774. DOI: 10.1145/127601.127767.

[85] E. Nurvitadhi et al. “Accelerating Binarized Neural Networks: Comparison of FPGA, CPU, GPU, and ASIC”. In: *2016 International Conference on Field-Programmable Technology (FPT)*. Dec. 2016, pp. 77–84. DOI: 10.1109/FPT.2016.7929192.

[86] E. Nurvitadhi et al. “Accelerating recurrent neural networks in analytics servers: Comparison of FPGA, CPU, GPU, and ASIC”. In: *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. Aug. 2016, pp. 1–4. DOI: 10.1109/FPL.2016.7577314.

[87] Kenneth O’Neal et al. “HALWPE: Hardware-Assisted Light Weight Performance Estimation for GPUs”. en. In: *Proceedings of the 54th Annual Design Automation Conference 2017 on - DAC ’17*. Austin, TX, USA: ACM Press, 2017, pp. 1–6. DOI: 10.1145/3061639.3062257.

[88] Berkin Ozisikyilmaz, Gokhan Memik, and Alok Choudhary. “Efficient system design space exploration using machine learning techniques”. en. In: *Proceedings of the 45th annual conference on Design automation - DAC ’08*. Anaheim, California: ACM Press, 2008, p. 966. DOI: 10.1145/1391469.1391712.

[89] B.M. Pangrle and D.D. Gajski. “Design Tools for Intelligent Silicon Compilation”. en. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 6.6 (Nov. 1987), pp. 1098–1112. DOI: 10.1109/TCAD.1987.1270350.

[90] P. G. Paulin and J. P. Knight. “Scheduling and binding algorithms for high-level synthesis”. en. In: *Proceedings of the 1989 26th ACM/IEEE conference on Design automation conference - DAC ’89*. Las Vegas, Nevada, United States: ACM Press, 1989, pp. 1–6. DOI: 10.1145/74382.74383.

[91] P.G. Paulin and J.P. Knight. “Algorithms for high-level synthesis”. In: *IEEE Design & Test of Computers* 6.6 (Dec. 1989), pp. 18–31. DOI: 10.1109/54.41671.

[92] M. Pedram, K. Chaudhary, and E.S. Kuh. “I/O pad assignment based on the circuit structure”. In: *[1991 Proceedings] IEEE International Conference on Computer Design: VLSI in Computers and Processors*. Cambridge, MA, USA: IEEE Comput. Soc. Press, 1991, pp. 314–318. DOI: 10.1109/ICCD.1991.139906.

[93] M. Pedram, M. Marek-Sadowska, and E.S. Kuh. “Floorplanning with pin assignment”. In: *1990 IEEE International Conference on Computer-Aided Design. Digest of Technical Papers*. Santa Clara, CA, USA: IEEE Comput. Soc. Press, 1990, pp. 98–101. DOI: 10.1109/ICCAD.1990.129851.

[94] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[95] P. Prabhakaran and P. Banerjee. “Simultaneous scheduling, binding and floorplanning in high-level synthesis”. In: *Proceedings Eleventh International Conference on VLSI Design*. Chennai, India: IEEE Comput. Soc, 1998, pp. 428–434. DOI: 10.1109/ICVD.1998.646645.

[96] K.M.G. Purna and D. Bhatia. “Temporal partitioning and scheduling data flow graphs for reconfigurable computers”. In: *IEEE Transactions on Computers* 48.6 (June 1999), pp. 579–590. DOI: 10.1109/12.773795.

[97] Jacques Pyrat. *A.L.S.E the FPGA Experts*. en.

[98] Renqiu Huang and R. Vemuri. “Forward-looking macro generation and relational placement during high level synthesis to FPGAs”. In: *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.* Santa Fe, NM, USA: IEEE, 2004, pp. 139–144. DOI: 10.1109/IPDPS.2004.1303114.

[99] Nery Riquelme, Christian Von Lucken, and Benjamin Baran. “Performance metrics in multi-objective optimization”. In: *2015 Latin American Computing Conference (CLEI)*. Arequipa, Peru: IEEE, Oct. 2015, pp. 1–11. DOI: 10.1109/CLEI.2015.7360024.

[100] R. A. Rutenbar. “(When) will FPGAs kill ASICs?” In: *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*. June 2001, pp. 321–322. DOI: 10.1109/DAC.2001.156159.

[101] Benjamin Carrion Schafer and Anushree Mahapatra. “S2CBench: Synthesizable SystemC Benchmark Suite for High-Level Synthesis”. In: *IEEE Embedded Systems Letters* 6.3 (Sept. 2014), pp. 53–56. DOI: 10.1109/LES.2014.2320556.

[102] Benjamin Carrion Schafer and Kazutoshi Wakabayashi. “Divide and conquer high-level synthesis design space exploration”. en. In: *ACM Transactions on Design Automation of Electronic Systems* 17.3 (June 2012), pp. 1–19. DOI: 10.1145/2209291.2209302.

[103] Mike Sheng and Jonathan Rose. “Mixing buffers and pass transistors in FPGA routing architectures”. en. In: *Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays - FPGA ’01*. Monterey, California, United States: ACM Press, 2001, pp. 75–84. DOI: 10.1145/360276.360302.

[104] Amit Kumar Singh, Anup Das, and Akash Kumar. “Energy optimization by exploiting execution slacks in streaming applications on multiprocessor systems”. en. In: *Proceedings of the 50th Annual Design Automation Conference on - DAC ’13*. Austin, Texas: ACM Press, 2013, p. 1. DOI: 10.1145/2463209.2488875.

[105] Sharad Sinha and Thambipillai Srikanthan. “Dataflow graph partitioning for high level synthesis”. In: *22nd International Conference on Field Programmable Logic and Applications (FPL)*. Oslo, Norway: IEEE, Aug. 2012, pp. 503–506. DOI: 10.1109/FPL.2012.6339265.

[106] S. Summers, A. Rose, and P. Sanders. “Using MaxCompiler for the high level synthesis of trigger algorithms”. In: *Journal of Instrumentation* 12.02 (Feb. 2017), pp. C02015–C02015. DOI: 10.1088/1748-0221/12/02/C02015.

[107] Qingshan Tang, Habib Mehrez, and Matthieu Tuna. “Multi-FPGA prototyping board issue: the FPGA I/O bottleneck”. In: *2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*. Agios Konstantinos, Samos, Greece: IEEE, July 2014, pp. 207–214. DOI: 10.1109/SAMOS.2014.6893213.

[108] PandA Team. *Bambu | panda.dei.polimi.it*. en-US.

[109] Russell Tessier et al. “The virtual wires emulation system: A gate-efficient ASIC prototyping environment”. In: *2nd International ACM/SIGDA Workshop on Field-Programmable Gate Arrays*. Citeseer. 1994.

[110] *The FPGA Migration Is On*. Apr. 2000.

[111] Nils Voss et al. “Convolutional Neural Networks on Dataflow Engines”. In: *2017 IEEE International Conference on Computer Design (ICCD)*. Boston, MA: IEEE, Nov. 2017, pp. 435–438. DOI: 10.1109/ICCD.2017.77.

[112] R.A. Walker and S. Chaudhuri. “Introduction to the scheduling problem”. In: *IEEE Design & Test of Computers* 12.2 (1995), pp. 60–69. DOI: 10.1109/54.386007.

[113] Li-C. Wang and Malgorzata Marek-Sadowska. “Machine Learning in Simulation-Based Analysis”. en. In: *Proceedings of the 2015 Symposium on International Symposium on Physical Design - ISPD ’15*. Monterey, California, USA: ACM Press, 2015, pp. 57–64. DOI: 10.1145/2717764.2717786.

[114] Xuechao Wei et al. “Automated Systolic Array Architecture Synthesis for High Throughput CNN Inference on FPGAs”. en. In: *Proceedings of the 54th Annual Design Automation Conference 2017 on - DAC ’17*. Austin, TX, USA: ACM Press, 2017, pp. 1–6. DOI: 10.1145/3061639.3062207.

[115] Xuechao Wei et al. “Throughput optimization for streaming applications on CPU-FPGA heterogeneous systems”. In: *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*. Chiba, Japan: IEEE, Jan. 2017, pp. 488–493. DOI: 10.1109/ASPDAC.2017.7858370.

[116] I. H. Witten and Eibe Frank. *Data mining: practical machine learning tools and techniques*. 2nd ed. Morgan Kaufmann series in data management systems. Amsterdam ; Boston, MA: Morgan Kaufman, 2005.

[117] Xilinx. *MicroBlaze Soft Processor Core*.

[118] Xilinx. *SoCs, MPSoCs and RFSoCs*.

[119] Xilinx. *Vivado High-Level Synthesis*.

[120] Min Xu and Fadi J. Kurdahi. “Layout-driven RTL binding techniques for high-level synthesis using accurate estimators”. In: *ACM Transactions on Design Automation of Electronic Systems* 2.4 (Oct. 1997), pp. 312–343. DOI: 10.1145/268424.268425.

[121] Sotirios Xydis et al. “Efficient High Level Synthesis Exploration Methodology Combining Exhaustive and Gradient-Based Pruned Searching”. In: *2010 IEEE Computer Society Annual Symposium on VLSI*. Lixouri, Greece: IEEE, July 2010, pp. 104–109. DOI: 10.1109/ISVLSI.2010.56.

[122] Kai Yu, Jinbo Bi, and Volker Tresp. “Active learning via transductive experimental design”. en. In: *Proceedings of the 23rd international conference on Machine learning - ICML ’06*. Pittsburgh, Pennsylvania: ACM Press, 2006, pp. 1081–1088. DOI: 10.1145/1143844.1143980.

[123] Yung-Ming Fang and D.F. Wong. “Simultaneous Functional-unit Binding And Floorplanning”. In: *IEEE/ACM International Conference on Computer-Aided Design*. San Jose, CA: IEEE, 1984, pp. 317–321. DOI: 10.1109/ICCAD.1994.629787.

[124] J. Zhao et al. “COMBA: A comprehensive model-based analysis framework for high level synthesis of real applications”. In: *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. Nov. 2017, pp. 430–437. DOI: 10.1109/ICCAD.2017.8203809.

[125] Hongbin Zheng et al. “Fast and effective placement and routing directed high-level synthesis for FPGAs”. en. In: *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays - FPGA ’14*. Monterey, California, USA: ACM Press, 2014, pp. 1–10. DOI: 10.1145/2554688.2554775.

[126] Xinnian Zheng, Lizy K. John, and Andreas Gerstlauer. “LACross: Learning-Based Analytical Cross-Platform Performance and Power Prediction”. en. In: *International Journal of Parallel Programming* 45.6 (Dec. 2017), pp. 1488–1514. DOI: 10.1007/s10766-017-0487-0.

[127] Guanwen Zhong et al. “Lin-analyzer: a high-level performance analysis tool for FPGA-based accelerators”. en. In: *Proceedings of the 53rd Annual Design Automation Conference on - DAC ’16*. Austin, Texas: ACM Press, 2016, pp. 1–6. DOI: 10.1145/2897937.2898040.