ON ABSTRACTION METHODOLOGIES FOR RTL-BASED VLSI DESIGNS TO MAXIMIZE HIGH-LEVEL SYNTHESIS DESIGN SPACE EXPLORATION AND APPLICATIONS

ANUSHREE MAHAPATRA

PhD

The Hong Kong Polytechnic University
2018

The Hong Kong Polytechnic University
Department of Electronic and Information Engineering

On Abstraction Methodologies for RTL-based VLSI Designs to Maximize High-level Synthesis Design Space Exploration and Applications

Anushree MAHAPATRA

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

March, 2018

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 acknowledgement has been made in the text.

____________________________ (Signed)

Anushree MAHAPATRA (Name of Student)

Abstract

Moore's law has driven the complexity of Integrated Circuits (ICs) to unmanageable levels. To address this issue, extensive research is being done to develop new methodologies that can enable the design and verification of these complex ICs. In addition, current consumer trends are forcing IC design companies to continuously reduce the time-to-market while at the same time, the time-in-the-market of their products is shrinking, thus increasing significantly the risk of not obtaining the return-on-investment (ROI) targeted.

One of the main design methodologies that helps addressing these issues is to re-use multiple components between different designs, as well as using third party intellectual properties (3PIPs). In addition, companies have started raising the level of VLSI design abstraction. From low-level Hardware Description Languages (HDLs), *.e.g.* Verilog and VHDL, to high-level languages (HLLs) that were originally designed for software (SW) development. High-Level Synthesis (HLS) enables the synthesis of these HLLs into efficient hardware that implements their behavior. HLS has multiple advantages over traditional HDL-based VLSI design. One of the advantages that this thesis studies extensively, is the ability to generate micro-architectures of unique characteristics (*i.e.* area, power, performance), without the need to modify the behavioral description. This is typically called Design Space Exploration (DSE).

In particular, since most companies have large amounts of legacy Register Transfer Level (RTL) code, this thesis investigates automatic methods to convert these HDLs into HLL optimized for HLS, and in particular optimized for HLS DSE.

The contributions of this thesis are multi-fold: First, this work proposes a robust translation framework which identifies patterns in RTL code (VHDL or Verilog) that translate into high-level constructs that can in turn be explored such that different unique micro-architectures are generated. These constructs mainly include loops, arrays and functions. Second, the work introduces an improved DSE system using a hybrid synthesis based predictive method. Third, the RTL abstraction framework is applied to accelerate cycle-accurate system-level simulations by generating fast behavioral templates. Finally, an open source synthesizable SystemC benchmark suite was released to study the effectiveness of the proposed methodology. Moreover, in three years the benchmark has grown from 13 designs to 18 and has reached over 100+ downloads.

Publications

Journal Articles

1. **Anushree Mahapatra** and Benjamin Schafer, Optimizing RTL to C Abstraction Methodology to improve HLS Design Space Exploration, *IEEE Embedded System Letters*, (under review), pp, 1-4, 2018.

2. **Anushree Mahapatra**, Yidi Liu and Benjamin Schafer, Accelerating Cycle-accurate System-Level Simulations through Behavioral Templates, *Integration, The VLSI journal (Elsevier)*, May, 2018 (accepted for publication).

3. **Anushree Mahapatra** and Benjamin Schafer, VeriIntel2C: Abstracting RTL to C to maximize High Level Synthesis Design Space Exploration, *Integration, The VLSI journal (Elsevier)*, May, 2018 (accepted for publication).

4. Benjamin Schafer and **Anushree Mahapatra**, S2CBench: Synthesizable SystemC benchmark suite for High-level Synthesis, published in *IEEE Embedded System Letters*, Vol.5(3), pp. 53-56, April 2014.

Conference Papers

1. **Anushree Mahapatra** and Benjamin Carrion Schafer, Machine-learning based Simulated Annealer method for High Level Synthesis Design Space Exploration,

in Electronic System Level Synthesis Conference (ESLsyn), Proceedings of the 2014, pp.1-6, San Francisco, CA, 2014.

Presentations

1. Benjamin Schafer and **Anushree Mahapatra**, S2CBench: Synthesizable SystemC benchmark suite for High-level Synthesis, presented in 20th meeting of the North American SystemC Users Group,2014.

Acknowledgements

First and foremost, I would like to thank my supervisor, Professor Francis Lau, for his immensely helpful support throughout the PhD tenure. His thorough advice on journal peer reviews and intensive discussions helped me to strengthen my research contributions. Moreover, his encouragement towards different career paths have enabled me to strive for higher goals and become a better researcher.

I would also like to acknowledge and thank my co-supervisor, Dr. Benjamin Carrion Schafer who was also my former supervisor. He has been quite instrumental in shaping my thesis. I would like to extend my sincere gratitude to him for everything.

I also wish to thank my colleagues, Nandeesh, Anjana, Yidi for their constant emotional support through my arduous time in PhD. I also wish to thank my close friends, Jiwei, Kimberly, Cigdem, who were always with me to support me emotionally and help me brainstorm my research solutions. Last but not the least is the unconditional love, support and sacrifice of my parents and my brother, who have always put my happiness above theirs. Their constant love and support have enabled me to endure and shine in my PhD studies. I will always be indebted for their love and guidance.

Hong Kong Anushree Mahapatra

October 3, 2018

List of Abbreviations

AHB	Advanced High-performance Bus
ALAP	As Late As Possible
ALU	Arithmetic Logical Unit
AMBA	Advanced Micro-controller Bus Architecture
API	Application Program Interface
ASAP	As Soon As Possible
ASIC	Application Specific Integrated Circuit
CDFG	Control Data Flow Graph
CPU	Central Processing Unit
CWB	CyberWorkBench
DFG	Data Flow Graph
DMA	Direct Memory Access
DSE	Design Space Exploration
DSP	Digital Signal Processing
EDA	Electronic Design Automation
ESL	Electronic System Level
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
FU	Functional Unit
GPU	Graphics Processing Unit
HW	Hardware

HWAcc	Hardware Accelerator
HDL	Hardware Description Language
HLS	High Level Synthesis
IC	Integrated Circuit
IP	Intellectual Property
IR	Intermediate Representation
MPSoC	Multiple Processor System-on-Chip
PN	Petri Net
QoR	Quality of Results
RAM	Random Access Memory
ROM	Read-Only Memory
RTL	Register Transfer Level
SoC	System-on-Chip
SW	Software
VHDL	Very High Speed IC Hardware Description Language
VLSI	Very Large Scale Integration

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The research problem is summarized, along with the motivation for the research problem. Moreover, a thorough background is presented to give a clear view to the reader about the current problems in the VLSI industry.

1.1 Background

Very Large Scale Integration (VLSI) circuits are reaching complexities never seen before. Most circuits are now heterogeneous Multiprocessor System-on-Chips (MP-SoCs), which typically include embedded micro-processors, memory controllers, memories and dedicated hardware accelerators (HWAccs), all interconnected through a single bus or bus-hierarchies.

They are faster and consume less power than general purpose solutions and hence are well suited for typical consumer products' applications. The main problem while designing these heterogenous SoCs (System-on-Chip) is their increasing design complexity, which leads to long development times. With the increasing demand for newer products with more and more powerful features, new or reviewed versions of the SoCs need to be taped-out at continuously shorter time frames. This means that

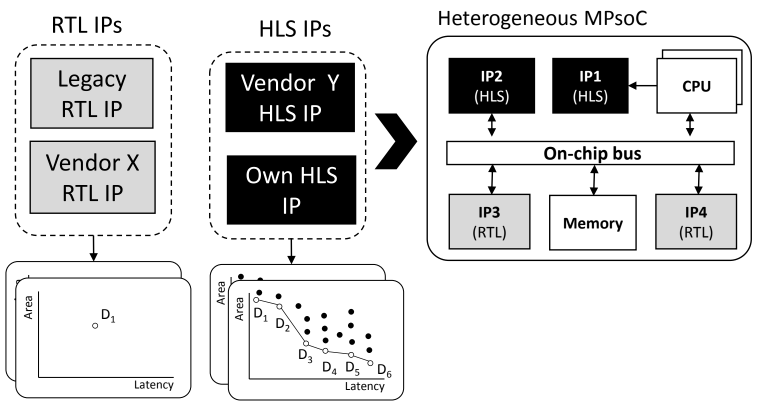


Fig. 1.1: Heterogenous MPSoC overview

the time-to-market for a SoC is getting shorter. However, the time-in-market also gets shorter and hence the risk of not achieving the estimated return on investment (ROI) grows significantly.

Figure 1.1 shows an example of a typical heterogenous MPSoC composed of multiple CPUs, on-chip memory and four different dedicated hardware (HW) modules. These are either specified in Register Transfer level (RTL) (e.g. VHDL or Verilog) or as behavioral untimed descriptions for HLS (e.g. ANSI-C, C++ or SystemC). These dedicated HW blocks are either developed in-house or sourced from third party vendors.

HW accelerators can be typically classified as either loosely coupled accelerators or tightly coupled accelerators. The tightly coupled are those that are directly controlled by a single master *i.e.* CPU, whereas loosely coupled HW accelerators can be accessed by any master in the system over the shared bus or Network-on-Chip (NoC). In figure 1.1, IP1 is tightly coupled with CPU whereas the rest of the accelerators are loosely coupled via the on-chip bus.

These accelerators are typically available as IPs in the form of legacy designs from other teams within the same company or third-party IPs from external vendors as shown in figure 1.1. In addition, these can be written in a HLS based behavioral-level language or in RTL. In the latter case, the system designer has very limited design options when re-targeting the IP for a new system, as each RTL IP has a fixed micro-architecture with unique area and latency. Thus for re-targeting, the designer needs to re-write the design manually each time to generate a different micro-architecture. This is time-consuming in order to explore alternative micro-architectures for a single RTL design. In contrast, the behavioral (HLS based) IPs can easily be explored automatically to create different micro-architectures tuned for every new system. Figure 1.1 shows this difference as the trade-off curves for RTL IPs and the HLS IPs. The RTL IP's micro-architecture has a fixed area and latency, while from the behavioral IP, a variety of micro-architectures can be obtained. Out of all the generated micro-architectures, the designer is typically only interested in the Pareto-optimal ones $(D_1, D_2, D_3, D_4, D_5, D_6)$. These pareto-optimal micro-architectures form a trade-off curve with unique area vs. latency (latency given in clock cycles). This is typically called Design Space Exploration (DSE). Moreover, IPs written in HLS based languages provide the advantage of easy integration of third-party vendor IPs (Vendor Y HLS IP in Figure 1.1) with existing in-house IPs. The reason is because the HLS languages provide higher levels of abstraction and with easier DSE, enable faster generation of HW architectures. In addition, raising the level of abstraction has another benefit of allowing the simulation of full systems orders of magnitude faster than at the RT-level (RTL). This thesis attempts to utilize these benefits to overcome the shortcomings of the traditional RTL design methodologies. Thus, the thesis proposes a method to convert RTL code (in particular Verilog) into optimized behavioral descriptions for HLS design space exploration.

1.2 Motivation

C-based design has multiple advantages compared to using low level HDLs. First, C-based designs have constructs which are actually *explorable* in nature. They may be in the form of loops, arrays, functions, etc. Secondly, one is able to automatically generate micro-architectures with unique trade-offs of design metrics (*e.g.* area vs. performance vs. power) without having to actually re-write the behavioral description. This method is termed as DSE. This is typically done using three main exploration *knobs*: (1) The synthesis directives in the form of pragmas can be used upon the *explorable* constructs that are extensively used by the commercial HLS tools. These directives then allow the control of the synthesis of these constructs, *e.g.* loops (unrolled, partially unrolled, not unrolled or pipelined), arrays (synthesized as RAMs or registers) and functions (inline or not). (2) The second exploration knob includes the number of functional units that the synthesizer can instantiate, while the third knob (3) include global synthesis options, *e.g.* target synthesis frequency and Finite State Machine (FSM) encoding.

Setting different knob values will lead to unique micro-architectures, which is not possible at the RT-level. Given a RTL IP, if converted to HLS based IP, it can then be used to perform DSE and maximize the design space of the RTL IP. Evidently, the use of exploration knobs on the *explorable* constructs in HLS based designs have the maximum impact in the quality of the DSE. The reason is that these constructs, if altered, widely vary the resulting micro-architecture thus giving rise to a larger design space of DSE. A pareto-optimal set is a set of dominating designs that cannot dominate each other for atleast one design objective. The designs in this set are the most promising for finding best suitable micro-architecture.

Thus, the main focus of this thesis is to investigate effective methods that can convert RTL descriptions given in synthesizable Verilog into HLS based descriptions consisting of *explorable* constructs. In other terms, the method abstracts the RTL designs to HLS based designs with *explorable* constructs. This implies generating loops, arrays and functions from a given RTL code (in Verilog) which, in turn, can be explored. From a single fixed RTL micro-architecture, the goal is to enhance its exploration, thereby generating a set of unique micro-architectures with trade-offs of different design metrics.

1.3 Challenges

Designs in RTL can be written in different ways as the program scope of RTL is very wide. Firstly, in order to develop methods for converting RTL descriptions, a robust method must be able to adapt to these diverse design styles written with different constructs available in the RTL scope. Secondly, the goal for a robust method is to be able to identify the functionality of the given RTL design and abstract the *explorable* constructs. At the same time, the framework must be able to adapt to different design styles of the RTL designs as input. These two constraints put together pose a harder problem to solve. Hence, these goals become a challenge while devising methodologies as one must make sure the quality of the exploration of the RTL design in consideration, must result in a maximized design space with a unique trade-off curve.

1.4 Scientific Contributions

Figure 1.2 pictorially describes the four main academic contributions in this thesis. The target heterogenous SoC architecture in the center is highlighted in gray. First, a set of synthesizable HLS designs complying with the synthesizable SystemC subset [80] was created (made open source). The benchmark suite termed as *S2CBench* served as the benchmarks for all our forthcoming experiments. The benchmark suite has been very well received with **100**+ downloads since its release. Next, we developed a novel HLS DSE exploration method using machine learning that has speed-up compared to existing DSE methods. The method follows a static-dynamic exploration approach. In our major contribution, we attempted to solve the objectives of this thesis by developing a method to convert synthesizable Verilog code into ANSI-C code optimized for DSE that we call *VeriIntel2C*. This method was refined by introducing further optimizations in our subsequent contribution. Finally, we use this method to speed up system-level cycle-accurate simulations.

In summary, the scientific contribution made in this thesis are:

- Introduce an open-source Synthesizable SystemC benchmark suite called *S2CBench*.

- Develop a runtime efficient machine learning based Design Space Explorer improving upon existing methods. This explorer serves to perform effective and scalable analysis on our proposed RTL to C translation as well as subsequent optimization methods.

- Propose and develop a graph-transformation based framework combined with network of rule-base search algorithms to abstract loops and arrays in RTL

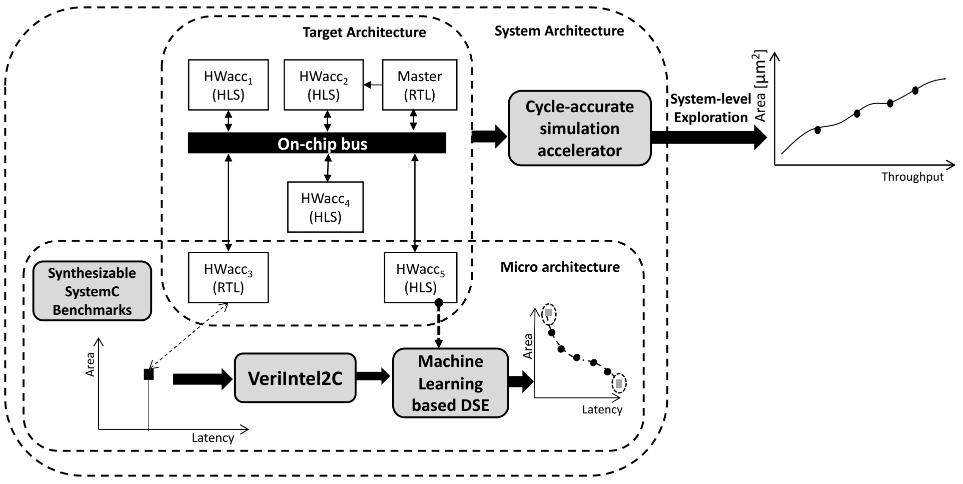


Fig. 1.2: Complete overview of our proposed system

descriptions and maximize the range of HLS DSE. This RTL to C translation framework is termed as *VeriIntel2C*.

- Design effective methods to extend the proposed RTL to C translation method by introducing novel optimizations in the translation method that further enhance the design search space, thus giving rise to new and *improved* alternative micro-architectures.

- Apply the *VeriIntel2C* framework to accelerate system level simulations by abstracting RTL components to fast behavioral templates.

1.5 Thesis Structure

The rest of the thesis is divided into several chapters which are as follows.

Chapter 2 introduces the fundamental theories behind High-Level Synthesis (HLS). Then Chapter 3 describes the design and implementation of HLS benchmark suite in SystemC. The benchmark suite is used for qualitative and quantitative analysis of *VeriIntel2C* as well as to evaluate the qualitative performance, efficiency of the proposed HLS DSE method. A brief introduction to C-based VLSI design methodology, HLS and their use and advantages is provided in Chapter 2. Next, the Chapter 4 describes DSE for HLS in detail, followed by our proposed improved method called Fast Simulated Annealing (FSA) using machine learning for efficient DSE. The Chapter 5 describes in detail, the automatic translation methodology called *VeriIntel2C* which abstracts the RTL designs to C/C++ descriptions (HLS). It consists of detailed explanation of the design of the graph-based framework as well as the decision algorithms. In Chapter 6, an optimization method is implemented upon the developed *VeriIntel2C* to enhance the quality, as well as further improve the design space of the exploration of RTL designs. The Chapter 7 describes an application use-case of our translation framework *VeriIntel2C*, where a template-based method is proposed to accelerate cycle-accurate system-level simulations of all combined HW accelerators. Finally, Chapter 8 discusses the conclusion and future work.

Chapter 2

High Level Synthesis

2.1 Overview

ITRS has suggested that by 2020, a 10x productivity increase is required in order to design complex SoCs [36]. Two main factors are mainly predicted which would enable to achieve this goal. One of the factors is the use of new VLSI design flow methodologies to raise the level of abstraction i.e. adoption of HLS. The second is the re-use of components, ITRS estimates that around 90% of the SoCs will be composed of re-used components. Design reuse is encouraged using languages of higher levels of abstraction. One of them is High-level Synthesis (HLS). HLS increases the level of abstraction in the VLSI design flow methodology.

2.2 Introduction to HLS

In earlier days, VLSI was based on a capture-and-simulate design methodology. The design process used to start with the product requirements without any information about the type of implementation, and block diagrams of the chip architecture would be built and then converted into circuit schematic. This circuit was in turn simulated to verify its functionality and timing. Over the past few decades, this process has shifted to a describe-and-synthesize methodology, where HW circuits are described using Hardware Description Languages (HDLs) like Verilog or VHDL. This improved methodology allows designers to describe the behavior of the HW with minimum or no implementation details. Logic synthesis tools then translate these HDLs into gate-netlists.

The use of higher-levels of abstraction have been proposed to further increase the design productivity [23]. These are based on synthesizing untimed behavioral descriptions, *e.g.* ANSI-C or C++ into Verilog or VHDL. This synthesis process is called High-Level Synthesis (HLS).

HLS provides several advantages compared to HLS over traditional RT-level.

2.3 HLS: Advantages

There are several points that highlight the advantages of HLS, and also motivates the work done in this thesis.

Advantage 1: Increase in Design Productivity: C-based VLSI design has many advantages compared to traditional RTL designs. Firstly, writing behavioral code is much faster and and easier than low level RTLs. It is estimated that a single line of C code generates $7\times$ more gates than RTL [20].

Advantage 2: Lowers barriers of entry to HW design : In addition, HLS takes as inputs high-level languages (HLLs) like C, C++ or Matlab to describe the behavior to be converted into HW. The advantage of using such HLLs are multi-fold: Firstly, the use of such commonly used languages cater to a larger base of designers with high-level programming knowledge. Secondly, different members in a design team require modifications on different aspects of a design. The HLS languages for this reason, have the advantages of readability, manageability and maintainability. The structure of these languages for HLS is very modular having separate constructs pertinent to every set of operations which enables the designer to modify design styles of individual constructs without affecting the overall structure of the design.

Advantage 3: Maximizes Resource Sharing: Moreover, HLS enables to maximize resource sharing by exploiting the resource allocation step of HLS design flow methodology in section 2.3.3. Hence, this allows the creation of much smaller designs, occupying lesser silicon area compared to hand-coded RTL designs [**?**]. Most HW designers avoid resource sharing because it is more difficult to debug these type of circuits and hence, choose to over-design their circuits to make them easier for verification.

Advantage 4: Design Space Exploration: In addition, one of the most important advantages of HLS is the ability of generating a set of different micro-architectures with different area vs. performance trade-offs without having to modify the original behavioral description also called Design Space Exploration (DSE). Using HLS DSE, the design team can easily create different configurations for the same description to meet demands of different design metrics. Trade-off curves are formed to compare the quality of the different micro-architectures created by implementing different constructs of the design in different forms in the hardware circuit. These modifications significantly impact the design objectives like area, latency, power, etc., thus, enameling the re-use of the behavioral code for different projects.

Advantage 5: Faster Verification: Finally, HLS allows the generation of fast simulation models at different abstraction levels ranging from transaction to cycle-accurate, which execute 10$\times$ to 1,000$\times$ faster than RTL simulations [20]. The main

advantage of using HLS is that it shortens verification cycles thus shortening the overall design cycles. Thus, this leads to adopting design reuse methodology which is an important factor to solve the challenges described in the earlier chapter. HLS allows faster cycle-accurate simulations which again are one-level down than the transaction-level simulation.
C-based VLSI design is finally being deployed extensively in industry for commercial designs. The increase in productivity combined with the improvement in the quality of the results of commercial HLS tools has convinced many design teams to make the transition towards HLS. This transition is nevertheless gradual and currently most of the applications being targeted through HLS are Digital Signal Processing (DSP). It has also been shown in the past that HLS can rival hand-coded RTL designs for these type of applications [75]. Now, we discuss the different steps involved in the HLS methodology that enable the synthesis of untimed behavioral descriptions into RT-level description (*e.g.* Verilog or VHDL).

Figure 2.1 highlights the main steps involved in HLS. HLS takes as input a behavioral description combined with a set of constraints and library of resources and synthesizes the description into a RTL description that can efficiently be executed. The resources constraints typically include the number and type of the functional units, like adders and/or multipliers or storage units like memories or register files. In addition, the global constraints include the target synthesis frequency, the FSM encoding scheme, etc.

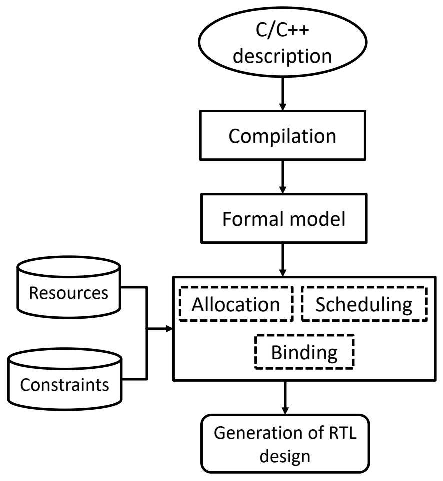


Fig. 2.1: HLS traditional flow

2.3.1 Front-End

The HLS process starts by compiling and parsing the behavioral description, performing syntactical checks and doing technology-independent optimizations like constant propagation and dead code elimination. This is often considered as the front-end of the HLS processor. It then continues to create a Control Data Flow Graph (CDFG) onto which the three main synthesis steps are executed. The CDFG captures both the control and data flow dependencies and are required by the subsequent operations to be performed upon them. The next three steps form the major part of the HLS process.

The synthesis process then continues with the three main stages in HLS. The three main stages, allocation, scheduling and binding, are described in detailed in the next subsections. The output of these stages are synthesizable RTL code which is typically composed of a controller in the form of a Finite State Machine (FSM) and a datapath.

2.3.2 Allocation

Allocation involves the selection of number and type of functional units (FU) onto which the different operation in the CDFG need to be mapped to. By default, the allocation stage tries to allocate as many functional units as possible to fully parallelize the CDFG leading to an implementation that maximizes the parallelism in the implementation. Allowing a single FU of each type will lead to a hardware circuit that can only execute one operation at a time, similar to a processor, and thus, making a dedicated hardware implementation not so useful.

2.3.3 Scheduling

Once the allocation stage has finished, the HLS process continues by scheduling the CDFG taking the resource constraints of the allocation stage into consideration. Scheduling sequences the operations in the CDFG and allocates them to individual control steps, where a control step can be defined as one clock cycle with delay=$1/f_{target}$. Moreover, independent operations can be scheduled at same clock cycles and executed in parallel to make the optimum use of resources and to reduce the latency. Also, operations can be directly connected another in the same control step to further reduce the latency if the clock period is large enough. This concept is called operation chaining.

It has been shown that resource constraint scheduling is an NP-hard problem [19], thus multiple heuristics have been proposed to solve the problem efficiently. Existing scheduling algorithms are broadly classified into two categories:

1. Data-flow (DF) based Scheduling

2. Control-flow (CF) based Scheduling

DF-based scheduling is more suited for data-flow intensive applications such as Digital Signal Processing or Image Processing. There are mainly two types of DF-based scheduling based on the optimization objectives, time-constrained and resource-constrained. Time and resource are two objectives that are in trade-off with each other. Force-directed scheduling [63] is a commonly used heuristic to solve time-constrained scheduling problem. Force-directed scheduling aims to reduce the resource usage by balancing the computations over the time steps. In contrast, list scheduling is used to solve the resource-constrained scheduling problems. In list scheduling, the available operations are sorted into a list according to a priority function and scheduled into the control states with available resources [62],[37]. For unconstrained scheduling problems, As Soon As Possible (ASAP) and As Late As Possible (ALAP) scheduling algorithms are the earliest and simple approaches. With ASAP scheduling, operations are scheduled in the earliest possible clock cycle whereas ALAP scheduling assigns operations to the latest possible clock cycle. The ALAP and ASAP schedules are often used in combination with other compute-intensive algorithms to identify the bounds of the particular problem.

In contrast, CF-based scheduling handles control intensive applications like controllers or network protocol processors. Path-based scheduling algorithms are the earliest approaches to solve CF-based scheduling using ASAP schedule [14]. SPARK [30] uses list scheduling algorithm to dynamically select and apply code transformations in a HLS framework. For time-constrained scheduling problems, Relative scheduling [40] is preliminarily used to solve maximum/minimum timing constraints. Ly et al. [46]

introduces behavioral templates by locking number of operations to certain scheduling templates. ILP based scheduling techniques are introduced in [35], [27], symbolic scheduling in [33], [84] and constraint-programming-based scheduling [41].

2.3.4 Binding

The final step in HLS is called binding. The process of binding involves the binding of the operations in the scheduled DFG to the available functional units capable of executing the operation. Every variable in the design should be bound to a specific storage unit and every operator in the design should be bound to the functional unit.

Binding techniques are often used in conjunction with the allocation stage. A vast amount of binding algorithms have been proposed in the past. In [61], the authors propose a novel technique to perform register binding by dividing the lifetime of variables into two intervals. Raj et al., [68] uses binding integrated with scheduling immediately after control steps are scheduled. Moreover, the HAL system [81] decides the number of registers for binding, by weighted clique partitioning method. The HYPER system [66] determines the binding after it allocates and binds the functional units and interconnections. In BUD/DAA system [51], the method is divided into two stages wherein one method determines the registers of each cluster and the DAA is a rule-based expert system to perform the mapping. The EMUCS system [81] uses a binding method where it assumes that the number and width of the registers are pre-determined either by the system or by users.

2.3.5 RTL Design Generation

After the execution of the three main stages (scheduling, allocation, binding,) the HLS process can finally output the RTL code in the form of synthesizable Verilog or

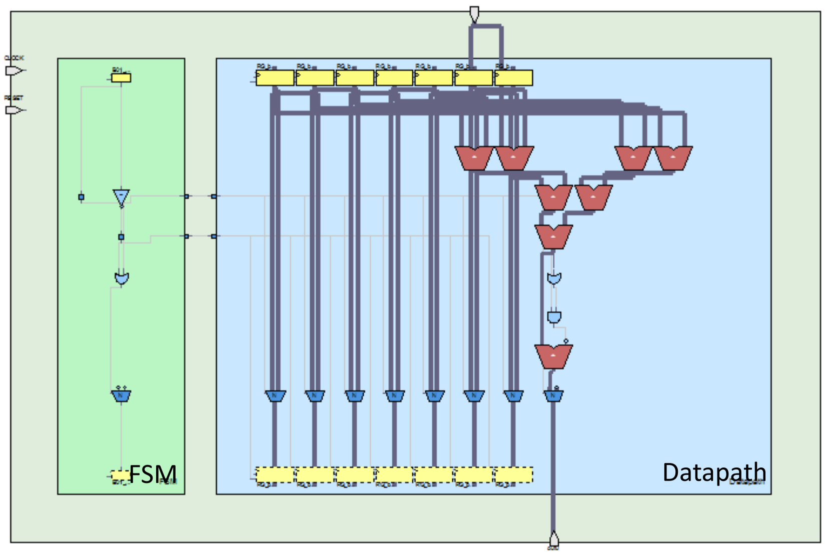
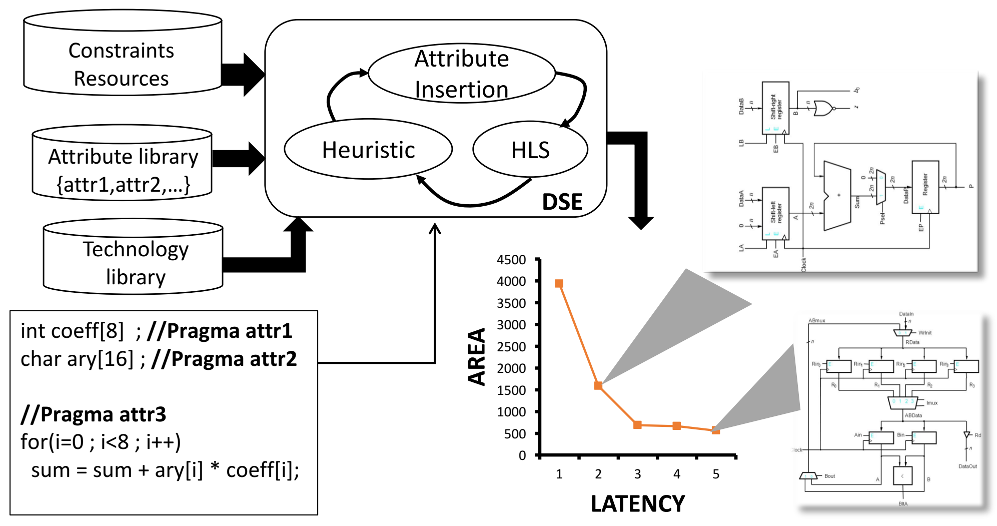


Fig. 2.2: Typical circuit generated after HLS including a Datapath and a controller (FSM)

VHDL. This is often considered as the back-end of the HLS process.

Figure 2.2 shows an example of a typical architecture generated from a commercial HLS tool [21]. The design usually consists of a data-path and a Finite State Machine (FSM) module which synchronizes the data through the data path. The FSM controls the flow of data is *circulated* in the data-path. This is important especially when FUs and registers are shared. In a FSM, every state is responsible for controlling a data-flow operation inside the datapath. States may have internal control logic to decide the sequential order of the states. While in data-path, the operations can be represented as multiplexers if there are not many resources available, or wires to perform combinational logic. Multiplexers need control signals that are generated from FSM. The reason is that the FSM sets the control signals for the multiplexers that control the order of data inputs to other blocks. The data-path consists of the flow of operations performing data manipulation, hence describing the behavior of



```
int coeff[8]  ; //Pragma attr1
char ary[16] ; //Pragma attr2

//Pragma attr3
for(i=0 ; i<8 ; i++)
  sum = sum + ary[i] * coeff[i];
```

Fig. 2.3: Basic working flow of a DSE system

2.3.6 HLS Design Space Exploration

A typical DSE is described in figure 2.3 using a simple working example of a design that computes the average of 8 numbers. The input C-based behavioral description are inserted with pragmas to control the design of the resulting micro-architecture. The constraints and resources refer to the number of hardware units and operators that are allowed and the total hardware units used for allocation to be used in the synthesis of the input description respectively. Attribute library is the database of all the possible pragmas whereas technology library contains the hardware components definitions in different technology standards. With these inputs, a typical DSE method uses several types of heuristic algorithms to explore the HLS description using combinations of different pragmas and generates a trade-off graph of different design points. Each design point corresponds to a unique micro-architecture circuit (logic circuit of data-path). Using DSE, designs using HLS can be enabled for re-use by choosing other micro-architectures. This approach will be discussed in detail along with the previous related work in chapter 4.

2.3.7 Conclusion

This chapter has presented the concept of HLS and its fundamental working methodology. We have described in detail, the three main internal steps that are (i) allocation (ii) scheduling and (iii) binding. Also, we highlighted some of its major advantages over traditional VLSI design methodologies based on low-level HDLs such as Verilog or VHDL. One particular advantage is the ability of generating multiple micro-architectures with different trade-offs for the same HLS based behavioral description. This thesis will exploit this particular advantage in the subsequent chapters.

Chapter 3

Synthesizable SystemC Benchmarks for High Level Synthesis

3.1 Overview

High-Level Synthesis(HLS) is being increasingly used for commercial VLSI designs. This has led to the proliferation of many HLS tools. In order to evaluate their performance and functionalities, a standard benchmark suite in a common language supported by all of them is required. Moreover, the benchmarks made for diverse applications, form a strong base for verifying the robustness of our proposed methodologies and contributions in the following chapters. This chapter introduces the concept and requirement of synthesizable benchmarks for HLS and describes the design and implementation of them as well. The benchmarks comply with the latest Synthesizable SystemC standard, called S2CBench: Synthesizable SystemC Benchmark. They have been carefully chosen to not only include applications of different sizes and from various domains typically used in HLS (e.g. encryption, image and DSP application), but also to test specific optimization techniques in each of them. This allows an easy comparison of not only Quality of Results (QoR) of the different HLS tools under review, but also to test their completeness.

3.2 Introduction

HLS has evolved significantly over the last decade and the QoR of commercial HLS tools has improved to the level where HLS has begun to be used for commercial designs. The adoption of HLS as part of standard VLSI design flows has led to the proliferation of HLS tools. The main problem faced by many designers, wanting to transition from traditional RTL to C-based design, is the absence of validated standards to evaluate the different HLS tools available (a good comparative review of these can be found at [52]). The evaluation phase is crucial in order to find the best HLS tool for the type of applications being designed. However, the lack of expertise in HLS combined with smaller time-to-market deadlines makes it hard to set up an efficient evaluation methodology. Moreover, HLS tools depend on a wide range of vendor-specific optimization features in order to get a good quality QoR. Thus, this makes it more difficult to master different tools. Finally, different tools support different input languages, further complicating the HLS tools evaluation process. This often results in the behavioral descriptions being sent to different HLS tool vendors and the designs would be manually optimized for that particular tool. Upon that, the designers eventually determine the appropriate tool by evaluating the QoR of the synthesized circuits.

Fortunately, there exists a common language supported by all of the main HLS tools: SystemC. There have been many discussions to find the best input language for HLS [25], but eventually SystemC has gained wider acceptance over ANSI-C and C++ . It is mainly due to the IEEE standardization efforts of OSCI (now Accellera). OSCI set up a working group to define a synthesizable SystemC subset currently supported by all of the HLS tools. Thus, in order to facilitate the evaluation of different commercial HLS tools, a synthesizable SystemC Benchmark suite is required. This benchmark suite should not only cover applications of different domains and designs of different sizes, but also include structures to test specific optimization techniques. The success of many HLS tool evaluations often depends on the expertise of the Field Application Engineer (FAE). The original behavioral descriptions often have to be manually modified by the FAE to obtain the best QoR. Re-writing of behavioral descriptions can sometimes be considerably time-consuming and should be taken into account while evaluating different HLS tools. By designing a benchmark suite that tests specific features, designers can easily understand the strengths and weaknesses of each tool. SystemC is most recognized to create TLM models of IPs [4]. Many HLS tools do not support ANSI-C. Using C benchmarks like Chstone etc, would require an extra effort for designers to convert benchmarks to SystemC. Moreover, converting to SystemC may reduce the capability of the original C benchmarks that they are designed for. There are no direct tools presently available that directly convert to SystemC. ANSI-C is inherantly sequential whereas SystemC is concurrent in nature. Moreover, the result SystemC design must be synthesizable as well. These factors pose a challenge for direct conversion from C to SystemC.

The main features to be tested may be classified in three main categories: (1) Language support (e.g. templates, structures, and fixed point data types) (2) synthesis optimizations (multi-dimensional array expansion, polynomial decompositions, functions synthesis, loop unrolling, pipelining and array synthesis) and (3) tool performance (e.g. synthesis running time, accuracy of area and timing report).

3.3 Related Work

Multiple efforts in the area of general-purpose computing benchmarks have been made since the 1980s with the SPEC benchmark suite [22] as one of the earliest. The SPEC benchmark mainly intends to analyze the performance of major system components. More recent benchmark suites specialize on specific domains. Amongst them, MediaBench [42] focuses on multimedia applications and MiBench [31] on modern embedded applications. These benchmarks are often used as HLS benchmarks because they represent computationally intensive applications amiable to HLS. However, these benchmark programs are not made synthesizable (by any HLS tool) and allow only a very limited number of HLS features to be tested.

In terms of HLS benchmarks, some of the initial efforts were the HLS92 and the HLS95 benchmarks [60]. Although many of the optimizations targeted in the aforementioned benchmarks are still relevant, they were written in behavioral VHDL, which is currently not supported by any commercial HLS tool. A more recent effort is the CHStone [32] benchmark suite. Similar to our work, it targets HLS and includes a set of ANSI-C programs. The main drawback of using ANSI-C is, as mentioned before, that some of the main commercial HLS tools do not support ANSI-C. As example, Forte Cynthesizer [24] only supports SystemC and Calypto's CatapultC [12] supports C++ and SystemC, but not ANSI-C. Other tools like NEC's CyberWorkBench [21], support SystemC and a subset of C called BDL (Behavioral Description Language). This tool forces descriptions given in C to be manually converted into this subset. Another important limitation of ANSI-C benchmarks for HLS is that ANSI-C does not support fixed point data types, which is extremely important in DSP applications, often targeted in HLS. There have also been recent works in benchmarks using HLS languages. One of them being Spector benchmark suite [26] that provides a benchmark suite for FPGA in openCL. Spector benchmarks aim to test the exploration knobs of FPGA HLS tools. However, Spector is written in openCL that restrict its use to FPGA only. Tools like Rosetta [88] provide benchmarks that can be targeted for FPGA HLS tools as well as the HLS research community. But Rosetta limits the application to FPGA, wherein the designs act as FPGA hardware accelerators in HLS.

In this chapter, we present S2CBench, a freely available synthesizable SystemC benchmark suite [76], initially consisting of 12+1 programs targeting a variety of applications typically used in HLS. This suite has been downloaded more than 100 times and is available open-source. Out of the original benchmarks, 12 complied with the latest SystemC synthesizable subset draft, while one design (FFT) is non-synthesizable because it contains trigonometric and floating point operations. This design was added in order to help users understand how the different HLS tools support these operations as most commercial tools have vendor specific ways to support them. One of the unique features of the benchmark suite is that every application is accompanied by its respective testbench. The testbench contains test vectors stored in a file and compares the simulation results with a golden output included for each design. The test vectors are not fixed and can be modified by the user. The option to create a waveform has also been made available. In this case, the simulation will produce a VCD file, which may be viewed by any waveform viewer. Finally, each benchmark is designed to test specific optimization options allowing designers, evaluating different HLS tools, to understand the support of the tools for these options.

The benchmarks are classified into different categories of applications. S2CBench is also used or highlighted in [44], [43], [6], [13].

3.4 Benchmarks Description

Synthesizable SystemC Benchmark (S2CBench) suite is a collection of 12+1 programs following the latest SystemC Synthesizable Subset draft 1.3. Some of the main objectives of S2CBench are:

- To enable the direct comparison of commercial HLS tools
- To test specific tool features classified as language support, synthesis optimization techniques and tool performance.
- To help researchers analyze and compare their own techniques.

3.4.1 Benchmark programs

Every benchmark program in S2CBench is designed to test particular features. Described below, is a brief overview of the programs included in the suite, each categorized according to its application domain. The benchmarks are also classified into data-dominant (dd) or control-dominant (cd) designs. HLS usually achieves very good results for the former category, whereas it sometimes creates sub-optimal designs for the latter category.

Automotive and Industrial

The category includes applications normally used in embedded control systems, which perform extensive basic math operations and bit manipulations.

qsort (dd): Quick sort design sorts data in ascending order using the well-known quick sort algorithm. Sorting of data is important for designs, so that they can be analyzed easier and priorities be established. This design helps in verifying the support for basic HLS optimization techniques which include loop unrolling, array synthesis (register or memory) and function synthesis with pointer argument support. It should be noted that various disciplines require fast sorting. Thus, qsort design could be categorized into many other categories.

A2 Security

The Security category includes several algorithms for data encryption and hashing. HLS has proved to be a very good solution for designing data security applications due to their mathematical complexity. Moreover, these types of applications are very difficult to verify in RTL [55].

aes_ cipher (dd): Advanced Encryption Standard (AES) Cipher encryption algorithm performs encryption/decryption. This program consists of many user-defined functions. It contains a large number of small *for* loops having inter-loop data dependencies. The main optimization techniques addressed are input port expansion, array synthesis (memory or registers), function synthesis (inline or goto operators) and large fixed arrays synthesized as logic or ROMs.

kasumi(dd): Kasumi is a block cipher algorithm used in mobile communication systems. The SystemC description includes two threads and multiple functions. Therefore, this design is useful to verify the synthesis and especially the verification of multi-process systems. The design also contains multi-dimensional I/O ports and multiple arrays. Finally, the kasumi algorithm, similar to most encryption applications, contains large amount of logic operations (e.g. and, or, xor). HLS tools are notably not efficient, for accurately estimating the critical path of these applications, because the discrete delay of all the operations are simply added, thus overestimating the critical path. This application can provide an indication of the accuracy of the HLS timing report, compared to that of the logic synthesis result.

md5C(dd): The Message Digest Algorithm is widely used in cryptography to generate hash functions and check data integrity. MD5C is a single process design consisting of multiple functions, arrays of different bit widths and different levels of loop nesting. One of the unique language constructs to be tested with this design is the extensive use of *define* macros.

snow3G (dd): Snow 3G is a stream cipher that produces a key stream that consists of 32-bit blocks using a 128-bit key. Apart from the main optimization options, this design tests the support of HLS tools for templates. A variable length multiplication operation is performed in this algorithm, which may be easily simplified using templates.

A3 Telecommunication With the explosion of portable electronic devices using wireless communication, constrained by limited power budgets, some of the telecommunication functions are frequently being implemented as custom HW blocks in SoCs (Systems on Chip). HLS is a natural choice for most of the complex applications in this domain, having well known legacy C descriptions.

adpcm (cd): Adaptive Differential Pulse-Code modulation (encoder part only) accepts 16-bit Pulse Code Modulation (PCM) samples as input and converts them into 4-bit samples. Some of the optimization techniques that can be tested with this design are loop unrolling, function synthesis, fixed array synthesis, the most important being the support for inclusion of structures. Some HLS tools do not support the use

Table 3.1: Benchmark domain and optimization summary

Design	Type	Domain	Optimization and language support
qsort	dd	Auto/Ind.	loops, arrays, functions, pointers
sobel	dd	Auto/Ind.	loops, functions, I/O array expansion, multi-dimensional arrays expansion, fixed arrays
AES cipher	dd	Security	I/O array expansion, multi-dimensional arrays expansion, large fixed arrays
Kasumi	dd	Security	multi-process, delay report accuracy
md5c	dd	Security	#define macros delay report accuracy
adpcm	cd	Telecom	structures
FFT	dd	Telecom	floating point, trigonometric functions
FIR filter	dd	Consumer	I/O array expansion, arrays, loops, functions, Sum of Products (SoP)
decimation	dd	Consumer	Resource sharing across loops, fixed point data type, SoP
Interpolation	dd	Consumer	Polynomial decompositions, fixed point data, SoP
IDCT	dd	Consumer	#include statement to initialize arrays
disparity	cd/dd	Consumer	hierarchical design, multi-dimensional array expansion, synthesis running time

of structures, forcing the designer to re-rewrite the original descriptions manually.

fft (dd): (Fast Fourier Transform): The fft algorithm is the only design in the benchmark suite that is not synthesizable since the design includes floating- point data and trigonometric operations, which are not synthesizable as per the latest synthesizable subset draft. However, the design has been included as part of the suite since most HLS vendors do support floating point and trigonometric operations, though the process of synthesizing is vendor specific. It is important to understand the level of support for these operations by the HLS tool.

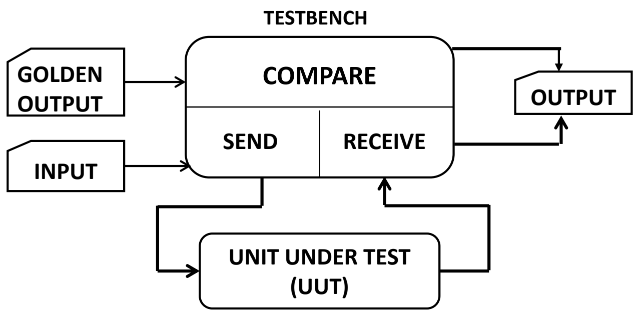


Fig. 3.1: Structure for testbench validation

A4 Consumer

Consumer benchmarks represent applications that are closely related to multimedia and digital signal processing (DSP) applications. The focus of this domain is primarily on filters as HLS has shown to produce very good results for applications involving filters and is widely used in this field.

fir (dd): The fir filter is a 10- tap FIR filter algorithm designed for 8- bit integer operations. The aim of this design is to check for loop unrolling, automatic array expansion of the I/O ports and accepting pointers to functions. This program can be pipelined as well.

decimation (dd): The algorithm is a 5-stage decimation filter. It consists of 5 FIR filters cascaded together where the output of one stage is the input to the next stage. The main purpose of the design is to evaluate the level of resource sharing that the HLS tool can extract by sharing the Multiply Accumulate (MAC) operations of the filtering function across multiple loops. The secondary purpose is to verify that the generated RTL is able to preserve the sum of product (SoP) construct provided in the SystemC code, so that the logic synthesis tool can optimize the construct further. Finally, this design also serves to identify if the HLS tool supports fixed- point data types and its different rounding and saturation modes.

interpolation (dd): The algorithm is a 4-stage interpolation filter. Apart from the above mentioned optimization techniques like loop unrolling and arrays synthesis, the main purpose of this design is to test if the HLS tool can perform automatic polynomial decompositions. It is because significant area reduction can be obtained if polynomials can be decomposed into terms, so that the total number of arithmetic operations required is reduced. Similar to the decimation filter, this design also makes extensive use of fixed point data types.

Sobel (dd): The sobel filter is an edge-detection algorithm that takes a bitmap image directly as the input and returns a new bitmap image solely consisting of the edges of the original image. The program specifically checks for nested-loop unrolling and pipelining optimizations, I/O ports expansion (expanded inputs specified as arrays to individual ports), multi-dimensional arrays expansion, fixed arrays synthesized as logic or ROMs and pointer arguments to functions.

idct (dd): The Inverse Discrete Cosine Transform expresses a finite sequence of data points in terms of, a sum of cosine functions of different frequencies. It is normally used in many applications, e.g. image compression (jpeg) and solution of partial differential equations. The synthesizable SystemC version of this algorithm included in S2CBench serves additionally as a unique language support feature, to test the initialization of an array using *include* statement.

disparity (cd/dd): This program estimates the disparity in a stereoscopic image. It is the largest of all the designs and consists of 4 processes executed in parallel (sc_cthreads). The design can serve to compare the synthesis running times of the different tools, since the main thread contains a large number of loops leading to extreme long synthesis run times, in case of the loops being fully unrolled or pipelined. This design allows the testing of most of the optimization techniques of the designs described previously as well as the ability of the HLS tools to deal with hierarchical designs and their respective synthesis running times. The design contains control as well as data dominant parts.

Table 3.1 shows the summary of all the designs included in the S2CBench benchmark suite and the associated targeted optimization features. Most of the designs

Table 3.2: Benchmark program characteristics

Program characteristics	qsort	sobel	aes cipher	kasumi	md5c	snow 3G	adpcm	fft	fir	decim	interp	idct	disparity
Lines of Code	204	269	429	415	467	522	270	334	176	422	231	450	634
Processes	1	1	1	2	1	1	1	1	1	1	1	2	4
Function	1	2	11	5	7	11	3	1	2	1	1	2	4
No. of arrays	2	3	7	13	5	5	1	2	2	10	5	2	6
if statement	1	8	3	2	3	1	12	0	1	19	0	2	16
for statement	5	8	20	12	8	4	1	2	2	15	5	3	11
while statement	1	1	1	2	2	2	1	10	1	1	1	1	9
Test vector	.txt	.bmp	.txt	.txt	.txt	.txt	.txt	.txt	.txt	.txt	.txt	.txt	.bmp
Operations													
add/sub	8	26	65	44	284	11	15	17	7	50	14	123	33
Multiplications	0	2	16	0	4	0	2	5	1	5	10	33	2
Divisions	0	0	3	0	0	0	0	2	0	0	0	0	1
Logic ops	7	0	22	39	274	67	9	0	5	4	2	33	13
Comparisons	0	17	29	22	16	10	16	10	0	43	8	36	42

include loops, arrays and functions and hence are omitted in designs having unique features.

3.4.2 Benchmark Characteristics

As discussed in the previous section, the designs range from smaller single process designs (e.g. quick sort or FIR filter) to larger multi-process designs (e.g. kasumi and disparity). Table 3.2 shows the detailed composition of the designs. The table is divided into two categories; the first category describes the number of program lines (not including comments, blank lines or the testbench), number of processes, functions, loops and arrays while the second category describes the variety of operations in the code.

From this table, it may be observed that the security applications contain a large number of logic operations and comparisons, while the DSP applications require many adders and multipliers mainly to compute the MAC operations of the filtering stage. All designs included in this benchmark suite are directly synthesizable without any modifications except for the FFT which is non-synthesizable.

3.5 Benchmark Validation

A SystemC test-bench is provided with all of the designs in order to verify their functionality. Figure 3.1 shows the modular structure of the test-bench interface with the synthesizable design. The test-bench module contains a *send* and a *receive* process which are modeled as clocked threads in SystemC. The *send* process transmits data to the Unit Under Test (UUT) continuously until the test vectors stored at the input file are exhausted and the *receive* process receives the data from the UUT and stores the output data into another text file. Finally, the simulation result is compared with the golden output before the simulation finishes and any discrepancies reported. Additionally, the test-bench also contains the option to dump a VCD file in order to view the waveform of the main signals. The input stimuli are all stored in text files and can be modified by the user with the exception of the sobel and disparity estimator benchmarks, which take bitmap file as input as indicated in Table 3.2.

3.6 Conclusions

This chapter has presented a SystemC benchmark suite, S2CBench which complies with the latest Accellera's synthesizable subset draft and which is freely available online. All designs were successfully synthesized using a commercial HLS tool [21] for validation. S2CBench is mainly targeted for designers wanting to evaluate different commercial HLS tools, as all of main commercial HLS tools support SystemC's synthesizable subset. The test cases have been carefully chosen to represent different application domains amiable to HLS and each of them serves to test the extension of the language support, specific synthesis optimizations and tool performance.

Chapter 4

HLS Design Space Exploration

This chapter highlights one of the most important advantages of raising the level of abstraction: The ability of perform HLS based Design Space Exploration (DSE) without the need to modify the behavioral description. In particular, this chapter makes use of the S2CBench benchmarks described in the previous chapter and proposes an efficient hybrid static-dynamic design space exploration method. The method is based on a decision tree machine learning algorithm complemented with a simulated annealing heuristic.

4.1 Introduction

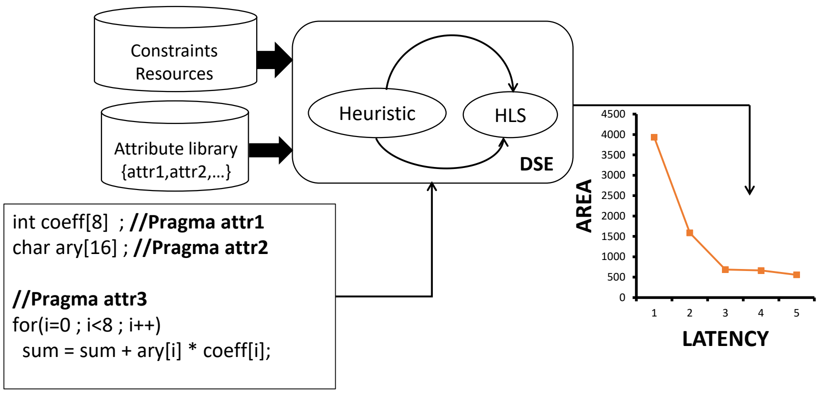
First we give a brief explanation about HLS Design Space Exploration(DSE) and how it takes advantage of *explorable* constructs to generate varying alternative micro-architectures of a single behavioral description.

Definition: HLS DSE can be defined as the automatic process of generating micro-architectures from an untimed behavioral description (*e.g.* ANSI-C or SystemC). Out of all the generated designs, the most important ones are the dominating designs, also called Pareto-Optimal designs. Pareto-optimal designs can be defined as a micro-architecture that is impossible to improve along one axis making the other worse. The curve formed by the set of Pareto-optimal designs is called the Pareto-front. In DSE, the set of all generated designs is called the design space. In the subsequent sections below, the terms Pareto-front and design space will be frequently used throughout.

Commercial HLS tools make extensive use of synthesis directives in the form of pragmas, also called attributes. These synthesis directives are directly inserted in the C/C++ source code as comments and provide information to the synthesizer in order to control the synthesis of the design. Some of the commonly used pragmas include loop unrolling, pipelining, array synthesis as registers or memories and function inlining. Pragma based synthesis has the main advantage of allowing a fine controllability over the synthesis result and the final micro-architecture respectively. The HLS explorer presented in this work, therefore targets the exploration of these synthesis directives to obtain a Pareto-front efficiently. The design objectives used in this work are the area and latency of the designs.

This can be explained using a small example. Figure 4.1 shows the graphical flow of a generic DSE on a HLS description. The DSE process takes as inputs, the behavioral description to be explored, a library with synthesis directives and a set of constraints, *e.g.* maximum delay . The pragmas inserted in the code act as directives to direct the type of resulting hardware implementations for the associated *explorable* constructs like loops, arrays and functions. As a result of the synthesis using the combinations of different pragmas, a variety of alternative micro-architectures are generated.

In the case shown in the figure 4.1, the arrays can be synthesized as memories or registers, and the loops can be fully unrolled, not unrolled, partially unrolled or pipelined. Other solutions outside this curve are non-dominating solutions and hence irrelevant.



```
int coeff[8]  ; //Pragma attr1
char ary[16] ; //Pragma attr2

//Pragma attr3
for(i=0 ; i<8 ; i++)
  sum = sum + ary[i] * coeff[i];
```

Fig. 4.1: DSE overall flow

4.1.1 Problems with HLS DSE

One of the problems with HLS DSE, is that the number of combinations increases exponentially with the number of *explorable* constructs. HLS DSE is a typical multi-objective optimization problem, thus, making it impossible to obtain one single optimal solution. Instead, the solution is a trade-off curve of designs with unique trade-off curves of design objectives [7]. In this thesis, we only consider design objectives as area vs. latency (in clock cycles), but other objectives like power could be easily added. Out of all the resultant micro-architectures, the designer is only interested in the Pareto-optimal designs.

One additional problem in HLS DSE is that Pareto-optimality can only be guaranteed if the search space is exhaustively searched for all possible design solutions. For most HLS descriptions, this is not possible. Firstly, the execution time grows exponentially and secondly, this leads to increase in the time-to-market. Thus, it is common to consider the combination of only best dominating designs obtained by all the heuristics. These are cumulatively presented as the Pareto reference front. Although this procedure is common in most HLS DSE work, however it only allows to compare the quality of the result among different heuristics.

4.2 Related Work

Previous work on HLS DSE can be classified in different ways. One would be based on the technique used for solving the multi-objective optimization problems, namely, Heuristics and Integer Linear Programming (ILP). ILP can lead to the optimal solution, but suffer from scalability issues having worst case run times that are exponential. Heuristics, on the other hand, show much better runtime complexity but create sub-optimal solutions.

Heuristics can be further sub-divided into two main categories: Meta-heuristics and design space pruning techniques that try to reduce the design space [28]. Design space pruning techniques typically try to restrict the design space, hence often missing entire exploration regions. Meta-heuristics search among alternatives and tolerate locally worse solutions with the hope to find global or better sub-optimal solutions. Amongst them, Simulated Annealing (SA) and Genetic Algorithm (GA) algorithms has proven to be very useful heuristics for solving multi-objective optimization problems. However, they suffer from long execution times and are very sensitive to the input parameters (*e.g.* initial temperature, rate of reduction in the case of SA and in the case of GA, mutation and cross-over rate).

Another classification of HLS DSE methods is based on the way they model the impact of a new set of options on the area and performance of the resultant circuit. In particular, these can be either predictive-based or synthesis-based.

The predictive-based methods generate a set of configurations (training set) and then derive a predictive model to avoid having to re-synthesize every new configuration, which is the most time-consuming part of the DSE. In [38] an example of predictive-based HLS DSE is presented, where early estimators of area and delay for FPGA implementations were used in order to evaluate the design space before using any behavioral synthesis. Schafer et al. were one of the first to propose this flow in [72]. Liu et al. [45] extended this flow and proposed a learning-based method based on transductive experimental design techniques. More recent work in this direction include [53]. Other predictive model based methods use response surface models (RSMs) combined with spectral analysis [86] and RSM with design of experiments (DoEs) [59]. The main drawback of predictive methods is that due to the large search space and frequent non-linear behavior (especially at the micro-architectural level) they seldom converge well. This implies that a relatively large training set is required. Typically between 10-20% of the entire search space is reported [45, 59, 72].

Synthesis-based methods generate new configurations based on multi-objective optimization heuristics and synthesize (HLS) each new configuration. These heuristics have been shown to produce good results for these types of multi-objective optimization problems. The meta-heuristics include simulated annealing [77], genetic algorithms [23] or ant colony [16]. Multiple faster heuristics have been later proposed, including techniques based on clustering dependent parameters were proposed to prune the search space before exploration [73]. Other techniques include the internal clustering of dependent *explorable* operations (*e.g.* nested loops) and exploring these separately and then merging the results to obtain the global best solution [74].

Other recent work, specifically target the loop unrolling factor combined with the array synthesis by analyzing the access patterns and then pruning the search space accordingly [65][17]. Most of these methods prune the design space more or less aggressively in order to reduce the execution time [85]. Other works combine both approaches by aggressively pruning the search space and predicting the system performance using genetic Fuzzy systems[3].

In this chapter, we introduce a decision tree learning based method to improve existing simulated annealing algorithm. The method follows a hybrid static-dynamic approach and directly aimed at reducing the execution time of DSE while achieving comparable results to a standard simulated annealing algorithm. In the initial phase, standard simulated annealing is used to generate designs based on the previous design's cost function. These initial designs are used to generate a decision tree to decide which of the attributes contribute to maximizing the cost function. These attributes are then fixed while those attributes which do not show any strong relation are selected pseudo-randomly. This reduces the design space considerably and hence allows finding the Pareto-optimal designs faster than using a standard simulated annealing algorithm.

4.3 Methodology

The primary objective of the design space exploration is to find a set of dominating designs as fast as possible. Ideally, these dominating designs should be the Pareto-optimal designs, although it is virtually impossible to prove the optimality of the results for larger designs. In this work, we focus only on single process micro-architectural DSE. An overview of the complete proposed explorer is shown in the flow diagram in figure 4.2. Our explorer takes as inputs, the behavioral description to be explored (ANSI-C or SystemC), a library file containing all the *explorable* attributes, and the annealing parameters (i.e. initial temperature and cooling rate). The algorithm then generates a new unique combination of attributes chosen from the library and inserts it into the behavioral description. The generated source code is parsed and synthesized in order to extract the area of the circuit from the Quality of Report file (QoR). Although the HLS tool used in this work reports the latency of the synthesized circuit, often this value only represents the state count, because the actual latency cannot be determined statically. This is mainly due to data dependent loops or continue/break statements in the behavioral description. Thus, in order to obtain the accurate latency of the design, a cycle-accurate simulation for every new design is performed. The cost function used for our work, is a linear function as used in Schafer et al. [77]:

$$Cost = A\alpha + L\beta \tag{4.1}$$

where α and β are weights to adjust the effect of the design objectives (area, A and latency, L) respectively. The weights α and β are adaptively modified during the

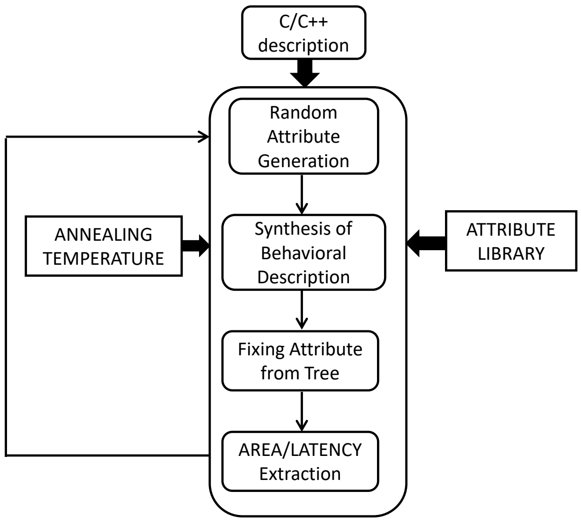

C/C++ description
Random Attribute Generation
ANNEALING TEMPERATURE
Synthesis of Behavioral Description
ATTRIBUTE LIBRARY
Fixing Attribute from Tree
AREA/LATENCY Extraction

Fig. 4.2: Flow diagram of FSA Design Space Explorer

exploration in order to capture the complete trade-off curve. Our explorer continues until a given exit condition is satisfied and returns the dominating results found during the exploration.

4.3.1 Simulated Annealing Review

Simulated Annealing was initially proposed by Kirkpatrick et al. [39], as an effective heuristic for multi-objective optimization problems which is similar to our problem. Since then, different kinds of optimizations have been implemented upon the basic simulated annealing. Simulated annealing has been optimized in various ways [77] [29]. However, the application of machine learning upon simulated annealing heuristic is a novel concept as the proposed method initially learns statically using a decision tree algorithm. Here, the training data is initially generated using several iterations of the standard simulated annealing.

One of the distinct characteristics of SA is that the decision to accept a solution is based on an acceptance probability function P, where $P = f(C', C, T)$. Here, C' is the cost of the new design, C is the cost of the previous design and T is the current temperature. Thus, the new design is accepted if $C' < C$ or if $exp^{(-(C'-C)/T)}$.

Some of the characterizations of SA specific to our implementation are mentioned as follows:

- For our work, we have evaluated the function for updating of temperature as mentioned below where T_{new} is the updated temperature. In this work we empirically make γ to be a constant at 0.95. The performance of SA is dependent on several factors, one of them being γ. Increasing γ enlarges the design space of SA but also increases the running time.

$$T_{new} = \gamma \times T_{old} \tag{4.2}$$

- The stopping condition to exit the algorithm has been evaluated by counting the number of designs which are not accepted by the default rule or acceptance rule. The algorithm exits when it reaches the given threshold or when the temperature is equal to zero. It has been empirically observed that either of these conditions similarly impact the algorithm's performance.

Thus the simulated annealing (SA) is executed in three steps: first, random designs are generated using the random generation function. Second, the acceptance of the design is decided using the acceptance probability function and then finally, the temperature is updated using equation 4.2 when its condition is satisfied.

4.3.2 Attributes for Exploration

The commercial HLS tool used in this work [21] includes a large number of attributes (200+). Since the design space exponentially grows with the number of *explorable* constructs and their combinations, our work focuses on the most important attributes. They include unrolling of loops, folding (pipelining), expanding functions as inline or goto, implementing arrays as registers, memories or as combinational logic circuits. These attributes affect the resulting micro-architecture most significantly. As shown in figure 4.1, these attributes or pragmas are inserted as comments in the source code targeting specific constructs i.e. a *for* loop and two arrays. It should be noted that some of these attributes also may be accompanied by sub-attributes. For example, the *array=RAM* attribute can be made more specific in terms of its specification by combining with sub-attributes such as number of memory ports (dual, two-port, etc.). For reference, the attribute type refers to the type of *explorable* constructs and the attribute value is the architecture desired for that construct, *e.g.* RAM or LOGIC.

4.3.3 Proposed Algorithm

The main objective here, is to find a trade-off curve that should be comparable with that from using standard SA, while reducing the execution time. During the analysis of the behaviour of SA, one of its characteristics is that it tends to traverse through down-hill curves and is likely to end up searching local minima thereby increasing the execution time. In order to make the SA more efficient, different methods have been proposed in the past as mentioned earlier, mainly aiming at improving its efficiency and execution time. The use of decision trees for design

Algorithm 1: Algorithm for implementing Decision tree

```
Data: $AL$ = Attribute List
T = Temperature of SA
Result: $PS$ = Pareto-optimal set of Behavioral designs
1  begin
2      Step 1: Training set Generation
3      Run SA for X iterations
4      Step 2: Decision Tree Generation
5      while $T > 0$ do
6          Calculate $P_+$, $P_-$ proportion of training samples
7          Calculate Information Gain for Attribute type, entropy for attribute
8          Attribute type with highest Information Gain is chosen
9          Root of Tree is created and children nodes as attributes
10         if all attributes in $AL$ are not tested then
11             Decision is taken on Leaf nodes
12             Decision to select the node under which the next attribute type to be
               tested is taken.
13     Step 3: SA with Decision tree
14     Continue SA using Decision tree to fix attributes that minimize cost function
15     Step 4: Delete all non-optimal designs
16     Step 5: Select Pareto-optimal designs
```

space exploration has been limited to either memory management levels [5] or on multi-processor platform [7] explorations.

Our proposed new methodology for optimizing the simulated annealing for effective DSE in HLS uses a type of dynamic programming technique. Decision tree learning (ID3 algorithm) is one of the most widely used and practical methods for inductive inference [54]. In our work, the objective of the decision tree is to predict the attribute combinations that make the most positive impact i.e. reduces the cost function. For this purpose, the decision tree must be able to infer the impact of all attribute combinations from the training set. The decision tree learns the performance of different attribute combinations from the training set. The decision tree's prediction is used to fix some of the attributes that have been predicted to make the

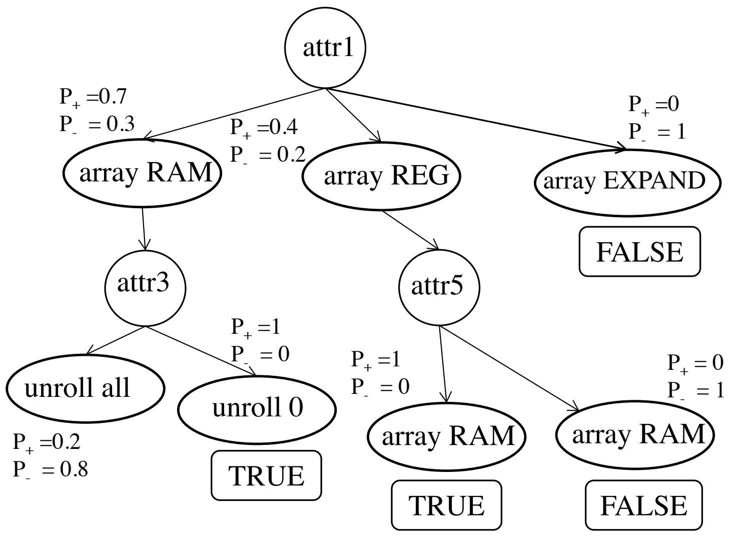


Fig. 4.3: Partially learned decision tree

highest impact on reducing the cost function. When SA is resumed, the predictions from the decision tree are used to fix certain attributes and the rest are generated randomly during the exploration. The basic function of decision trees is to classify instances from the training set into attributes that contribute positively or negatively towards reducing the cost function. The tree structure thus becomes hierarchical in nature as shown in figure 4.3. The algorithm for implementing the decision tree in the explorer is described in Algorithm 1. The overall methodology is executed in the following steps.

Step(1) Training Set generation: In the initial phase of the explorer, the standard simulating annealing algorithm is executed. The designs generated at this stage are half of the total number of possible combinations of attributes, N_h, in the library, which act as the training set for creating the attribute tree. The exit condition at this stage is evaluated by checking if the number of designs generated have reached the value of N_h. (line 2 of Algorithm 1)

Step(2) **Decision tree generation:** Once the training set is created, our method builds a decision tree of attributes based on the ID3 algorithm. The term attribute (ν) refers to the specific attribute option inserted into the source code. The term, attribute type (A), refers to the category of attributes which perform a specific family (loop, array, function) of function under which the attributes are categorized. Each attribute is evaluated using a statistical test to determine its impacts on the quality of the training set [54]. (lines 4-11)

Step(3) **SA with decision tree:** Once the decision tree is created, our method continues by generating pseudo-random attribute configurations using the decision tree. Attributes with high impact on reducing the cost function are fixed based on the criteria explained previously, while attribute with no clear decision are selected randomly. (lines 12-13)

Step(4) **Delete non-optimal designs:** The last step of our method is the deletion of the dominated designs (non-optimal) in order to obtain the designs on the trade-off curve. For this purpose, our method re-visits all the designs generated and keeps only the dominating ones. (line 14)

4.3.4 Decision tree learning

Prior to the explanation of the method of generating attribute tree using ID3 algorithm, we define the following terms:

Entropy: Information Gain (IG) as the statistical property for the purpose of sorting the attributes (ν) is based on their capacity to maximize/minimize one of the cost function objectives (Area or Latency). The *entropy*), is used to characterize the purity or impurity of a specific attribute for a collection of samples, S.

$$Entropy(S) = -p_{+}log_{2}p_{+} - p_{-}log_{2}p_{-} \tag{4.3}$$

In the equation, p_{+} represents the positive proportion of the training examples, and p_{-} represents the negative proportion of the examples. The term positive proportion refers to the fraction of the training designs satisfying the minimization of an objective and negative proportion is the fraction not contributing to minimizing of objectives. With reference to DSE in HLS, the process of classifying the training designs into a tree is different from its standard implementation used by ID3 algorithm. For every design objective, the preliminary training set is initially sorted in ascending order, and then the value of the objective of the first example in the sorted set with the addition of 20% of its value is considered for setting the range of threshold. For our implementation, all designs with objective values below this threshold are positive. Empirically, it has been shown that the indicated range achieves the best results. The design space to be explored can be easily controlled by changing the threshold value. Higher threshold value will fix more attributes and hence reduce the design space to be explored reducing the execution time, while specifying a lower threshold value would fix fewer attributes and hence increase the design space and thus also the execution time. The user can therefore set this parameter in our system. Given the entropy as a measure for the weight of each attribute, we define information gain, IG_i which decides the best attribute type as a root node to be effective for classifying the designs.

Information Gain(IG): The information gain (IG) of an attribute type A_i, relative to a set of training designs, S is formulated as,

$$Gain(S,A) = Entropy(S) - \sum_{v \epsilon values(A)} \frac{\mid S_\nu \mid}{\mid S \mid} Entropy(S_\nu) \tag{4.4}$$

In the above equation, values (A) is the set containing all the attributes under attribute type A and S_ν is the subset of S for which the attribute type A has attribute ν. The first term in the equation is the entropy of the original collection of the examples S, but the second term is simply the summation of the expected values of entropies of each specific attribute under the attribute type A. Thus, $Gain(S, A)$ is the information provided about the classification of the sets, given the values of the attribute type A.

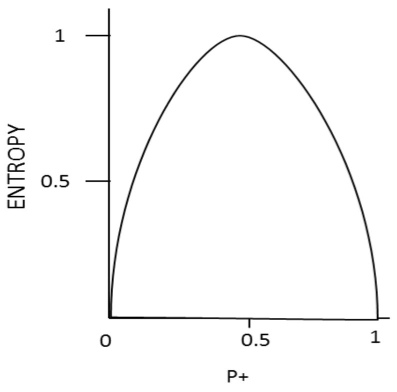


Fig. 4.4: Graph representing the relation between entropy and p_+

The ID3 algorithm uses a greedy, hill- climbing approach, beginning with an empty tree and subsequently progressing by searching through the entire design space for a tree which would completely classify the training data. In this work, we have a binary target function for classifying each attribute in the library. For each attribute, the decision is either true or false. A leaf node in the tree is declared only when the decision about the attribute has been taken i.e. if the attribute has either zero true values or zero false values in the subset S_ν. At each node of the tree, the attribute type with the highest information gain is chosen for being tested at that level. The procedure continues until all the attribute categories have been completely exhausted or all the nodes in the tree have been classified as leaf nodes. The algorithm in figure 1 shows the summary of the implementation method for creating the attribute tree. A partially created tree consisting of roots representing attribute types and nodes representing the attributes is shown in figure 4.3. During the tree search at each level, the attribute declared as a leaf node and having a *true* value, is accepted for fixing, otherwise the attribute having the maximum entropy is selected for fixing in the newly generated design. The rest of the attributes are left for random selection. A *true* value indicates that the attribute satisfies the particular design objective.

One of the important features of this implementation strategy of the decision tree is the selection of an attribute under a root, in a situation when the root does not have any leaf node with a *true* value. In these situations, we have developed a unique strategy in order to find the exact attribute combination which contributes to minimization of the design objective. The relation of entropy of an attribute and the positive proportion of examples related to that attribute represents an inverted bell curve as shown in figure 4.4. If p_+ of an attribute is less than 0.5, then the entropy is directly proportional to p+ otherwise the entropy is inversely proportional to p_+. Depending on the characteristics of the nodes under a particular root, if there are no attributes with p_+ more than 0.5, then the direct proportionality rule is applied to check for the attribute with maximum entropy. If a single node exists with entropy more than 0.5, then the inverse proportionality rule is applied to check for the attribute having the minimum entropy since with the p_+ more than 0.5, the lesser the entropy, higher is the positive proportion. Thus, in this way, in the absence of true leaf nodes, the attributes with highest p_+ prove that they contribute to relatively more number of designs having true values for the target function and those attributes are chosen. This procedure proves to be more efficient, in a way such that the search technique is more focused on the combinations predicting to lead to minimizations of the design objectives.

As shown in figure 4.3, the most important factors for creating the decision tree is the information gain of each attribute type and the entropy of each attribute. While the information gain helps in deciding the root at each level of the tree, the entropy enables the decision for which attribute shall be used for inserting the next root in the tree. The mode of selection of attributes from the tree is a depth-first search. At each level of the tree, only if the attribute is a leaf node with all positive proportions, then that attribute is selected otherwise the attribute with highest entropy is selected.

The positive proportions of each attribute play the most important role in deciding the number of attributes to be fixed from the tree. For satisfying the design objectives, it is important to choose the attribute having the largest positive proportion, or having entropy as zero. It may be noted that while creating the decision tree, the impact of the attributes on the quality of designs decreases as we go downwards the tree. Thus, the priority of the attribute types is sorted in descending order throughout the tree. Thus, the upper half of the tree shall prove to be more effective. Once the decision tree is created with its classifications for the different attributes, the tree is parsed to obtain a unique list by selecting the attributes from each level within the upper half of the tree until a leaf node is reached. This list is used for fixing the attributes as they are confirmed to have the highest impact on reducing the cost function. The process is continued as shown in figure 4.2 until exit criterion is fulfilled.

The most timing consuming parts of our proposed method are the synthesis time after every unique attribute configuration is generated and the cycle-accurate simulation required to determine the exact latency of each newly generated configuration. Nevertheless, by fixing some of the attributes it is possible to considerably reduce the exploration space and thus reducing the execution time. Upon analysis, it is seen that performance of the decision tree depends on the initial X iterations of the simulated annealing. First, the standard SA algorithm is studied for its behavior. When the T_{new} parameter is at the maximum, the tendency of SA for choosing attributes for a design is purely random. It is also observed that higher the value of X, the more attributes will be fixed by the decision tree thereby leading to a smaller search space for SA. Given this case, the decision tree has higher chance of over-fitting, and SA might leave out some designs on pareto-front. Thus, there is always a trade-off between the algorithm's running time and the quality of the resulting pareto-front with the number of iterations. Choosing a low value of X leads to under-fitting of the tree. Thus, it is important to have a balance between the two scenarios.

4.4 Experimental Results

The experiments for validation of the proposed methodology were performed using benchmark designs from the open source Synthesizable SystemC Benchmark suite S2CBench [34]. Our proposed method FSA is evaluated by comparing against the DSE method using standard Simulated Annealing. Although there are various methods used in DSE discussed in related work, FSA method acts as a helper based improvement upon standard SA method, thus SA is the baseline method for comparison. Table 4.1 summarizes the results of analysis of exploration using SA algorithm and our proposed FSA algorithm. Column 1 indicates the benchmark name and

Table 4.1: Experimental Results for FSA

Benchmark		*execution time*			*Dominance*		*Cardinality*	
name	lines	SA [sec.]	FSA [sec.]	diff [sec.]	SA [ratio]	FSA [ratio]	SA	FSA
Fibonacci	31	343	182	161	1	1	3	3
IDCT	131	4202	4167	35	1	1	2	2
Snow 3G	311	5519	3161	2358	1	1	1	1
FIR filter	110	2179	934	1245	0.8	0.6	4	3
Gfilter	381	2309	1922	387	1	1	3	3
AES cipher	259	5281	3398	1883	0.3	0.67	1	2
Kasumi algorithm	291	11376	6292	5084	1	1	2	2
ADPCM encoder	113	892	505	387	1	1	2	2
Avg.	203.3	4012.6	2570.1	1443	0.8	0.90	2.1	2.2

column 2 their size in terms of numbers of lines of code. *Fibonacci* is a Fibonacci sequence generator, IDCT an inverse cosine transform, *Snow3G* is a stream cipher which produces a key stream that consists of 32-bit blocks using a 128-bit key, FIR is a 9-tap FIR filter. *Gfilter* is a graphical filter, *sobel* an edge detection filter. Finally AES is an advanced encryption standard implementation and ADPCM is an Adaptive-differential pulse-code Modulator (only encoder part).

The experiments were run on an Intel dual 2.40 GHz Xeon processor machine with 16GBytes of RAM execution Linux CentOS version 3.11.7-200. The HLS tool used is CyberWorkBench v.5.4 [21]. The target HLS frequency in all cases is 100MHz (10ns maximum delay) and the target technology is Nangate's OpenCell 45nm ASIC technology [57]. The execution time discussed here comprises of the entire flow from the parsing of the behavioral descriptions until the end of the DSE. Two quality indicators were used in order to compare our proposed method (FSA) against a standard SA: (i) Pareto dominance and (ii) Cardinality. Pareto dominance is equal to the ratio between the total numbers of points in the Pareto set being evaluated, also present in the reference Pareto set. The higher the value, the better the Pareto set is. The cardinality indicates the number of dominating designs found by each method. A high cardinality indicates a larger number of solutions to choose from, which should be considered to be positive, although it needs to be interpreted carefully with the rest of the results. The results reported were computed by comparing the pareto-front of each method compared to the reference Pareto front (the combination of the best non-dominated results of each method). The execution times for each benchmark are used for quantitative analysis.

Columns 3 to 5 in Table 4.1 show the execution times of SA and FSA executions and their differences. Results indicate that our proposed FSA explorer runs faster than SA by an average of 36% and can run up to a maximum of 48% faster than the standard SA algorithm (almost 2x faster). According to our qualitative analysis, our proposed FSA algorithm produces comparable sets of dominating designs compared to SA while execution almost twice as fast. Based on these results it can be concluded that our proposed algorithm produces the similar results as a generic SA algorithm, while reducing the execution time significantly.

4.5 Conclusion

High Level Synthesis is becoming more necessary in order to further increase VLSI design productivity. In this chapter we have presented HLS DSE explorer that enables designers to automatically generate a trade-off curve of unique micro-architectures given a behavioral description for HLS. Results show that our proposed method is faster than a standard SA-based explorer by a maximum of 43%. Since our algorithm statically learns the combination of attributes which minimizes the given cost function, it can reduce the search space considerably and hence reduce the execution time of the exploration.

[0]This work has been published as proceedings of Electronic System Level Synthesis Conference (ESLSyn), 2014.

Chapter 5

VeriIntel2C: Abstraction of C from RTL to enable Design Space Exploration

As discussed earlier, there is a need for abstracting RTL to C. Thus, a new method called *VeriIntel2C* is introduced to abstract C program with *explorable* constructs from RTL designs given in synthesizable Verilog. The subsequent section starts by describing the modeling of the Petri Net based framework that generates the Control Data Flow graph (CDFG). In the second phase, the rule-based search methods including graph matching is introduced and described. Finally, the experiment results section evaluations the quality of the conversion.

5.1 Introduction

Most System on Chips (SoCs) are created from a combination of legacy Register Transfer Level (RTL) blocks, Intellectual Properties (IPs), newly developed RTL and C/SystemC descriptions synthesized using behavioral synthesis (also called High-Level Synthesis). This re-use of existing legacy code allows design teams to focus only on the new features that need to be implemented, thus reducing time-to-market considerably.

However, one problem faced by most VLSI design teams using HLS is having large amount of legacy RTL code/IPs. It is challenging for the teams to integrate newly generated C/SystemC code with those legacy RTL designs [8]. RTL designs are challenging to re-target using different constraints because of their highly restricted levels of abstraction. In contrast, C-based designs, using commercial HLS tools, allow to generate alternative micro-architectures with varying design metric trade-offs without having to modify the original code, called as Design Space Exploration (DSE). Also, C designs enable maximum resource sharing [20] using HLS, and their efficiency can rival hand-coded RTL designs for Digital Signal Processing (DSP) applications [18].

DSE using HLS are most exploited by targeting *explorable* constructs in a HLS based design. *Explorable* constructs imply constructs that when explored result in very different RTL micro-architectures, which result in a larger design space. Examples of these constructs maybe loops, arrays, functions.

For this reason, a previous work, Bombieri *et al.* [9] (*R2C*) proposed abstraction methods to convert legacy RTL IP components of SoCs to HLS-optimized C++ programs in order to enhance re-usability enable DSE at the system-level using HLS commercial tools. This approach translates RTL designs syntactically with all processes (synchronous and asynchronous) mapped into C++ functions, while using static-scheduling to resolve the functionality. Their method is able to recover the IP functionality in the form of C++ program with functions, and also enable component-level and system-level exploration.

This approach, however, has its own set of limitations. The method firstly does not focus on identifying loops, arrays from the RTL design. During the translation, it identifies and rolls up loops only when the RTL design contains *for-generate* syntax statements or multiple instances of RTL *modules*. Loops and arrays in RTL can be described in different ways and are mostly not straightforward with *for* or *while* syntax, thus allowing designers to write RTL designs in diverse styles. This makes RTL designs without *for-generate* syntax and multiple modules unable to be converted using this approach. Their approach works well if the RTL IP contains multiple components, from where they create input/output (I/O) arrays of the interfaces connecting these components. However, I/O arrays are unable to significantly improve the component-level DSE of a RTL component as operations of I/O arrays are restricted to purely read or write, implementable in the form of registers or wires. Their approach treats a RTL design as a system with several modules as individual components with their main focus being to roll up instances of every component into functions and create I/O interfaces, to enable and enrich system-level exploration.

It is therefore desirable to design abstraction methods to abstract C/SystemC designs optimized for DSE in particular, from RTL designs with purely single modules. C-based designs optimized for DSE must consist of *explorable* constructs, in particular, loops and arrays. As the level of abstraction of RTL is highly limited, the loop forms like unrolled, partially unrolled are described indirectly using hardware details. Also, arrays in the computations are embedded within the RTL as registers, wires, logic, memories, etc. which makes the identification further challenging.

In this chapter, we address the aforementioned problems by proposing a robust translation method, *VeriIntel2C*[49] that identifies the *explorable* constructs from

```
module ave8(clock, reset,indata, avg);
input clock;
input reset;
input [7:0]indata;
output [7:0]avg;
reg [7:0] rg1,rg2,rg3,rg4,rg5,rg6,rg7,rg8 ;
reg [11:0] o1;
  always @ (posedge clock or posedge reset)
  begin
  if (reset)
    begin
    rg1 <= 0; rg2 <= 0; rg3<=0; rg4<=0;rg5<=0; rg6<=0; rg7<=0; rg8<=0; o1<=0;
    end
  else
    begin
    rg1 <= indata ; rg2 <= rg1 ;rg3 <= rg2 ; rg4 <= rg3 ;
    rg5 <= rg4 ; rg6 <= rg5 ;rg7 <= rg6 ; rg8 <= rg7 ;
    o1 <= rg8 + rg7 + rg6+ rg5 + rg4 + rg3 + rg2 + rg1;
    end
  end
 assign avg = o1[11:3];
endmodule
```

Fig. 5.1: RTL (Verilog) design example that computes the average of 8 numbers (*ave8*))

RTL designs that, upon DSE, have the strongest impact on the resulting micro-architecture. Our proposed approach purely focuses on generating C designs with *explorable* constructs to enable the DSE of RTL designs with single modules to broaden the design space of micro-architectures. Our robust translation methodology is able to identify array structures in forms of registers, wires, or memories and in variable or constant forms unlike previous approach as well as internal and I/O. While the functionality is extracted and preserved using Hardware Petri Nets, the generated CDFG along with the various graph search techniques allow our methodology to identify loops in unrolled, folded, nested loop forms from RTL designs and generate them in

```
int rg[8]; //array
int ave8(){
    int j,sum=0, indata;
    //loop
    for(j=7;j>0;j--)
       rg[j] = rg[j-1];
    rg[0]=indata;
    sum = rg[0];
    // loop
    for(j=1;j<8;j++){
        sum = sum + rg[j];
    }
    return sum/8 ;
}
```

(a) Resulting C design from *VeriIntel2C*

```
int ave8(){
    int sum=0,indata;
    int rg1,rg2,rg3,
    rg4,rg5,rg6,rg7,rg8;
    rg8=rg7; rg7=rg6;
    rg6=rg5; rg5=rg4;
    rg4=rg3; rg3=rg2;
    rg2=rg1;
    rg1=indata;
    sum=rg1+rg2+rg3
        +rg4+rg5+rg6+rg7+rg8;
    return sum/8 ;
}
```

(b) Resulting C design from previous work (*R2C*)

Fig. 5.2: Resulting C code comparison of *VeriIntel2C*with previous work for *ave8* design

the resulting C programs.

Figure 5.1 shows a typical RTL design that computes the moving average of 8 numbers. Figure 5.2 shows the results of the translation of the given RTL design. Figure 5.2a shows the C code generated by the proposed method in this chapter *VeriIntel2C*, while Figure 5.2b shows the results obtained by the closest previous work [9]. The differences are clear. *VeriIntel2C* is able to generate a loop and an array which in turn can be explored generate micro-architectures with different trade-offs, while the previous work C code cannot be explored as it does not have any *explorable* constructs.

Figure 1.2 in chapter 1 showed the overview of the system using the proposed abstraction methodology indicated in dotted boxes. The proposed methodology uses an extended Hardware Petri Nets [50] to model and extract the behaviour of RTL designs (synthesizable) written in diverse styles, unlike previous approaches, and generates a CDFG. The CDFG is then analyzed using graph matching techniques along with a rule-base to identify loops in unrolled, folded, nested loop forms. These forms identified from RTL designs are generated in the resulting C programs. The HLS DSE is purely used as a tool for our experiments to prove the quality of the resulting C designs from *VeriIntel2C*. The impact of the generated C design with loops, arrays on the design space can be evaluated by performing HLS DSE and analyzing the trade-off curve of micro-architectures obtained. It is to be noted here that the proposed translation method does not identify or generate pipelined loop forms and nested loops that have conditional blocks.

The next section discusses the literature review of previous works related to this topic. Section 3 describes the proposed methodology in detail, consisting of three components. The first component describes the Petri Net modelling of RTL designs, followed by the mapping method for generating the CDFG. The third component introduces and describes the rule-based search and graph matching methods to generate and construct loops and arrays. Finally, the experiment section describes the experimental setup and the experimental results.

5.2 Related Work in RTL to C Translation

With respect to abstraction of C/C++ from RTL, most of the approaches describe the generation of C/C++ models to optimize simulation. Verilator [83] is a simulation tool that generates a generic C++ executable from RTL (Verilog) but does not produce a *synthesizable* C program which can be directly used for HLS or DSE. In Stoye et al. [79], synthesizable RTL (Verilog) design is again translated into C++

executable to reduce the number of delta cycles in simulation by merging processes together. Bombieri et al. [10] generate C++ models by abstracting architectural details of the original RTL code for faster simulation. There are also some commercial products that target this application domain. Aldec [2] tool converts RTL test-benches into C++. Carbon Design Systems (recently acquired by ARM) [15] converts RTL models into cycle-accurate and register-accurate C++ models targeted mainly for the creation of virtual platform from legacy RTL code. The purpose of the above mentioned works is mainly to enable faster verification and do not target to create synthesizable behavioral designs with high-level constructs.

Approach for DSE: Closely related to this area, a recent approach discussed earlier, Bombieri et al. [9] (*R2C*) focuses on the optimization of RTL to C++ conversion for HLS DSE with the similar objective of maximizing the design space of the HLS-optimized RTL designs and IPs. As described earlier, although their work is motivated for enabling DSE as *VeriIntel2C*, their method aims to enable system-level exploration with RTL designs treated as a system with modules as components. Whereas, the proposed work aims to focus on DSE for single-module RTL micro-architectures with no components or interfaces. Secondly, their method fails to identify and generate internal arrays and loops in the absence of *for-generate* syntax, which can be easily shown in the experiment results section.

Structural approaches: Other works related to this thesis are presented here. Fummi et al. [10] transforms a RTL IP model into C++ by extracting and generating Extended Finite State Machines (EFSM). It also uses a rule-based approach to abstract the RTL design aiming to preserve concurrency of the original design. Their method generates a software which attempts to partition the RTL code into several processes e.g., sequential and combinational statements, and concurrent statements. The processes are simulated using a simulator model (scheduler), to extract the behavior and ultimately generate the EFSM with the C++ code. CHESS tool [56] proposes a method to directly extract the CDFG from VHDL designs for HLS of VLSI systems. Here, the method heavily relies on the presence of *high-level* VHDL constructs like *if-else, for*, etc., to generate CDFGs with abstracted loops and array constructs. Song et al. [78] proposes an automatic method to extract pure dataflow paths from RTL designs, but their objective is to verify functionality of large-scale synchronous VLSI designs, and they remove all the control information, which does not help the objective of this thesis. Bombieri et al. [11] presents an approach for automatic generation of C++ code by abstracting HW details from existing RTL IPs and extracting their functionality. However, the objective of this work is to enable HW/SW partitioning for MPSoCs and does not focus on enabling DSE of RTL designs, thus not aiming to create loops or arrays.

Petri Net related approaches: One of the main contributions of this work is the use of Petri nets to generate the CDFG since direct transformation to CDFG may result in a loss of inter-connections between control and data signals and define dual functionalities of different signals which are present in complex RTL designs. This is further explained in detail in the following section.

Thus, the important related work in the use of Petri nets is also reviewed in this section. Petri net, as a model, has had many applications at various levels of abstraction in hardware design. A summary of the most relevant works can be found in [50]. A HLS system called CAMAD uses timed Petri nets for modeling the intermediate design description at the behavioral level, prior to subsequent synthesis transformations to finally obtain an RT-level representation [64]. In another work [67], Petri nets have been modeled using VHDL to simulate the operations at different levels of abstraction. For modelling at the behavioral level, Rust et al. [71] proposed an approach for realizing Petri Net models in SystemC. Petri Nets have also been used for modeling asynchronous digital circuits, whose basic analysis and explanation has been provided in [87]. Petri Nets described in the previously mentioned work have been mainly used for verification, modelling, and synthesis. In another work [70], the authors extend Petri Net models to Hardware Petri Nets (HPN), used as an intermediate representation for generation of synthesizable VHDL code.

The work of this chapter is different from previous works, because it considers all the main *explorable* constructs (arrays and loops) in HLS DSE and explicitly targets their generation without relying on fixed or specific syntax in the RTL code. Moreover, the use of Hardware Petri Nets (HPN) [70] in *VeriIntel2C* is unique, as previous works using HPN have not targeted reverse translation methods from RTL to C. *VeriIntel2C* extends HPN to define a Petri net graph and extract the behaviour of the synthesizable RTL design in order to facilitate the extraction of loops and arrays. A clearer explanation with a use case of the Petri Nets is explained in the following sections which highlights the importance of Petri Net in this chapter.

5.3 RTL to C Abstraction Methodology

This section presents *VeriIntel2C*, a translation methodology for RTL descriptions to generate behavioral descriptions optimized for HLS DSE. The objective of the methodology is to identify structures of loops and arrays in diverse forms. These forms can be represented in different ways in a RTL design and it is imperative for a method to effectively identify the behaviour of these structures and map them to loops and arrays accordingly.

The novelty of *VeriIntel2C* lies in the robust methodology that can generate flattened, folded, nested as well as variable (R/W), read-only, constant array forms and their operations, independent of the RTL program style and constructs (structural or behavioral).

The major components described in the proposed *VeriIntel2C* framework are: Petri Net (PN) modelling of the RTL program (section 5.3.1), CDFG generation (section 5.3.2) and finally, loops and arrays identification (section 5.3.3). The work flow of *VeriIntel2C* consists of two phases. In the first phase, the first step is the novel mapping method of RTL components to model Petri Net graph representation. The next step generates the CDFG from the Petri Net. The second phase is loops and arrays identification which analyzes the CDFG using rule-based graph search and matching techniques to generate the C program with *explorable* constructs.

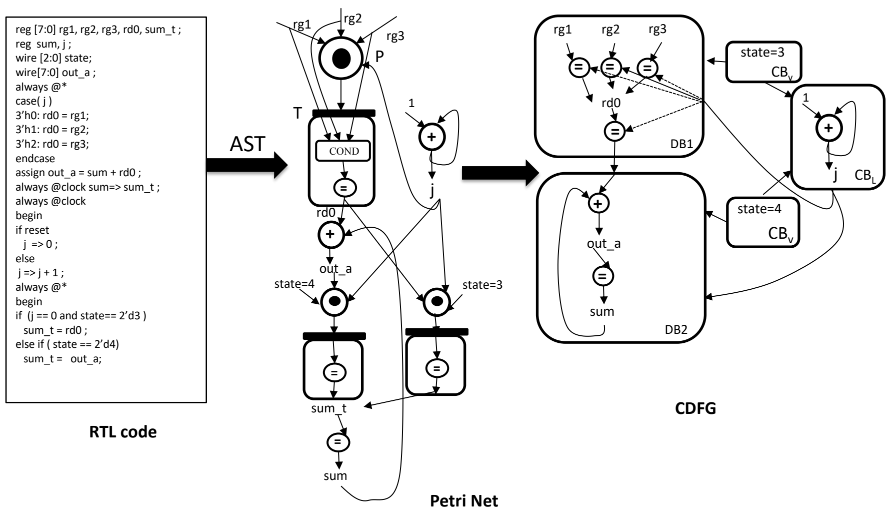
5.3.1 Petri Net Graph

Overview. In this section, a novel mapping method between different RTL design components and Petri Net graph is proposed, and the model specification of the Petri Net graph is defined. The use of Petri Net graph for behaviour modelling is significant for *VeriIntel2C*. Petri Nets have been commonly used for modelling in EDA [50], but mostly to represent the control part of synchronous designs. In contrast, the approach in this chapter extends the HPN to abstract C program with loop and array structures from RTL designs. Petri Nets is used instead of typical state machines because they have shown to be a powerful formal method to model the behaviour of parallel designs at a high abstraction level (C/C++) and they are more flexible in nature. The Petri

Net representation of RTL components enables to model the underlying behaviour of the control and data flow in RTL, so that the operations and their sequences can be obtained for the resulting C program. Thus, the mapping to Petri Net identifies the nature of the components, if any are of dual nature, control and/or data dependent, and enables their separation in the CDFG irrespective of their design styles in the RTL program. Overall, the role of the Petri Net graph is key to create the CDFG without which the CDFG will be a pure data-flow graph without control blocks.

Any given RTL design represents two separate aspects of the process, computational (data) and control but the design does not explicitly distinguish between control and data. The proposed mapping method is required to model the behaviour of the RTL design, because it ensures to represent RTL components of varying structures and enables a simplified view of the inter-connection between the components which may be control or data in nature. This is needed since the objective is to preserve the operation sequences and their dependencies.

C program is purely sequential in nature in contrast to the concurrent nature of the RTL design. Thus, only the information about the sequence of operations and their dependencies needs to be preserved during the translation and the rest to be abstracted away. The mapping method of Petri Net graph representation abstracts away the hardware level details from the RTL design and preserves the information necessary to represent the control and data dependencies between every RTL component and their sequences. Thus, the objectives of modelling a Petri Net are twofold: (1) model behaviour between data and control components in RTL design to identify functionality irrespective of program style (2) enable the CDFG to identify and separate control and data loop structures.



```
reg [7:0] rg1, rg2, rg3, rd0, sum_t ;
reg  sum, j ;
wire [2:0] state;
wire[7:0] out_a ;
always @*
case( j )
3'h0: rd0 = rg1;
3'h1: rd0 = rg2;
3'h2: rd0 = rg3;
endcase
assign out_a = sum + rd0 ;
always @clock sum=> sum_t ;
always @clock
begin
if reset
  j  => 0 ;
else
 j => j + 1 ;
always @*
begin
if  (j == 0 and state== 2'd3 )
  sum_t = rd0 ;
else if ( state == 2'd4)
  sum_t =   out_a;
```

Fig. 5.3: Flow of translation from RTL to CDFG for a simple motivational example design

The transformation to Petri Net graph from RTL design has to be achieved using an intermediate representation. For a faster and stable transformation, the RTL program was initially parsed using a commercial parser (Verific [82]) to obtain a parse tree. The parse tree ensures stability and representation of different design styles. Since this parse tree is difficult to traverse or manipulate, for implementation, the parse tree is converted to an Abstract Syntax tree (AST) categorizing the different types of RTL constructs into different nodes in the tree.

We extend the basic HPN [50] to define the Petri Net graph structure using definition .

Definition 5.3.1. Petri Net(PN) is a 3-unit tuple: $PN = (P, T, OP)$ where P is a place , T is a transition and OP is an operation node.

The fundamental structure of the Petri Net graph extended from HPN [50] is devised in figure 5.4. In this representation, the place P holds all data/control variables

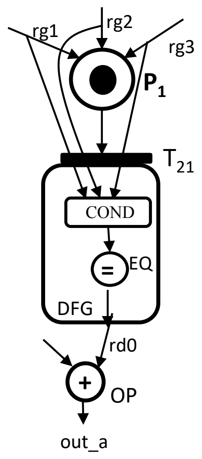


Fig. 5.4: A fundamental view of Petri Net defined for *VeriIntel2C*

which affect the data-flow inside a transition. A transition T describes data-flow operations using a DFG. A data-flow graph DFG is used to model processes inside *Always* blocks of RTL designs. DFG is described in detail in the next section. A Petri Net graph is modeled for a motivational example of RTL design in figure 5.3. In figure 5.3, variables $rg1$, $rg2$, $rg3$ are inputs to a place which controls a transition represented by a conditional ($COND$) and assignment node (EQ) respectively. A control variable is only present in the input to a place, whereas a data variable is an input to a place as well as a transition. Moreover, *Always* RTL components with *for* loops can easily be mapped to a DFG using the mapping rules presented below.

The edges of the Petri Net graph play a key role to represent the dual behavior of certain components. If an output of a place or an OP node controls another place, the edge is incident on that place from the variable but is not incident on its corresponding transition. This allows the framework to have better reachability in terms of searching control nodes. Before we describe the mapping rules used to model the Petri Net graph, the most important RTL design components will be briefly explained to enable the mapping method.

Always block: Synchronous and asynchronous processes are described in *Always* block i.e., procedural blocks where the data-flow operations are executed based on the control inputs in the form of a sensitivity list.

Continuous assignments: Assignment expressions which are described outside procedural blocks remain in force until the variables are de-assigned and are independent of any conditions.

Non-blocking assignments: Assignment expressions are dependent on 'clock' or 'reset' signals synchronous in nature and are mostly assigned to registers. Since a behavioral design does not require the presence of these signals, the expressions are treated as ordinary assignments in C-based program.

The RTL design consisting of above mentioned components are mapped to model Petri Net graph using the following mapping rules:

1. The place and transition nodes are formed from *Always* RTL blocks where a place holds all the control and data inputs of *Always* component.

2. Conditional nodes ($COND$) are mapped from *if-else* and *switch-case statements* in RTL design which is internally created using assignment nodes.

3. Operation nodes (OP) are formed from statements with arithmetic operators in RTL design.

4. Assignment node(EQ) are mapped from various forms of procedural and continuous assignments in RTL design, shown in Table 5.1.

Table 5.1: Types of combinational constructs

Construct	Example
Assignments(variable)	$out\_a = sum$
Assignments(constant)	$out\_a = 4'h9$
Operations	$out\_a = sum + rd0$

5. If back edges are identified in a path using the classic back-edge detection method [20], loops are formed in the path of connecting the nodes.

Using these mapping rules, The Petri Net graph is able to model RTL designs independent of program style. There are several important features of the Petri Net graph representation which are to be mentioned. Firstly, the graph is able to recognize the *Always* block with multiple statements in the RTL code and identify them as separate place-transition pairs. Secondly, the mapping method, while traversing the RTL program, identifies back edges in a path using the classic back-edge detection method [20], and forms a loop in the path of connecting the nodes. Thirdly, the resulting model enables to identify the nature of the loops formed i.e., control or data. Overall, the Petri Net representation enables to generate a CDFG from which the complete data-flow can easily be separated from the control path.

5.3.2 CDFG generation

We define the CDFG for VeriIntel2C using definition :

Definition 5.3.2. Control Data Flow Graph(CDFG) is a directed graph that shows data and control dependencies inter-connected to each other using the form:
$CDFG = (CB, DB)$ where $CB = (CB_L, CB_v)$, $DB = (OP, EQ)$ and every graph is of the form $G = (V, E)$ and V = Vertex, E = Edge connecting between V.

In the generation of CDFG, we propose a mapping method from Petri Net graph. A CDFG can be described as a directed graph that represents control and data dependencies among different functions. From the definition in 5.3.2, CB is a control block which can be of two types. CB_L is a graph G that has the form of loop sub-graph. CB_v is of the form G that can have only V. DB is a data block having the standard data-flow graphs described by G. Some of the major features of the CDFG created in this section are: it adapts its structure based on the corresponding Petri Net graph and all the loop sub-graphs from the previous stage are classified as control loop and data flow loop graphs.

Previous works like [56], [78] have described methods to extract CDFG from RTL designs for HLS where mostly, the data-path is extracted from the module and purely created as a CDFG and the control blocks are either trimmed or not required. However, in the CDFG, we separate control blocks with control loops and control variables from the data blocks which contain the data-flow path.

We describe the mapping rules from Petri Net graph to the CDFG structure as follows:

1. A loop sub-graph in Petri Net that purely controls the place nodes is transformed to a control loop block (CB_L) shown in figure 5.3.
2. A loop sub-graph which does not control a place is mapped inside a data block.
3. The nodes that generate a control input to a place in Petri Net are mapped as control blocks and the transition of the place is mapped inside the corresponding data block.

Figure 5.3 using the above mapping rules shows control loops separated from data-flow and *state* variables are mapped as control blocks (CB_V). The dotted edges indicate that the control block controls the assignments for certain conditions which are stored internally.

Figure 5.3 uses the mapping rules stated in the above sections to create a CDFG. Given an example RTL design that computes the sum of 3 numbers, a Petri Net graph is obtained where it shows the different *always* blocks in a loop and operations acting concurrently. Using the concept of place, transition nodes in Petri Net, the data and control operations are separated in the CDFG. Variables *rg1*, *rg2*, *rg3* are inputs to a place which controls a transition represented by a conditional (COND) and assignment node (EQ) respectively. The loop with j variable is a control signal to the place nodes, thus it becomes a control loop block in CDFG, CB_L. In this way, the Petri Net graph next enables the generation of the final CDFG.Thus, the generated CDFG represents diverse forms of array and loop operations which enables the methods in following section to identify those structures.

5.3.3 Loops and Arrays Identification

In order to form loop and arrays that broaden the design space, this section uses the generated CDFG in section 5.3.2 in the proposed rule-based classification method with graph matching. The proposed method *VeriIntel2C* identifies patterns in the generated CDFG that map to different types of loop and array operations of a resulting C program. This rule-based method poses an advantage over the usual continuous graph-traversal or graph analysis methods as the algorithm uses the explicitly defined edges and nodes in the CDFG to directly look for standard structures and patterns defined in the section 5.3.2. This reduces the search space of the algorithm and traverses the edges indicated by the rules.

The rule-based method uses a set of hierarchical conditions to describe the rule

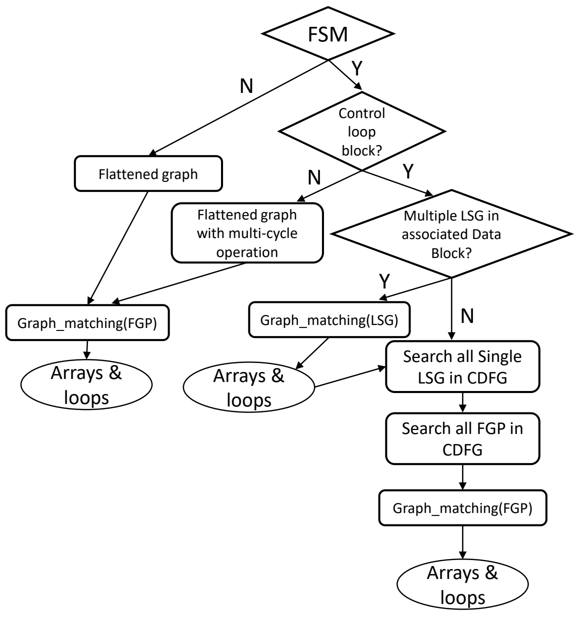


Fig. 5.5: Rule-based flowchart for arrays and loops formation

base, shown in the flowchart in figure 5.5. The flowchart along with Algorithm 2 and graph matching identifies structures of loops and arrays based on a specific set of search conditions.

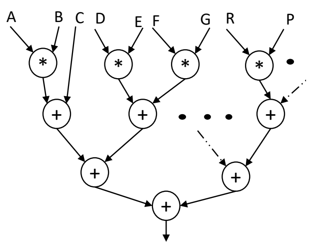
The rule-based flowchart uses state variables (CB_V) as the root condition to identify presence of state registers from FSM modules of the RTL design. These variables enables the method to deduce that the CDFG structure is parallel in nature i.e., fully flattened, having flat graph patterns (FGP) without the presence of any back edge. In order to map these structures, the flowchart with algorithm 2 uses graph matching method i.e., *Graph_ matching(FGP)* (lines 17-22). This algorithm maps the matched graph patterns to loop and array operations. The CDFG is classified as a flattened

Algorithm 2: Algorithm for array and loop detection

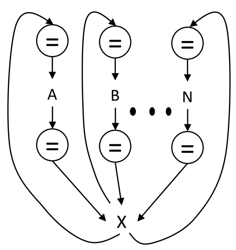
```
Data: $CB_L$ = Loop control block
$FGP[M]$ = Flat graph patterns or Flattened graph
$G_{cdfg}$ = Generated CDFG from Phase 1
Result: $ARR[N]$ = Set of array elements
$L_d[N]$,$CB_L[N]$ = set of Loop sub-graphs with Control loop block
$S[M]$ = Graph patterns to be folded in loop
1  begin
2      if $CB_L$ exists then
3          $LSG[N] \leftarrow$ Multiple loop sub-graphs in same Data Block
4          $L_p[K] \leftarrow$ Graph_matching($LSG[N]$)
5          if Graph_ matching($LSG[N]$) exists then
6              $ARR[i] \leftarrow$ Arrays from matched graphs
7          for $i \leftarrow 1$ to $n$ do
8              if multiple assignments controlled by $CB_L$ and converge to same variable
               then
9                  $ARR[i] \leftarrow$ Array-read operations
10                 if Input of assignments are constant then
11                     $ARR[N] \leftarrow$ array of constants
12             $L_d[i] \leftarrow$ Disjoint loop sub-graphs(LSG[N])
13     if $FGP[M]$ OR not $LSG[N]$ then
14         $L_d[N], CB_L[N] \leftarrow$ Search all single LSG($G_{cdfg}$)
15         if not $LSG[N]$ then
16             $FGP[M] \leftarrow$ Search_Graph_patterns($G_{cdfg}$)
17         $MatchedGraphs \leftarrow$ Graph_matching($FGP[M]$)
18         for $i \leftarrow 1$ to $M$ do
19             for each assignment node in matched graph do
20                 if assignment nodes are parallel then
21                     $ARR[i] \leftarrow$ Inputs of assignments
```

graph with FGP executed in multi cycle operations in the presence of state variables but without loop control blocks.

Algorithm 2 identifies and extracts multiple loop sub-graphs with back edge (LSG) in every data block which are connected from a control loop block. The *Graph_ matching(LSG)* method searches and groups the array operations from matched



(a) Example of Flat graph patterns (FGP) in CDFG



(b) Example of multiple loop sub-graphs (LSG) in CDFG

Fig. 5.6: Example patterns for *Graph matching* algorithm

graphs (lines 4-6), as an example shown in figure 5.6b. The design scenario of multiple assignments controlled by loop control blocks (CB_L) depicted in the CDFG of motivational example in figure 5.3 is identified using the rule-base in Algorithm 2 (lines 8-11). Algorithm 2 exhaustively extracts all possible patterns of assignments and operation node paths from the CDFG which include multiple LSG with their corresponding loop control blocks as well as multiple FGPs (lines 15-17). In this way, all disjoint loops in folded forms can be extracted and mapped to C program, whereas graph patterns if matched, are formed as individual loop structures and array operations. This rule set proves that multiple types of structures can be identified and mapped to behavioral operations of arrays and loops by the proposed method.

Graph Matching: Algorithm 2 also uses algorithm 3, a graph matching method to identify and match graph patterns and map them to loop structures or array operations following specific rules. The method takes as input, two different types of patterns, multiple LSGs and flat graph patterns (FGPs). A header node here is the entry point of a loop sub-graph [20]. In the case of multiple LSGs in a single DB shown in figure 5.6b, the assignment paths and header nodes are matched for each sub-graph being investigated (lines 8-10). Graph patterns as assignment and

Algorithm 3: Graph matching algorithm

```
   Data: $FGP[M]$ = Set of entry assignment nodes
   $LSG[N]$ = Set of loop sub-graphs
   Result: $P[M]$ = Set of similar sub-graphs or graph patterns
   $Li$ = Loop index
1  begin
2      if FGP present then
3          for $j \leftarrow 0$ to $Y$ do
4              Match and group sub-patterns
5              if Child operator matches previous grouped graph then
6                  $P[j] \leftarrow$ Add to matched patterns
7                  $Li \leftarrow$ Number of matched groups
8      for $j \leftarrow 0$ to $Z$ do
9          if assignment paths of $LSG[j]$,$LSG[j+1]$ match then
10             if Headers are similar then
11                 $P[N+j] \leftarrow$ Add to matched patterns
12     return $P[M]$
```

operation node paths are extracted from CDFG typically like in figure 5.6a and taken as input, *FGP* by the algorithm. These patterns are iteratively grouped together as a single graph until the patterns mismatch with the already formed graph (lines 4-7). The degree of matching is also considered as the algorithm uses the matched portion of the structures to extract the patterns and uses this degree to compute the loop index.

Once the CDFG is fully analyzed, the proposed method writes out the synthesizable ANSI-C code. In cases e.g. when the graph is fully flattened with nested loops and conditional blocks, *VeriIntel2C* is unable to find possible loops and arrays. In the other cases, it can find a large number of arrays and loops, thus enabling a wider range of design space and hence should lead to a trade-off curve with many dominating designs.

Table 5.2: RTL Benchmarks Overview

Benchmark	Lines of Code	Arrays		Loops		CC
		2D	1D	Nested	Single	
bnch_enc	226	0	1	5	7	high
dct_JPEG	631	0	25	0	5	high
quantizer_JPEG	845	0	40	0	5	high
aes	174	0	3	3	1	low
mean_filter	207	0	2	0	2	low

5.4 Experiments and Discussion

The experimental setup is now described and experimental results are discussed for evaluating *VeriIntel2C* using qualitative as well as quantitative analysis.

5.4.1 Experimental Setup

In order to fully characterize the effectiveness of our proposed method, three separate set of experiments are conducted. In the first set, four open-source RTL IPs from OpenCores [58] are used to demonstrate the ability of *VeriIntel2C* to enable DSE of hand-coded RTL descriptions, as this is the final goal of this work.

Although the open source designs highlight the robustness of our method, it is also required to measure the ability of our method to identify loop and arrays of different forms in RTL designs and expose full strengths and weaknesses of our proposed method. Therefore, additional experiments are required. The second set of experiments compare the quality of DSE of original C-based descriptions against the C designs generated by *VeriIntel2C* and finally the third set of experiments evaluate our method against a previous work. These 3 experiments are described in detail in the following sections termed as *Experiments 1*, *Experiments 2*, *Experiments 3* .

In order to quantitatively analyze the results of our method, we evaluate the number of loops and arrays which have been abstracted successfully into the resulting

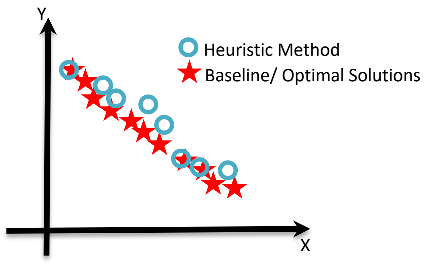


Fig. 5.7: Example of DSE result with a reference Pareto-set

C description from the input RTL designs that are obtained from DSE of original C-based benchmarks. In terms of qualitative analysis, we perform DSE of the resulting C descriptions from *VeriIntel2C*, translated from the various RTL design forms of all the stated benchmarks.

For the purpose of the analysis of the DSE results as multi-objective optimization functions, there are a multitude of unary indicators available, e.g., average distance from reference set(ADRS), hypervolume indicator, cardinality measure, a review of which is provided in [89]. A reference Pareto-set is obtained by combining the Pareto-sets from DSE of D_{small}, D_{med}, D_{fast} and forming the best curve. The approximate Pareto-set is the one to be measured. In figure 5.7, a reference set is obtained by combining the design points in all pareto-fronts being tested, and choosing the designs that most minimize the objectives on X and Y axes. This pareto-set is the most optimal set in a graph and all approximate sets are evaluated against this reference set. In this case, ADRS and Pareto Dominance are used to measure the quality of the methods which are explained as following:

1. *Average distance from reference set(ADRS):* ADRS indicates the average distance between reference pareto front and the approximate pareto set i.e., tells hows close a pareto front is to the reference front. Given a reference Pareto front, $\Gamma = \gamma_1 = (a_1, l_1),\ \gamma_2 = (a_2, l_2), ..., \gamma_n = (a_n, l_n)$ and an approximate pareto set $\Omega = \omega_1 = (a_1, l_1), \omega_2 = (a_2, l_2), ..., \omega_n = (a_n, l_n)$ with $a \in A$ and $l \in L$ where A is the design area and l is the corresponding latency, then,

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

   where

   $f(\gamma = (a_\gamma, l_\gamma), \omega = (a_\omega, l_\omega)) =$

   $$\max \left\{ \left| \frac{a_\omega - a_\gamma}{a_\gamma} \right|, \left| \frac{l_\omega - l_\gamma}{l_\gamma} \right| \right\}$$

   The value of ADRS is inversely proportional to the degree of similarity between two pareto-sets, reference and approximate, being compared. A high ADRS value (%) implies that a significant number of points of the reference pareto-set is missing in the approximate pareto-set. The higher the ADRS is, the lower is the quality of the approximate Pareto-set.

2. *Pareto Dominance*: This index is equal to the ratio between the total number of designs in the Pareto set being evaluated (obtained by executing one exploration method), also present in the reference Pareto set. The reference Pareto set is obtained by combining the best results of each method over 4 runs. The higher is the value, the better is the Pareto set.

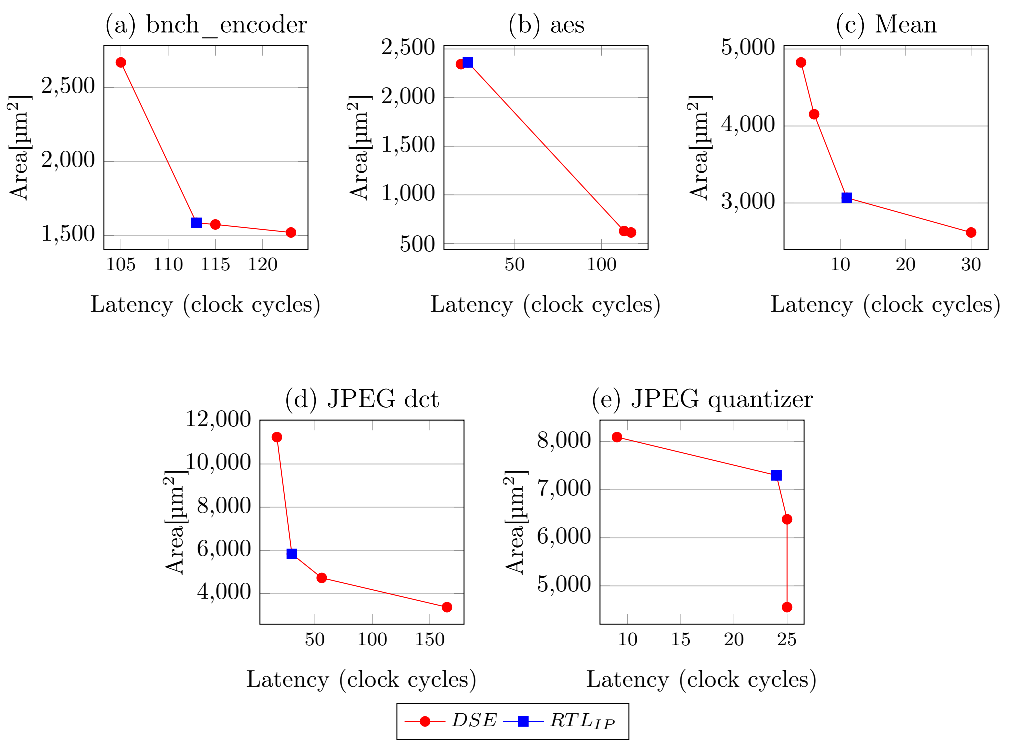


Fig. 5.8: Conversion results for open-source RTL designs.

The experiments were run on an Intel Xeon processor running at frequency of 2.4 GHz, having a RAM of 16GBytes. For the purpose of design space exploration, a commercial HLS tool is used, i.e., CyberWorkBench v5.4, wherein Nangate's open cell (45nm) technology library (set by simulator) is chosen as the target technology. The target frequency for generating RTL descriptions is 100 MHz, with the trade-off objectives restricted to area and latency respectively (set by simulator). The design space explorer used solely for the experiments is based on [47] and is performed on each original benchmark design as well as on the resulting C design respectively. The set of best quality design points chosen by combining all trade-off curves of each benchmark scenario of Fig. 5.9 is computed as the reference Pareto-front. The ADRS

Table 5.3: C Benchmarks Overview

Benchmark	Lines of Code	Arrays		Loops		CC
		2D	1D	Nested	Single	
ave8	42	0	1	0	2	low
fir	23	0	2	0	2	low
sobel	85	3	1	4	1	low
Interp	74	0	4	0	4	medium
disparity	119	1	5	5	2	high

Table 5.4: Experimental Results Summary (Original HLS DSE vs. *VeriIntel2C*)

Benchmark		Converted by *VeriIntel2C*					Original					ADRS	Dom
Name	Ver	Arrays		Loops	ADRS	Dom	Arrays		Loops	ADRS	Dom	diff	diff
		2D	1D		[%]	[%]	2D	1D		[%]	[%]	[%]	[%]
ave8	D_{fast}	0	1	2	20	60	0	1	2	9	40	11	20
	D_{med}	0	1	2	10	20	0	1	2	9	40	1	-20
	D_{small}	0	1	2	6	40	0	1	2	9	40	-3	0
fir	D_{fast}	0	2	2	0.15	83	0	2	2	3	50	-2.85	33
	D_{med}	0	2	2	3.1	33	0	2	2	3	50	0.1	-17
	D_{small}	0	2	2	3.1	16	0	2	2	3	50	0.1	-34
sobel	D_{fast}	3	1	5	18	20	3	1	5	22	40	-4	-20
	D_{med}	0	4	4	7.7	40	3	1	5	22	40	-14.3	0
	D_{small}	1	1	5	22	20	3	1	5	22	40	0	-20
interp	D_{fast}	0	4	4	7	20	0	4	4	7.4	30	1.6	-10
	D_{med}	0	4	4	4.1	50	0	4	4	7.4	30	-3.3	20
	D_{small}	0	4	4	6.5	30	0	4	4	7.4	30	-0.9	0
disp	D_{fast}	1	5	7	9.3	69	1	5	7	15	15	-5.7	54
	D_{med}	1	5	7	25	7.6	1	5	7	15	15	10	-7.4
	D_{small}	1	5	6	19	7.6	1	5	7	15	15	4	-7.4
Avg.					**10.7**	**34.4**				**11.3**	**35**	**-0.416**	**-0.58**

and Pareto-dominance for every trade-off curve under evaluation, is calculated against this reference front.

The three set of results should fully validate our proposed method. The following sections describe the different sets of experiments and explain their results, namely *Experiments 1*, *Experiments 2*, *Experiments 3*.

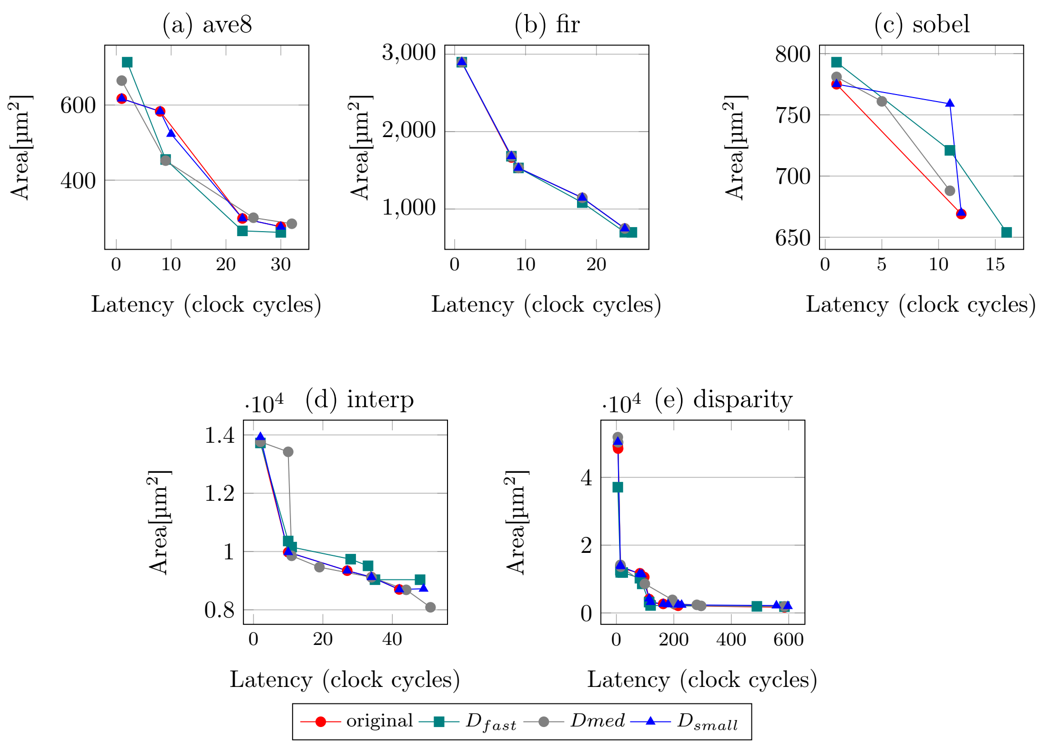


Fig. 5.9: Design Space Exploration trade-off curve results using S2CBench designs (Original HLS DSE vs. *VeriIntel2C*)

5.4.2 Experimental Results

This subsection describes the experimental results of the three sets of experiments described previously.

Experiments 1: Hand-coded RTL to C using VeriIntel2C

In the first set, five different RTL IPs from OpenCores [58] are used to verify the ability of *VeriIntel2C* to generate loops and arrays from hand-coded RTL descriptions, as this is the final goal of this work. The designs used are BNCH encoder, AES encryption, DCT and Quantizer components of JPEG encoder design and a Mean filter, where each design has distinct program style. Our method generates a synthesizable C

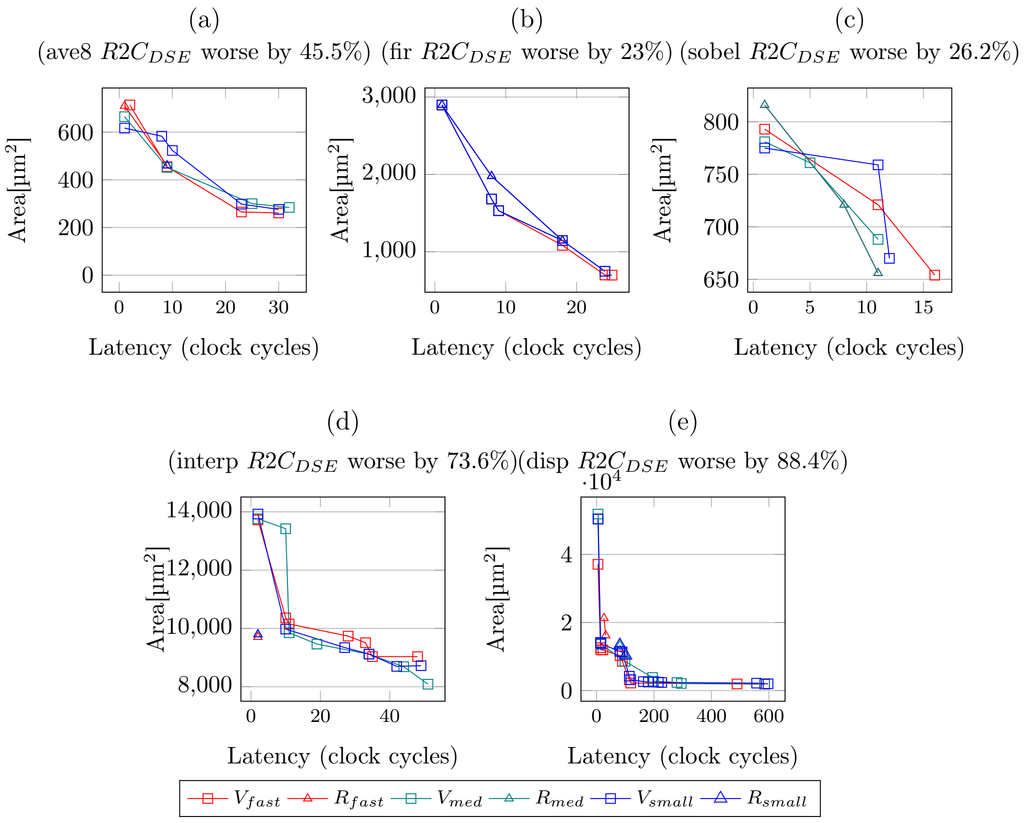


Fig. 5.10: Design Space Exploration trade-off curve results for S2CBench designs (*VeriIntel2C* vs. *R2C*[9]).

program for each RTL design. The complexity of the RTL designs are provided in Table 5.2. *CC* refers to the conditional complexity of a design. The number of loops and arrays for the RTL designs correspond to the number found upon translation to C design using our method. The reason to select these designs is because of their varying complexity and that the designs have loop and array constructs. *VeriIntel2C* however has some limitations in the type of RTL designs, that it cannot handle designs with pipelined loops, and partially folded loop forms with conditional constructs.

Fig. 5.8 shows the trade-off curves obtained upon performing DSE of the resulting

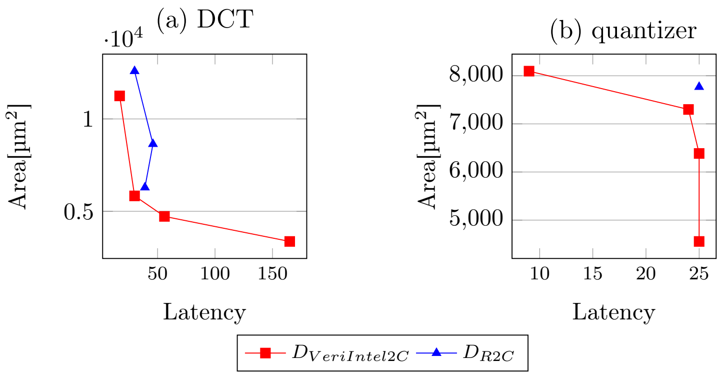


Fig. 5.11: DSE results of open-source RTL JPEG encoder components (*VeriIntel2C* vs. *R2C*).

C codes generated by our method (indicated by red as *DSE*) against the single, hand-coded RTL designs (indicated by a single blue square as RTL_{IP}). The graphs in Fig. 5.8 prove that our proposed conversion method enables DSE of manually written RTL designs with diverse structural styles.

Experiments 2: VeriIntel2C vs. HLS DSE

The second set of experiments begins with a set of C-based designs taken from the open-source Synthesizable SystemC benchmarks suite (*S2CBench*) [76]. Table 5.3 gives an overview of the complexity of these benchmarks in terms of their number of lines of code, arrays (one and two dimensional) and loops (nested and single). Here, CC refers to the conditional complexity of the benchmarks inside the loops where, 'low' means the code is mainly data-path centric and 'high' indicates control-path centric. Conditional complexity indicates the number of loops in the design that has conditional constructs, and the level of complexity of these constructs inside the loops. If a nested loop contains conditional constructs, then the *CC* is high. The design space of these benchmarks varied from a minimum of 36 design points to a maximum of 1728. The most important thing to note is their varying complexity of the benchmarks that is reflected in *VeriIntel2C*'s evaluation analysis. The quality of the generated C code

Table 5.5: Experimental Results comparison of *VeriIntel2C* and *R2C*

Benchmark		Converted by *VeriIntel2C*					*R2C*					ADRS diff	Dom diff
Name	Ver	Arrays		Loops	ADRS	Dom	Arrays		Loops	ADRS	Dom		
		2D	1D		[%]	[%]	2D	1D		[%]	[%]	[%]	[%]
ave8	D_{fast}	0	1	2	12	75	0	1	0	37	25	-25	50
	D_{med}	0	1	2	10	100	0	0	0	NA	NA	NA	NA
	D_{small}	0	1	2	5	80	0	1	0	35	20	-30	60
fir	D_{fast}	0	2	2	0	83.3	0	2	1	29	16.6	-29	66.7
	D_{med}	0	2	2	0	60	0	2	1	20	40	-20	20
	D_{small}	0	2	2	0	60	0	2	1	20	40	-20	20
sobel	D_{fast}	3	1	5	9	50	3	1	1	8.2	50	0.8	0
	D_{med}	3	1	5	16	50	3	1	1	16	50	0	0
	D_{small}	3	1	5	0	100	1	1	1	78	0	-78	100
interp	D_{fast}	0	4	4	8	80	0	4	0	74	20	-66	60
	D_{med}	0	4	4	8	80	0	3	0	96	20	-88	60
	D_{small}	0	4	4	8	80	0	3	0	75	20	-67	60
disp	D_{fast}	1	5	7	23	80	1	2	0	93	20	-70	60
	D_{med}	1	5	7	1	100	1	5	0	99	0	-98	100
	D_{small}	1	5	7	0.7	100	0	3	0	98	0	-97.3	100
JPEG	*DCT*	0	25	5	0	100	0	12	3	63	0	-63	100
	quantizer	0	40	5	0	100	0	24	1	26	0	-26	100
Avg.					**6.71**	**78.5**				**54.2**	**20.1**	**-48.5**	**59.79**

using *VeriIntel2C* depends on the number of loops and arrays generated, since these *explorable* constructs aim to increase the design space of a single RTL design. In our experimental setup, to evaluate this, we first used a previously developed HLS DSE explorer [47] to obtain sets of pareto-optimal RTL micro-architectures for each of the above mentioned 5 SystemC benchmarks. Each of the pareto-optimal RTL designs have different characteristics e.g., arrays implemented as registers, wires, memory blocks or loop structures unrolled (flattened), fully or partially folded in different ways.

From each of these sets obtained from the DSE of 5 systemC benchmarks, 3 different RTL micro-architectures (D_{fast}, D_{med} and D_{small}) were chosen to test our proposed flow. Figure 5.12 shows a pareto-set that is generated for each SystemC

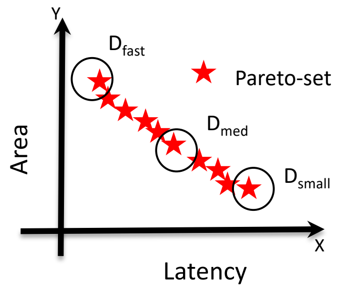


Fig. 5.12: Setup for *Experiments 2*

benchmark. In this trade-off curve, a design that has minimum latency is chosen as D_{fast} is selected. The design that has smallest area is chosen as D_{small}, and a random design is chosen from the middle, that minimizes the cost function as D_{med}. Thus using 3 different RTL variants for all 5 benchmarks, a total of 15 unique RTL designs were used for testing.

Each of the RTL micro-architecture selected has a distinct design structure, *i.e.* fastest design D_{fast} is the fastest design (lowest latency) but highest area, smallest D_{small} has the lowest area but highest latency and intermediate, D_{med} is an intermediate design. Our work uses area and latency as the design trade-off metrics, where the latency is given in clock cycles.

The quantitative and qualitative results reported in Table 5.4 were obtained by running the DSE explorer for every original C-based benchmark as well as every abstracted C variant design translated by *VeriIntel2C*. In Table 5.4, the ADRS and dominance *diff* columns indicate the difference between the original benchmarks' values and that of the converted variants.

Table 5.4 evaluates the ADRS and dominance values of all the DSE curves, where the ADRS and dominance is obtained by comparing the trade-off curves obtained by each of the methods against a reference front obtained by combining all of the results of both methods. As mentioned earlier, a decreased ADRS value indicates better quality of DSE. Thus, a negative value in the ADRS *diff* column in Table 5.4 implies that the quality of DSE curve generated by Converted design is better than that by the Original design and vice-versa. Also, an increased Dominance value indicates better quality of DSE *i.e.*, a positive value in Dominance *diff* column implies that the DSE quality of the Converted design is better than that of the Original design.

On average, using *VeriIntel2C*, the ADRS improved by reducing by 0.41%. Moreover, the dominance reduces by merely 0.58%. From the results, it can be observed that our proposed method works well as it overall maintains the similar quality to that of the original HLS DSE.

Fig. 5.9 provides the graphical representations of the set of dominating design points for every benchmark along with that of their corresponding design variants with respect to design trade-offs of area and latency.

Table 5.4 also shows that for most of the designs *e.g.fir*, *ave8*, *interp* and *disparity*, *VeriIntel2C* is able to re-generate all the loops and arrays that were present in the original benchmark (original column of Table 5.4). The results also show that our proposed method can re-generate different types of arrays including multi-dimensional arrays implemented in varying forms, e.g. dedicated logic, memories or registers.

One interesting observation happens in the *sobel* benchmark case, where the resulting curve of D_{fast} using *VeriIntel2C* has a design point with higher latency, but smaller area, than that of the original DSE curve. This is because *VeriIntel2C* generates if-else conditional expressions inside the main inner nested loop. As a result, the micro-architecture generated has a higher latency, but lower area. Thus, it can be said that the method expands the existing design space by adding more design points on the reference Pareto-front. The results obtained from D_{med} and D_{small} were able to re-produce all the loops with one-dimensional arrays rather than 2-dimensional arrays without any conditional statements inside the loop (*converted* column in Table 5.4). Thus, their DSE produces similar trade-off curves compared to that of the original one. As a result in Table 5.4, the ADRS % for D_{med} in converted column is lower than that of the original column.

As observed in Table 5.4, almost the entire search space is very well replicated, as compared with the average ADRS of the original designs (11.3%) being similar to that of the average ADRS of the converted designs (10.7%). This indicates that *VeriIntel2C* can very well re-generate and conserve the quality of the DSE of the original benchmarks even if their design styles are varied.

Experiments 3: VeriIntel2C vs. R2C[9]

Finally, the very last set of experiments compares *VeriIntel2C* with a previous work that is most closely related to this work, called *R2C* [9]. For this purpose, *R2C* was reproduced. The D_{small}, D_{med}, D_{fast} RTL variants of the same benchmarks used in *Experiments 2* were passed as input to both *R2C* and *VeriIntel2C*. The resulting C designs from *R2C* are denoted as R_{small}, R_{med}, R_{fast} while that from *VeriIntel2C* as V_{small}, V_{med}, V_{fast}. ADRS and Dominance of the final trade-off curves obtained after DSE of the resulting C codes is used to measure the quality of the conversion. Finally, DCT and quantizer, i.e., Verilog components of a JPEG encoder taken from

Opencores[58] are used in this experiment to fully characterize both methods.

Table 5.5 tabulates the qualitative and quantitative comparison results of both the methods *VeriIntel2C* and *R2C*, that can also be graphically correlated with the exploration trade-off curves of this experiment in Fig. 5.10. Similar to the previous results, the quality of the exploration for each of the curves in Fig. 5.10 is measured against the reference pareto-front composed by combining the overall dominating results of both methods. Fig. 5.10 corroborates with the Table 5.5. Here, *disparity*, *interp* and *ave8*, *R2C* cases produces few of the arrays but does not generate any loops, thus leading to higher ADRS and lower dominance % values i.e. worse in quality.

From Table 5.5, the averaged ADRS of these cases worsen by 88%, 73%, 27% respectively. Moreover, in the case of *ave8*, the D_{med} variant, *R2C* does not produce any loops or arrays making any qualitative analysis non-applicable (*R2C* column in Table 5.5). In the case of *fir*, the *R2C* method produces 1 of the 2 loops that leads to a much worse design space. This results in the average ADRS of *fir* being worse by 27% for *R2C*.

It is also shown in Fig. 5.9c *sobel* that most of the points in the reference front is produced by *VeriIntel2C* since the dominance improves on average by 33%.

With respect to the comparison of *R2C* against our proposed method for the hand-coded Verilog designs (DCT and quantizer), Table 5.5 (last two rows) shows that our proposed method also outperforms R2C. Here, our proposed method generates trade-off curves with ADRS lowered by 63% and 26% (negative indicating improved value) for the DCT and quantizer designs respectively. For example in Fig. **??**, the reason for *R2C* to be able to generate a trade-off curve is because the Verilog design has *for-generate* syntax, thus *R2C* identifies only those loops with that syntax and arrays associated with them. However, the quality of the DSE is worse than that of *VeriIntel2C* as stated above.

In summary, previous work generates trade-off curves that are on average, worse in quality compared to that generated by our method (ADRS improved by 48.5%, dominance by 59.79%) . Moreover, our proposed method is able to generate trade-off curves that almost replicate the quality of DSE of original designs with ADRS being better by 0.41% and dominance being worse by only 0.58%. This clearly proves that our proposed method leads to a better quality design space, with a higher number of micro-architectures to choose from, thus, expanding the design space. Based on these results, we can therefore conclude that *VeriIntel2C* works well and it is able to generate arrays and loops from different RTL styles.

[0]This work has been accepted for publication in Integration the VLSI journal, 2018.

Chapter 6

Optimization of RTL to C Abstraction to Maximize Design Space Exploration of RTL designs

6.1 Introduction

As discussed in the previous chapter, a translation method called *VeriIntel2C* is proposed and described. This method is able to abstract a RTL design into C design and leverage HLS to perform DSE and enlarge its design space.

This work aims to expand the above mentioned design search space, by proposing an optimization method from the CDFG generated by *VeriIntel2C* based on feature-extraction. The proposed algorithm determines the formation of the multi-dimensional arrays by obtaining the various characteristics of the existing array forms and their respective read/write operations in associated loop structures in the CDFG. Generating multi-dimensional arrays helps expanding the search space as modern HLS tools allow to either *fuse* them into a single array, *expand* them into separate arrays, or *expand* different dimensions of the array. The loop fusion and array merging techniques are classic concepts of compiler design [1]. They have mostly been implemented so far in commercial HLS tools as these techniques are implementable upon CDFG of sequential programs (C/C++). Sequential languages are high in the abstraction level. The inputs to HLS tools are C/C++/SystemC languages that are sequential in nature. Thus, the HLS tools may use merging techniques to improve latency overheads or enhance parallel processing, leading to circuit area, latency or throughput improvements. However, in the current industry, the RTL abstraction level cannot take direct advantage of these techniques as the behaviour of RTL is generally concurrent as opposed to that of the HLS languages. Thus, in this case, *VeriIntel2C* is put to use as it extracts a CDFG from the RTL design and creates the sequential flow of operations of the design written in RTL. Custom optimizations which are based on the classic merging techniques can then be implemented on the CDFG in order to obtain a much improved C design which when synthesized directly in hardware would lead to reduced area/latency overheads.

In particular this chapter makes the following contributions:

- Proposes a feature- based extraction method to perform array merging to extend the search space of behavioral descriptions obtained from synthesizable RTL descriptions.

- Uses a rule-based method along with the features extracted, to merge existing loop structures of a C program generated from a previously developed RTL to C translator.

- Performance based experimental analysis to compare the impact of these optimizations on the resultant trade-off curve compared to that of the state-of-the-art.

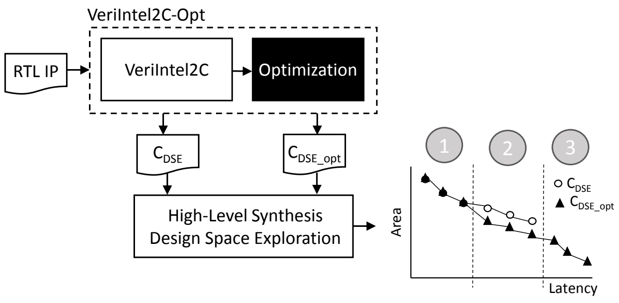


Fig. 6.1: Complete flow overview of overall system

The figure 6.1 explains the overall flow of the proposed system. Based on a previously developed translation system from RTL to C,the system takes a single RTL IP as input and uses *VeriIntel2C* to translate into C program(C_{DSE}). In this thesis, we develop a further layer of *optimization* called *VeriIntel2C-Opt* as shown in figure 6.1 and subsequently translate into a synthesizable C program(C_{DSE_opt}). The HLS Design Space exploration, previously in-house developed in [47], is used as a experiment platform to measure the quality of the generated C design against the C program generated without using *optimization*. The resulting trade-off curves as shown in figure 6.1 indicate that using the optimization layer expands the existing design search space indicated in fig 6.1, by introducing additional micro-architectures, some of which are better in quality. The entire design space is divided into 3 phases in fig 6.1. Phase 1 has designs that minimize area, Phase 2 that minimize the cost function, and Phase 3 for minimizing the latency. Merging loops and arrays, in general, gives rise to more designs with increased latency thus, extending the curve in phase 2 and 3. The resulting trade-off curves as shown in figure 6.1 indicate that using the optimization layer leads to either the same results are the previously generated C code (phase 1), improves the quality of the trade-off curve (phase 2) and/or extends the search space by introducing additional micro-architectures (phase 3).

6.2 Motivation

Upon exploration of the generated C design from *VeriIntel2C*, it was observed for certain test-cases, that the pareto-front of the variant is better in quality than that of the original design. This occurred in the designs wherein *VeriIntel2C* is able to unroll multi-dimensional arrays into series of one-dimensional(1D) arrays which upon exploration, expands the design space by adding more designs to the reference front. Figure 6.2 shows the multi-dimensional NxN sized array(left) being split into series of inter-connected one-dimensional N-size arrays during the formation of C design in *VeriIntel2C*. The framework's algorithm interprets the structure in the generated CDFG as a series of 1D arrays and creates the C design accordingly. This observation formed as a base for our motivation to experiment several 1D arrays together and perform exploration on this modified design. Thus, upon exploration of this modified design using HLS as a black box, commercial tools perform automatic splitting of multi-dimensional arrays with varying dimensions in each iteration of creating different micro-architectures during the exploration. This would lead to generation of several new micro-architectures of better quality than that of the exploration of designs generated by *VeriIntel2C*. This experiment led to the formation of optimization techniques discussed in this chapter. The overall reference front for every exploration is computed by collecting and selecting the pareto-optimal designs amongst all the design points from the exploration of the four designs in the search space. From this observation, it leads to our hypothesis, that by merging existing

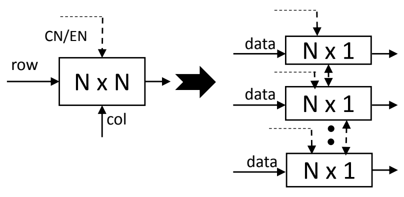


Fig. 6.2: Graphical view of array unrolling by *VeriIntel2C*

arrays of the resulting design together, the optimization creates more designs which are smaller in area but much slower.

6.3 Related Work

There have been some works in the past related to translation of RTL to C/C++ for different objectives. Bombieri et al., [9] focus on creating synthesizable C++ models from RTL optimized for HLS that in turn can be explored using a previously developed DSE, which is closely related to the preliminary work. The shortcomings of this work are that its loop rolling method relies on the fact that the input Verilog/VHDL design must contain syntax of *for generate* statements, ignoring other representations of loops or fully unrolled loops which may not be explicitly instantiated in the RTL design. In another work, [10] transforms a RTL IP model into C++ by extracting and generating Extended Finite State Machines(EFSM). Namballa et al., [69] develop a tool for automatic CDFG extraction from behavioral level VHDL descriptions, but, however, they do not focus on the optimization of the extracted CDFG for enhancing DSE for HLS. Also, the tool depends on the syntax of the VHDL keywords for creation of the CDFG. Commercial HLS tools, mostly provide options for optimizing the behavioral designs using *loop merging* and *array merging.*

6.4 Methodology

6.4.1 Feature Extraction

The methodology for both the stated optimizations, begin with the feature extraction from the generated CDFG followed by the optimization methods. The feature

Algorithm 4: Algorithm for merging arrays

```
Data: $CDFG_{in} = Graph_{array}, Graph_{CL}, Graph_{DL}$
Result: $CDFG_{out} = Graph_{ArrMerged}, Graph_{CLm}, Graph_{DLm}$
begin
    $A_{2Dn} \leftarrow Search2DArrays(CDFG_{in})$
    $A_{1Dn} \leftarrow Search1DArrays(CDFG_{in})$
    if $A_{2Di}$ exists then
        $A_{base} \leftarrow Search\_Largest\_Array(A_{2Di})$
        if $A_{base}$ is homogeneous then
            for $A_{1Di} \leftarrow A_{1D1}$ to $A_{1Dn}$ do
                if $A_{1Di}Size == A_{base}row\_size$ then
                    $A_{base}[row\_size + 1] = A_{1Di}[Size]$
                $A_{1Di} \leftarrow A_{1Di+1}$
        else
            Only if all $A_{1Di}$ is homogeneous
            total_elements $= [(row\_size * col\_size) + ... + (row\_size * col\_size)]_{Abase} + [index_1 + ... + index_n]_{A1D}$
            $(X_1, Y_1), .., (X_n) \leftarrow SearchPairOfFactors(total\_elements)$
            $(X, Y) \leftarrow$ Search if $X$ or $Y$ is same as $index_{A1D}$
            if $X == rowsize_{Abase}$ then
                $A_{base}[rowsize_{Abase} + 1] = A_{1D}$
    else if $A_{1Di}$ only exists and $A_{1Di}$ Homogeneous then
        total_elements $= [index_{1D_1} + .. + index_{1D_n}]$
        X = Number of $A_{1Di}$
        Y = total_elements
        $A_{2D} = New_{arr2D}[X][Y]$
        $New_{arr2D}[X][Y] \leftarrow$ Replace $A_{1Di}$
        Modify_loop( $A_{1Di}$ , $New_{arr2D}[X][Y]$);

Modify_loop($A_{1Di}$ , $New_{arr2D}[X][Y]$) begin
    $L_i \leftarrow Search\_loop\_structures(A_{1Di})$
    $[Ep_1, Ep_2, .., Ep_N] \leftarrow Search\_ExitPoint\_of\_loop$
    if $Ep_i$ is connected to $A_{1Di}$ then
        if $CL_{index}$ of $L_i$ == X or Y of $New_{arr2D}[X][Y]$ then
            $Z \leftarrow Search\_position\_inLoop(A_{1Di}, New_{arr2D}[X][Y], L_i)$
            Z is fixed position of $A_{1Di}$ in $New_{arr2D}[X][Y]$
        else
            Insert outer or inner loop of $New_{arr2D}[X][Y]$
```

extraction method extracts the different characteristics of the CDFG, as requirements for merging arrays and loops, and enumerates them in the form of features. The proposed algorithms are applied on the CDFGA generated from Verilog to be converted [48]. The features can be enumerated as follows:

1. A vector representing the list and number of the one-dimensional arrays which are not read-only and not constant, where $L_{nd} = A[]_1, A[]_2, \cdots, A[]_i, N_{nd} >$ where N_{nd} is the cumulative number of dimensions from $A[]_1$ to $A[]_i$

2. A vector representing the list, number of the multi-dimensional arrays which are of the highest dimension in the C design along with the highest dimension among

them, where $L_{hd} =< A[][]_1, A[][]_2, \cdots, A[][]_i, N_{hd}, D_H >$ where N_{hd} is cumulative number of dimensions in the first set and D_H is the highest dimension

3. Number of the read-only accesses of the arrays from the loop structures in a single iteration, N_{re}

4. Number of the write-only accesses of the arrays from the loop structures in a single iteration, N_{we}

5. A vector consisting of the number of simultaneous read and write accesses of the existing arrays in a single operation along with the respective arrays and the respective operation label, where $L_{srw} =< N_{sr}, N_{sw}, A_n, OP_n >$ is the list of information for A_n, OP_n the number of operations

6. A vector representing the basic characteristics of every loop, $Feature_{loop} = [CL_{index}, OP, N_{exit}]$, where CL_{index} is the control index of the loop, N_{exit} is the exit node with the output data, and OP are all the operations executed inside the loop sub-graph

The above features are collected and used as inputs to the two optimization algorithms discussed in Algorithm 4 and Algorithm 5.

6.4.2 Array merging

The effect of array merging optimization on the existing CDFG impacts the operations and assignments in the program which read or write onto the arrays. Thus, this method must serve two objectives, first arrange the array structures into a single structure, and as a result of that, the second is modifying the operations accessing these arrays inside loop sub-graphs such that the existing functionality of the program is not altered.

The method to merge arrays begins with the extraction of the above described features(1-5), from the CDFG. These features are then used upon two principle conditions to compute the dimension of the merged array. If the arrays from the features are of varying dimensions, then the highest dimension amongst them is selected, otherwise, in other cases, a new array is created wherein the cumulative dimension may be calculated using the following formula:

$$\begin{aligned} X &= X_{in} + N_{ndX}, \\ Y &= Y_{in} + N_{ndY}. \end{aligned} \tag{6.1}$$

The rule-based algorithm shown in algorithm 4, describes the rule base for all the stated conditions. The term 'homogenous' refers to the condition that 4 size of the 1D arrays has to match either the length of the rows or columns of the existing 2D array. The function *Search_ Largest_ Array* searches for the array with the highest dimension amongst existing 2-D arrays, during the former condition(lines 5-10). In the case where dimensions of the existing 1-D arrays are homogenous, having existing 2-D arrays(lines 12-15), the row and column indices can be computed using following steps:

Step 1: The indices of the existing 1D arrays are added together and all the possible factor pairs of the sum is produced.

Step 2: The factor pair is selected based on if one of the elements of the pair match the majority of the 1D array indices.

If there are no existing 2-D arrays found(lines 18-24), then the cumulative row

Algorithm 5: Algorithm for merging loops

```
Data: $CDFG_{in} = Features_{loop}$[N]
Result: $CDFG_{out} = Graph_{CLm}$,$Graph_{DLm}$
1  begin
2      $CL_{index}$[N] = Loop indices from $Features_{loop}$[N]
3      for $CL_{index}[i] \leftarrow CL_{index}[1]$ to $CL_{index}[N]$ do
4          if $CL_{index}[i] >= CL_{index}[i+1]$ then
5              if $CL_{index}[i+1] < CL_{index}[i]$ then
6                  Insert condition with operation of $L_{i+1}$
7              Merge the loops
8              if $L_i$ and $L_{i+1}$ are independent then
9                  Merge the loops
10             else
11                 if result of $L_i$ is not updated by past iterations then
12                     Conserve order and Merge loops
13         $CL_{index}[i] \leftarrow CL_{index}$[i+1]
```

index of the newly formed array is the sum of the indices of all the existing arrays and the column index being the number of existing arrays. The second objective of modifying the loop operations is implemented using the *Modify_ loop* function in Algorithm 4. In this function, it is also checked if the control loop indices of the graph correspond to the size of the existing 1D arrays(A_{1Di}) and obtains the row or column size of the merged array,New_{arr2D} which must be equal to the control loop value(lines 31-34). In this case, the existing loop is transformed into a nested loop structure. In this way, the algorithm handles both the conditions which are inter-dependent on each other.

6.4.3 Merging of loops

Merging of loop sub-graphs with similar structures creates a more complex sub-graph by increasing the cumulative operations inside the loop, but allows the HLS process to optimize these reducing the total latency and area, as often HLS tools do not perform optimizations across loops. Loop fusion, thus, in some cases, can lead to better results than the original behavioral description. For implementing this optimization, feature 6 is used to extract the characteristics of each loop.

Algorithm 5 describes the flow of the rule-base used to merge loops using these features. Here, CL_{index} from feature 6, is the control loop index of every loop and N_{exit} is the set of exit nodes of each loop, which is used to check for data dependencies between the resulting outputs of the different loops. OP contains the set of operations, described in CDFG form, which is used to search if the overall output of the loop is dependent on the operation in the past iterations of the same loop. Algorithm 5 uses these features with their associated conditions in a sequential manner, to make modifications inside the selected loop structure to accommodate the combined operations. In the event of unequal loop indices(lines 4-7), the loop with the lower index is merged inside the loop with the larger index, using a conditional construct.

The output of Algorithm 5 is $Graph_{CLm}$ which is the control loop for the merged structure and $Graph_{DLm}$ is the merged data-flow loop sub-graph. In the event of a loop's total iterations($CL_{index}[i]$) lesser than that of another loop($CL_{index}[j]$), L_i is executed for CL_{index} iterations, with their associated OP preserving the OP of L_j in the same loop at the same time.A new condition is inserted for controlling the execution of operations of L_i only until its respective CL_{index}, and their associated OP are inserted under that condition, preserving the OP of L_j in the same loop at the same time.
The purpose of inserting the condition for searching if the output of L_i is dependent on and updated by past iterations(lines 11-12), in the event of L_i and L_{i+1} being

data dependent, is to make sure that the values would get over-written wrongly by the former loops and functionality would get tampered. The statement, *Conserve order and merge loops* preserves the sequential order of the loop operations before inserting the condition accordingly to merge the loop with lower $CL_{index}[i]$. Lines 11-12 in algorithm 5 describe the scenarios where data dependencies exist between loops but the updating of the variables used in the dependent loop is not affected by past iterations of the primary loop. Thus, it necessitates the requirement of preserving the order of operations of both loops as identified in the CDFG.
Thus, in this way, the two optimization layers, consolidated into one as *VeriIntel2C-opt* is implemented upon the CDFG generated from the basic *VeriIntel2C*.

6.5 Experimental Results

6.5.1 Experimental Setup

In order to effectively test our proposed optimization method over the previously developed framework *VeriIntel2C*, a qualitative analysis is performed by comparing the quality of the Pareto-optimal trade-off curves of the C designs generated by *VeriIntel2C-Opt* and *VeriIntel2C*.

Three different RTL designs were generated using benchmarks from open-source Synthesizable SystemC benchmarks suite (S2CBench) [76]. These benchmarks were initially *explored* using a previously developed DSE[47] to obtain a pareto-optimal set of alternative RTL micro-architectures, amongst which the fastest RTL design (D_f) and the smallest(D_s) were chosen. Another three hand-coded RTL designs were selected from open-source domain[58] having varying complexity and characteristics. The first set are designs with alternative variant micro-architectures which tests the

Table 6.1: Benchmarks Overview

Benchmark	Lines of Code	Arrays		Loops		level
		2D	1D	Nested	Single	
IIR filter	30	0	1	0	2	low
Interp	74	0	4	0	4	medium
sobel	85	3	1	4	1	medium
disparity	119	1	5	5	2	high
encoder	140	0	1	3	7	high
decimation	332	0	10	0	15	high

ability of our proposed method to optimize the C design irrespective of the design style of RTL description. The second set of open-source hand coded RTL designs test the ability of *VeriIntel2C-opt* to perform optimization on open-source fixed architecture RTL IP's written manually by designers. Table 6.1 provides overview of the benchmarks used where *comp* is the complexity level.

The ADRS and pareto-dominance(*dom*) was used to measure the quality of the pareto-optimal sets from both methods [89]. The ADRS value 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. Pareto-dominance indicates the ratio of the total number of designs in the pareto-set being evaluated, also present in the reference pareto-set. Column 1 in table 6.2 indicates the designs where, column *ver* shows the type of RTL variants for every SystemC benchmark whose variant RTL micro-architectures upon being translated using *VeriIntel2C* has produced diverse C descriptions . The columns of *VeriIntel2C* and *VeriIntel2C-Opt* consist of the respective ADRS and dominance values.

Table 6.2: Experimental Results Summary

Benchmark		*VeriIntel2C*		*VeriIntel2C-opt*		ADRS	*Dom*
name	version	ADRS [%]	*Dom* [%]	ADRS [%]	*Dom* [%]	diff [%]	diff [%]
disparity	D_f	9.1	63	1.5	90	-7.6	27
disparity	D_s	6.7	21	1	78	-5.7	57
decimation	RTL IP	21	25	7.2	75	-13.8	50
interp	D_f	2.7	50	3.5	50	0.8	0
interp	D_s	1.4	60	4.2	40	2.8	-20
encoder	RTL IP	41	25	3.5	75	-37.5	50
sobel	D_s	26.1	29	0	86	-26.1	57
IIR filter	RTL IP	21.4	60	7.8	40	-13.6	-20
Avg.		**16**	**41**	**3.5**	**66**	**-12.58**	**25**

6.5.2 Results and Discussion

Table 6.2 compares the quality of the trade-off curves generated by *VeriIntel2C-opt* over the basic framework, *VeriIntel2C*. The ADRS values of the exploration trade-off curve obtained by design using our proposed method overall improve by 12.5% which indicate that our proposed method generate better quality pareto-optimal designs compared to that of the original results. The average decrease in ADRS indicates it generates a Pareto-front better in quality than that of *VeriIntel2C*, since lower the ADRS, the better is the quality. The increase in dominance values suggest the same improvement in quality as higher the dominance is, the better is the quality. The dominance values overall increase as well by 25% thereby indicating an increased number of designs in the resulting trade-off exploration curve. From the observations in table 6.2, in the case of interpolation design (*interp*), the merging of the loops does not affect the quality of the trade-off curve significantly. The reason is because the loops used for merging were independent of each other and had equal number of iterations. In contrast, the *disparity* design for both its variants, upon merging two of the loops having unequal iterations, significantly improves the quality of the pareto set and introduces more number of designs in the design search space, similarly for other designs respectively. The *disparity* design's results indicate that new designs were introduced in phase 3 of the design space as loop merging was performed by *VeriIntel2C-opt*. It should be noted that our proposed optimization method *VeriIntel2C-opt* does only work in designs that contain memories or registers blocks which can be converted to arrays and iterative structures in the CDFG, thus for some benchmarks without these constructs, the results of our proposed method is the same as the *VeriIntel2C*.

[0]This work has been submitted for review in IEEE Embedded System letters, 2018.

Chapter 7

Application of RTL to C abstraction methodology to Accelerate System-level simulations

The need for shortening verification time of SoC in order to reduce the overall design time, leads to identifying some important bottlenecks in verification of accelerators. We introduce some of the problems faced in this area, and propose methods using *VeriIntel2C* to solve some of the problems and enable system-level exploration of a SoC.

7.1 Introduction

VLSI circuits are reaching complexities never seen before. Most circuits are now heterogeneous Multi-Processor System-on-Chips (MPSoCs), which typically include embedded micro-processors, memory controllers, memories and dedicated hardware accelerators (HWAccs), all interconnected through a single bus or bus-hierarchies.

The problem that arises while designing these complex Integrated Circuits (ICs) is the complexity to determine overall system architecture. For this purpose, system designers tend to use fast transaction level models (TLM) based on high-level languages such as SystemC, which allows modeling of concurrent processes. Once the overall system structure has been fixed, the model needs to be refined by using more accurate models, typically cycle-accurate models. These models allow to fine-tune the system's performance by e.g. matching the Data Initiation Interval (DII) of certain dedicated hardware accelerators (HWaccs) with the memories' DII and/or the bus bandwidth. For large systems, this can prove inviable or take extremely long execution time.

However, this strategy has several key advantages: Firstly, the workload pattern of the entire system is preserved (considering that the master is not using the returned data for control actions). Secondly, the compile time of the entire model is accelerated, as the complexity of these behavioral templates (BT) is much lower than that of the actual IPs (it should be noted that for larger circuits the compilation time can be significant). Thirdly, the cycle-accurate simulation is much faster as each template does not require to perform any actual computation. Lastly, it allows the exploration of configuration of any latencies, hence it is very easy to generate different *what-if* scenarios.

The main problem is that many IPs often have variable execution latency. Figure 7.1 shows an example of a FIR filter, which computes the sum of products (*sop*) of the given data and coefficients for a certain number of taps (N). In this case, if the *sop* reaches a maximum saturation value, the sop computation is terminated. The latency of the synthesized design is therefore data dependent. In the worst case, the loop is iterated N times, while in the best case, it only executes the loop a single time, i.e, $L = [1, N]$. If this IP is substituted by a behavioral template, in order to maintain the accuracy, conditional constructs ($COND$) such as *if-else*, *break*, *continue* that affect the latency need to be preserved.

Thus, new methods to speed-up these cycle-accurate simulations are needed. The main contributions of this chapter can be summarized as follows:

- Developed an extended feature using the translation framework of *VeriIntel2C* to identify different conditional scenarios in RTL IPs with data dependencies in behavioral system descriptions.

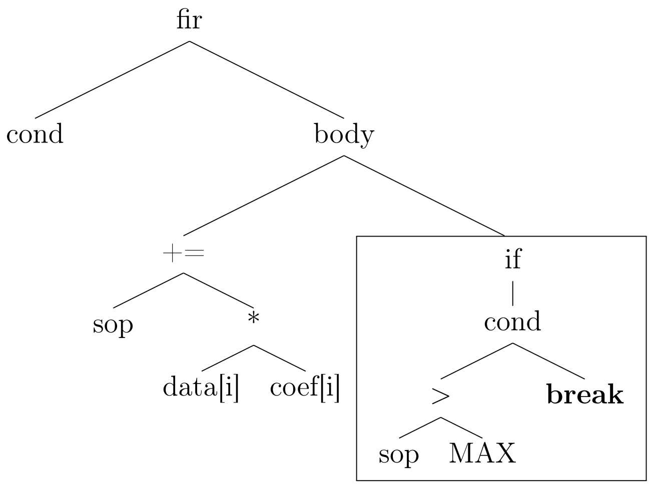
- Introduced the concept of behavioral template to accelerate cycle-accurate simulations by abstracting away the functionality of dedicated hardware accelerators, while maintaining their timing accuracy.

- Proposed different types of templates based on the internal structure of the accelerator, by investigating data dependencies in order to increase the accuracy of the simulations.

The goal of creating behavioral IP(BIP) templates is to substitute each IP (either RTL IP or behavioral IP) with a template which mimics the IPs' I/O behavior, but is *empty* inside. This implies that it only reads data from the master and returns data after X cycles similar to the original IP, where X is the latency of the IP. Although the results returned are functionally incorrect, the timing behavior is preserved. As a result, the advantage of preserving the workload pattern is preserved as mentioned earlier. The compile time of the entire model is accelerated as the complexity of these BIP templates is much lower than that of the actual IPs, and the simulation is much faster as each template does not require to perform any actual computation.

```
int fir(int *data,
    int *coef){
int sop=0;
for(i=0;i<N;i++){
   sop+=data[i]*coef[i];
   if(sop>MAX)
      break;
}
return sop;
}
```

(a) FIR filter with control dependent *break* example.



(b) Parse tree of FIR example.

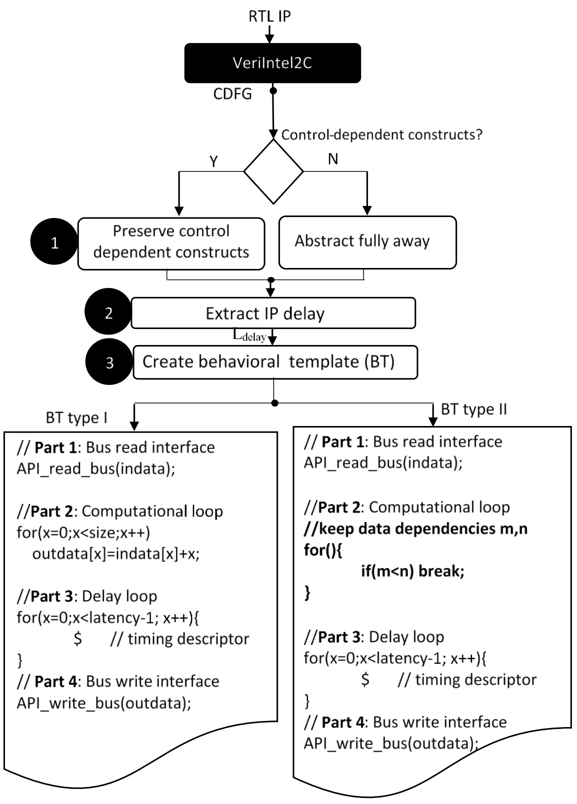
Fig. 7.1: Behavioral description of FIR filter with its parse tree representation

7.2 Methodology

7.2.1 Proposed Cycle-accurate System Simulation

This work introduces the concept of Behavioral IP(BIP) templates to substitute the hardware accelerators mapped as slaves in the system to reduce the run-time of cycle-accurate simulations of very large complex systems. The overall flow of the process is shown in figure 7.2 and algorithm 6. The input to the system can be either a BIP written in C/C++/SystemC or a RTL IP written in Verilog. This chapter describes the methodology solely for the latter input type. The three main steps are outlined below which are used to design a BIP template generator for the system.

Pre-Step: The input of the proposed method can be either a behavioral IP (BIP) given in ANSI-C or SystemC or a RTL IP given in Verilog. In this second case, a


RTL IP
VeriIntel2C
CDFG
Control-dependent constructs?
Y
N
1
Preserve control dependent constructs
Abstract fully away
2
Extract IP delay
L_{delay}
3
Create behavioral template (BT)
BT type I
BT type II

```
// Part 1: Bus read interface
API_read_bus(indata);

//Part 2: Computational loop
for(x=0;x<size;x++)
   outdata[x]=indata[x]+x;

//Part 3: Delay loop
for(x=0;x<latency-1; x++){
        $     // timing descriptor
}
// Part 4: Bus write interface
API_write_bus(outdata);
```

```
// Part 1: Bus read interface
API_read_bus(indata);

//Part 2: Computational loop
//keep data dependencies m,n
for(){
        if(m<n) break;
}

//Part 3: Delay loop
for(x=0;x<latency-1; x++){
        $     // timing descriptor
}
// Part 4: Bus write interface
API_write_bus(outdata);
```

Fig. 7.2: BIP template generator and BIP template overview.

previously developed RTL to C (RTL2C) converter is used to convert this RTL IP into synthesizable ANSI-C code(lines 1-4). This RTL2C converter has been modified in order to identify data dependencies that might affect the timing, especially the latency, of the synthesized circuit. The timing of each accelerator is critical to speeding up cycle-accurate simulation. At this stage, the functional correctness is not important as the objective is to generate and improve the traffic of the overall SoC. Once the overall SoC architecture is fixed, the design can be forwarded for circuit generation. The next subsection describe in detail these changes. The output of the

RTL2C converter has the same format as the synthesizable ANSI-C or SystemC code taken as inputs otherwise.

Algorithm 6: System creation using Behavioral Templates (BTs).

```
Data: $IP, interface, L, f, techlib$
$IP$: IP in RTL or C/SC
$L$: IP latency
$f$: HLS target frequency
$techlib$: technology library
Result: $BT$: : Behavioral Template for IP
/* Pre-Step: Convert RTL to C code */
if $IP=RTL$ then
    $C = convert\_rtl2c(IP)$;
else
    $C = IP$;
/* Step 1: Check for data dependent code */
$AST = parse(C)$;
/* Step 2: Extract delay of loop */
$(L_{min\_IP}, L_{max\_IP}) = hls(C, f, techlib)$;
if $(AST)$ then
    $BT\_no\_dly = gen\_BT\_no\_dly(C)$;
    $(L_{min\_BT}, L_{max\_BT}) = hls(BT\_no\_dly, f, techlib)$;
    $L_{dly} = L_{min\_C} - L_{min\_BT}$;
else
    $L_{dly} = L_{min} = L_{max}$;
/* Step 3: Write Behavioral template */
$BT = gen\_BT(C, L_{dly}, bus)$;
return($BT$);
```

Step 1: Check for Data Dependencies. The synthesizable C code is parsed and analyzed for data dependencies (DD) that might affect the latency of the synthesized circuit (line 9) . These data dependencies include $DD = \{break, continue, exit, return\}$ keywords in the source code combined with their execution condition ($i.e.if - else$, $switch - case$ and loops). Although $if - else$ conditions by themselves might also result in executions of different latencies, modern HLS tools extensively exploit static speculation. This significantly reduces or even eliminates the uneven execution of conditional statements. We also did not see any latency difference in the benchmarks that contained if-else conditions.

Upon synthesizing (HLS) behavioral descriptions having these constructs, the latency of the resultant circuit will not be constant and will depend on when the $COND$ constructs are executed. This implies that the latency L is not a single value, but can take any value between $L = [L_{min}, L_{max}]$, where L_{min}, is the smallest possible latency (e.g. when all the $COND$ statements are executed on the very first iterations of each loop) and L_{max} implies that none of the $COND$ statements are never executed and hence all loop iterations are fully executed. The average latency would therefore be $L_{avg} = \frac{(L_{max}-L_{min})}{2}$. Thus, it is important to take into account these data dependencies in order to preserve the timing accuracy. If any $COND$ exist, the proposed method generates a Abstract Syntax Tree (AST) of the C code, as shown in figure 7.1.

Step 2: Latency estimation. If the behavioral description does not contain any $COND$ constructs, the BIP is synthesized (HLS) and the latency (in clock cycles) reported by the synthesizer L_{HLS} assigned to the accelerator $L_{dly} = L_{HLS}$. The HLS tool by default returns the latency as $L_{HLS} = [L_{min\_IP}, L_{max\_IP}]$, where in this case $L_{min\_IP} = L_{max\_IP}$ and thus the BT latency is $L_{dly} = L_{min\_IP} = L_{max\_IP}$.The BT is in turn fully abstracted away and substituted by the behavioral template (BT). The structure of the template is described in detailed in the next subsection. In the scenario of BIP containing $COND$ constructs, the portion of the code that is required until the $COND$ construct is resolved is preserved in the BT, while the rest is abstracted away using algorithm 6 (lines 1-4). In this case the latency of the behavioral template needs to be adjusted in a different way. The method is described in Algorithm 6. In this case, the method synthesizes (HLS) the code twice. First, similarly to the previous case (without $COND$), it synthesizes the original C code obtaining $L = [L_{min\_IP}, L_{max\_IP}]$, where in this case $L_{min\_IP} \neq L_{max\_IP}$. Then, a trimmed version of the BT is generated which only contains the code with the data dependent part of the C code and is again, synthesized. The result is another latency pair $L_{min\_BT} \neq L_{max\_BT}$. The final BT latency is hence the difference between any of the two latencies obtained, $L_{dly} = [L_{min\_BT} + (L_{min\_IP} - L_{min\_BT}), L_{max\_BT} +_{max\_IP}$ $- L_{max\_BT}]$ (lines 11 to 15 of Algorithm 6).

Step 3: Behavioral Template Output. This very last step, takes as inputs the C code with or without data dependent code, the delay required by the delay loop of the template (L_{dly}), and the interface type ($interface$) and generates the synthesizable C code for the behavioral template (line 21).

The important contribution in this scenario is to detect the $COND$ constructs from input RTL HW accelerators using an extended version of our already devised framework, *VeriIntel2C*. The framework creates a CDFG of the input design and then identifies the $COND$. The following sub-section describes the algorithm over *VeriIntel2C* that preserves the code section with the identified constructs.

7.2.2 CDFG of conditional constructs

The section of CDFG particular to identifying the $COND$ constructs, is the control block which also houses the conditional structures, if present, in the design. There are mostly two distinct scenarios inside a conditional construct of a loop which can occur in a design. The control dependencies may be present in conditions inside loops and/or along with **1.** *break* or **2.** *continue* keywords. These scenarios also termed as $COND$ are mostly like to impact the execution latencies.

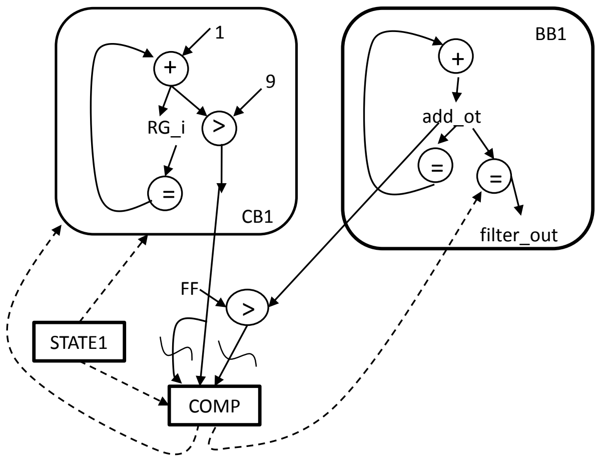


Fig. 7.3: CDFG with conditional scenarios

The generated CDFG consists of a loop representation split in two parts. The first part is the loop control graph defined as the control loop block(CLB) and the second part is the loop body in the basic block. Thus, if a loop structure consists of conditional forms, their respective data-flow nodes(part of loop sub-graph) are controlled by conditional control blocks representing comparator nodes. Figure 7.3 shows the graphical description of the CDFG of a loop with a conditional structure having *break* and *continue* scenario for FIR filter code in figure 7.1a. The first part is the loop control graph defined as the control block and the second part is the loop body in the basic block. Thus, if a loop structure consists of conditional forms, their respective data-flow nodes (part of loop sub-graph) are controlled by conditional control blocks representing comparator nodes.

The CDFG generated in *VeriIntel2C* already provides a clear distinction of all the representation of the aforementioned scenarios. The loop structures in the basic block are executed depending on the result of the conditional control node. In the event of this scenario, there is also the possibility of a loop sub-graph having multiple exit points. The exit points of a loop may also be incident on the conditional control node($COMP$) connected from the main loop sub-graph and then the resultant edge from $COMP$ controls the final assignment to the output node of the program $filter\_out$. The negation on the conditional edge indicates it should be executed if the condition is not satisfied. The negation operator controls the data-flow within resulting nodes. This particular CDFG indicates that upon the execution of the conditional loop sub-graph, the following nodes from the exit-point constitute operations external to the loop body and this is an indirect branch connecting loop operations and non-loop operations. These scenarios are created as a standard *template* and categorized as *break* scenarios wherein, the control switches to operations outside the loop under specific conditional statements inside the loop. The method builds an abstract syntax tree (*AST*) shown in Figure 7.1b and keeps all the code required to resolve this data dependency. In the case of figure 7.1b, all the code is kept in the same tree branch(in the enclosed box). In the worst case, this could imply that the BIP can not be abstracted away and thus the *BT* would be exactly the same as the BIP.

The execution behavior of *continue* scenario is functionally opposite to that of the *break* condition. Thus, the structure of CDFG must still be able to adapt to the loops with these conditions while maintaining the basic underlying form bounded by its basic structural rules. The counter variable node, *RG_i* (*i* in figure 7.1a) is controlled using a control edge by the $COMP$, where the value of the control variable is sent to the edge controlling the write operation to the array. The negating edge from the *greater than* operator node, is described for the *continue* scenario wherein

Algorithm 7: Rule-base algorithm for identifying *break* and *continue* scenarios in CDFG

```
Data: $CDFG_{loop}$
Result: $BTwithloop$
/* Step 1: Extract all CLB from CDFG */
if $CLB$ is present then
    for each $CLB$
    /*Extract all exit points from $CLB$*/
    if $COMP$ exists as exit node then
        /*Search for negation edges */
        /*Trace path of negation edges*/
        if path leads to loop operations then
            $loops_{BT} \leftarrow CLB$
            /*Insert break in $loops_{BT}$*/
        else
            /*Trace path of non-negation edges*/
            if /*path repeats in loop operations*/ then
                $loops_{BT} \leftarrow CLB$
                $OP_{loop} \leftarrow$ Extract node before entering loop
                /*Insert continue before $OP_{loop}$in $loops_{BT}$*/
```

the negation of the edge decides the flow of control to iterate the RG_i. Thus, in occurrence of this type of scenario, it is implied that the conditional CN demands its associated nodes to execute following which the loop condition should continue to iterate further, hence justifying the use of *continue* keyword at the end of the condition. This scenario is classified as the *continue* condition.

7.2.3 Search algorithm

A proposed rule-based search-and-traverse algorithm shown in algorithm 7 is used upon the generated CDFG to identify the aforementioned scenarios. The algorithm uses CLB to traverse through the exit nodes until one of them has a $COMP$ attached to one of them (lines 3-5). Here, it is to be noted that $COMP$ is a conditional node and hence has two exit edges, amongst which one has a negation operator called as negation edges. The algorithm 7 clearly shows that the *break* scenario requires the negation edge must lead to loop operations along with the other edge re-iterating to the external non-loop operations(lines 8-10) and *continue* having the reverse conditions to be satisfied(lines 12-16). This is based on the programming theory that a *break* scenario occurs when a condition inside a loop is satisfied and the control passes onto the external operations beyond the loop. A *continue* scenario occurs when the condition of the loop is satisfied and the control continues to be upon the loop block in the CDFG. Finally, a template code is generated consisting of the I/Os identified from the RTL design with the loop constructs having conditions with *break* and *continue* scenarios. In this manner, the CDFG framework is utilized for generating behavioral templates to improve the run time of behavioral simulations.

7.3 Behavioral Template (BT) Generator

Figure 7.2 also shows the main structure of the BT. The input to the template generator is the target HLS frequency and technology library, the interface type, either standard or custom, and the latency (L_{dly}) given in clock cycles. Alternatively the user can also manually specify the $BT's$ latency. This allows the user to explore different scenarios quickly. In order to create a more flexible template, the latency is passed as an input parameter to each template, thus making it fully parameterizable at runtime (it does not require to be re-synthesized each time the latency changes).

The template generator takes these inputs and generates two different types of synthesizable ANSI-C programs: BT type I and BT type II. Both types of templates are composed of 4 main parts:

Part 1: Interface read. The first part contains the synthesizable interface API

```
module fir_dat ();
filter_out = aot;
assign li2 = 4'h9 ;
assign gi1 = aot ;
assign gi2 = 8'hff ;
assign li1 = RG_i + 1;
assign aot = sop+mot;
assign mot = in00 * coeff00;
always @ ( aot or ST1_02d )
 sop = ( { 8{ ST1_02d } } &
     aot );
always @ ( iot or ST1_02d )
 RG_i_t = iot ;
always @ ( posedge clk )
 RG_i <= RG_i_t ;
assign out_en = ( ( ST1_02d &
    got ) | ( ( ST1_02d & (
    ~got ) ) & (~lot ) ) ) ;
always @ ( posedge clk )
 if ( out_en )
  filter_out <= aot ;

endmodule
```

(a) RTL description of a FIR filter

```
int fir(void){
int RG_SOP=0;
int in_read[8];
int coeff_read[8];
int mulot,addot;
int filter_out;
for(i=0;i<9;i++){
 mulot = in_read[i] *
     coeff_read[i];
 addot = RG_SOP + mulot;
 RG_SOP = addot;
 if(addot < 255)
  filter_out = addot;
 else if(addot>255)
  break;
}
return filter_out;
}
```

(b) Generated C program with *COND* constructs.

Fig. 7.4: Input RTL design and the resulting C design using proposed method of *VeriIntel2C*

required to read data into the accelerator. The choice of interfaces is based on the synthesizable APIs provided by the commercial tool used in this work. HLS tools provide libraries of synthesizable APIs for standard interfaces in order to facilitate the work of the designer. Currently, FIFO, RAM, AHB, and AXI are supported. For custom interfaces (e.g. a module which has to interface with an LCD display), the user can encapsulate the interface as a synthesizable function and include into the interface API library used by the *BT* template generator. In this case the interface is taken from this library and only the computational and delay loop are generated (parts 2 and 3).

Part 2: Computational Loop. The second part differs based on the type of BT to be generated. In the case that no $COND$ constructs are found in the original code to be abstracted, BT type I is generated. For this type of BT, the computational loop performs some basic computation on the input data. This is important because the HLS tool would optimize the logic of the entire template away if no computation is performed. Hence this part ensures that the template structure is preserved. In the case that $COND$ are present, a BT type II is generated. This implies that the the code that computes the $COND$ condition is preserved from the original BIP including the loop where it is used. The rest of the code is fully abstracted away. In the worst case, this could imply that the BIP can not be abstracted away and thus the BT would be exactly the same as the BIP. It should be nevertheless noted that most of the accelerators lead to BT of type I.

Part 3: Delay Loop. The third part contains the delay loop to make the $BT's$ final latency match the latency of the original BIP. For this purpose a loop containing only a timing description is used. The timing descriptor symbolizes a clock cycle. Commercial HLS tools normally support different types of scheduling modes. The traditional one is the automatic scheduling mode, in which the HLS scheduler automatically times the behavioral description. Another scheduling mode provided is manually mode. This implies that the user can manually time the behavioral description by inserting the clock boundaries directly in the code. In SystemC this is done with wait statements, while at the ANSI-C level, this is vendor specific. In the case of the commercial HLS tool used in this work, a *\$* sign is used to denote a clock boundary. Hence when having a loop with N iterations it will take N cycles to execute the loop. For HLS tools that do not allow this type delay control, the delay can easily be achieved in the computationally loop by performing simple operations in a for loop sequentially. As mentioned previously, the main problem in some applications is that the exact circuit latency is unknown a priori. For this purpose the proposed method has two options. By default, if the user does not specify the latency as an input, the method synthesizes (HLS) the input BIP once and extracts the latency reported by the HLS tool. For case I types, with no $COND$, this should be accurate. For case II, with $COND$, the method synthesizes the BIP and extracts the min and max latency reported by the HLS tool for the full original C description and for a C description with only the $COND$ part of the code.

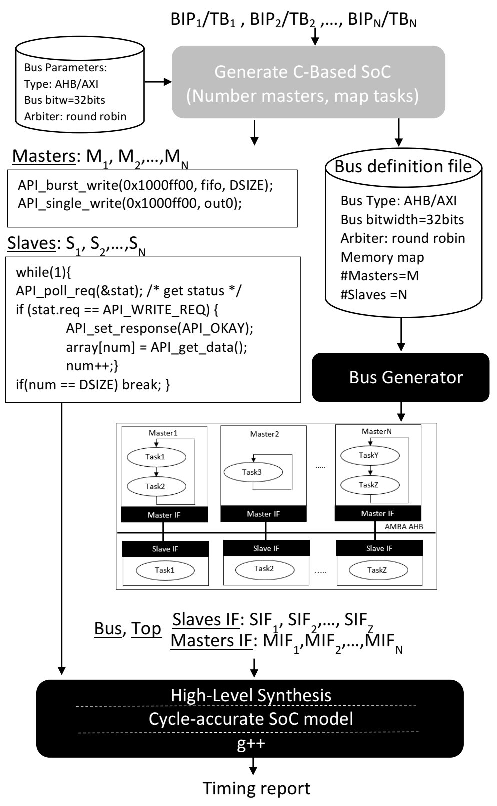
Part 4: Interface write. Finally, the last part contains write back portion of the interface using again the synthesizable API provided by the vendor or the custom interface encapsulated in a library by the user.

With regards to the time required to generate the BT, it should be noted that the most time consuming part is the HLS in order to extract the latency of the accelerator. Only a single HLS is required for BT of type I (without data dependencies), while for BT of type II (with loop data dependencies), our template abstraction method requires two HLS. The first synthesis on the original behavioral description and the second of the optimized one in order to determine the latency of the delay loop. During the experimental phase we noted that a single HLS on any of the benchmarks did not exceed 10 seconds. It should also be noted that the RTL to C conversion is extremely fast, taking less than 1 second. In case that the user knows the accelerator's latency or wants to explore different *what-if* scenarios, the time to generate these templates is negligible as the latency is passed as a parameter to the BT.

7.4 Experiment Results

The experimental results here, shall solely focus on the result of implementation of the algorithm using the *VeriIntel2C* and would not be reflecting upon the overall quality of the result of cycle-accurate behavioral simulations since the objective of this work is to show the ability of the designed translation framework to be applied for solving challenges in different domains and not just for enabling DSE for RTL descriptions.

The explorer is built around a behavioral MPSoC generator shown in figure 7.5, which takes as inputs M masters and N slaves acting as accelerators connected through a standard shared bus (*e.g.*, AHB/AXI). Thus, the type of bus, its bitwidth and arbiter also need to be specified as inputs. The masters and slaves are given as synthesizable (HLS) behavioral IPs in ANSI-C or SystemC. These BIPs have been modified to include synthesizable APIs for sending and receiving data over the bus that the commercial HLS tool used in this work supports to abstract the bus interface away during the design. The bus definition file with the bus information and the BIPs are passed to the commercial HLS tool bus generator which after generating the bus and bus interfaces synthesizes each module and generates a cycle-accurate model of the complete SoC. This cycle-accurate model is generated in SystemC, which in turn can be compiled with any C++ compiler. Figure 7.5 is color coded, where black boxes represent third party tool and gray boxes parts of the flow implemented by us.



```
API_burst_write(0x1000ff00, fifo, DSIZE);
API_single_write(0x1000ff00, out0);
```

```
while(1){
API_poll_req(&stat); /* get status */
if (stat.req == API_WRITE_REQ) {
        API_set_response(API_OKAY);
        array[num] = API_get_data();
        num++;}
if(num == DSIZE) break; }
```

Fig. 7.5: Automatic MPSoC generator overview.

As mentioned in the introduction, C-based design has an additional advantage over traditional RTL design. A single behavioral description allows the generation of different micro-architectures with unique area vs. performance trade-offs. This is typically done by setting different synthesis options to determine how to synthesize e.g. arrays (RAM, register), loops (unroll, folded), and functions (inline or not). Thus, in order to enable the system explorer, a variety of micro-architectures obtained from a previously developed HLS DSE proposed in the earlier chapters [47] is used as input for each hardware accelerator used in the system .

Table 7.1: Runtime results for 10 million cycles of S2Cbench benchmarks .

Kernel	Lines	Lines BT	L_{diff}	Run [s]	Speedup
No runtime data dependencies					
qsort	104	25	4.2	59	8.6
sobel	134	25	5.4	71	10.3
aes	429	25	17.2	173	25.1
kasumi	321	25	12.8	130	18.9
md5c	215	25	8.6	70	10.2
snow3G	314	25	12.6	103	14.9
adpcm	297	25	11.9	111	16.1
fft	142	25	5.7	98	14.2
fir	109	25	4.4	69	10.0
interp	134	25	5.4	106	15.4
Avg.			**8.8**		**14.7**
With runtime data dependencies					
decim	156	66	2.4	100	8.8
cholesly	161	161	0	113	0
idct	187	95	2.0	97	7.6
disparity	604	273	2.2	203	8.5
jpeg enc	1,509	874	1.7	287	5.3
Avg.			**2.2**		**8.3**

The experiments were run on an Intel dual-core 2.40 GHz Xeon processor machine with 16 GBytes of RAM running Linux Fedora release 19. The HLS tool used is CyberWorkBench v.5.52 [21]. The target technology is Nangates 45nm Opencell technology and the HLS target frequency for all of the processes in the system is set to 100MHz. The experimental results are divided into two sets of experiments.

A. Single Kernels: This first set of experiments compares the running time of a cycle-accurate simulation of single kernels against the proposed abstracted BTs. For this, all of the 14 designs of the open source synthesizable SystemC benchmarks suite S2CBench were used [76]. Table 7.1 shows the list of benchmarks, divided into two groups. The first does not contain data dependencies, while the second group does. Thus, for each of them a different type of BT is generated. All these benchmarks are synthesized (HLS) with default synthesis options using a commercial HLS tool and a cycle-accurate model is generated for each of them using the same commercial HLS tool[21]. The running time comprises the compilation time as well as the runtime to simulate 10 million random inputs.

The first observation is that, as expected, the BT simplifies the description of the BIP as indicated by the reduction of the lines of code. In the case of BT of type I, the template size is fixed (only the latency changes), and therefore the reduction in number of lines depends on the complexity of the accelerator. In the case of BT of type II, the complexity of the template depends on when the $COND$ can be resolved. The average line number reduction is 8.8× in the first and 2.2× for the latter. For BT of type II, the line number reduction obviously varies case by case, e.g. for the *cholesky* case no abstraction was possible. The simplification of the accelerator directly translates into faster execution time. From the results, it can be observed that the BT of type I is on average 14.7× faster, while the BT type II is slower, as expected, achieving averages speedups of 8.3×. The results indicate that the running time of the cycle-accurate simulations depend on the complexity of the kernel. From these results, it can be observed that the number of lines reduction seem to be a good indicator of the simulation speedup for BT of type I. In the case of BTs of type II, there is not such a direct relationship as in the worst case, no speedup is achieved(e.g. *cholesky* benchmark).

B. MPSoC Design Space Exploration: Different computational intensive applications,

Table 7.2: Complex System Benchmarks.

			Only BTs of type I				Only BTs of type II				BTs of type I and II			
Kernel	BT Type	DSE	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12
Behavioral Benchmarks														
md5c	I	3	1	1	1	1					1	1	1	1
kasumi	I	3	1		1	1					1		1	1
adpcm	I	5		1		1						1	1	1
idct	II	3					1	1	1	1	1	1	1	1
cholesky	II	4					1	1	1	1	1		1	1
jpeg enc	II	8						1		1		1	1	1
RTL Benchmarks														
aes	I	1			1	1								1
decim	II	1							1	1				1
Tasks			2	2	3	4	2	3	3	4	4	4	6	8
Designs			6	8	7	12	7	15	8	16	13	19	26	28
Masters			1-3	1-3	1-3	1-3	1-3	1-3	1-3	1-3	1-3	1-3	1-3	1-3

amenable to HW acceleration, were selected and grouped together into complex systems in order to test our proposed method. These designs were again taken from the open source Synthesizable SystemC Benchmark suite (S2CBench) [76]. Table 7.3 shows the individual benchmarks selected from S2Cbench and their characteristics in terms of data dependencies, lines of code, arrays and functions. Two RTL IPs were also used from open-source [58]. Table 7.2 shows how these complex benchmarks were formed. The first column indicates the name of benchmark, and the second column indicates the type of behavioral template that can be generated for this particular benchmark. The third column indicates the total number of dominating designs reported by the DSE for each benchmark. Columns S1-S12 indicate the benchmark used to build each complex system benchmark. The last three rows report the total number of applications (benchmarks/tasks) used in each system benchmark, the total number of design candidates contained (adding up the results of the DSE of each application) and the number of masters considered. Three distinct group of systems

Table 7.3: Characteristics of Hardware Accelerators (kernels).

Kernel	DD	lines	arrays	functions
Behavioral Accelerators				
cholesky	Y	211	2	2
md5c	N	265	5	7
kasumi	N	373	13	5
adpcm	N	342	1	3
idct	Y	237	2	2
jpeg enc	Y	1,561	16	8
RTL Accelerators				
decim	Y	231		
aes	N	174		

are created in order to fully characterize our proposed method. The first (S1 to S4) only contain benchmarks that can be translated into BTs of type I (no $COND$). The second (S5 to S8) only contain benchmarks which are translated into BT of type II ($COND$ present). Finally, group three contain a mixture of both (S9 to S12).

The target architecture, for these experiments, is a multi-core processor system with masters ranging from 1 to 3 depending on the benchmark. The masters and slaves are connected through a 32-bit AMBA-AHB bus using a round robin arbiter (loosely coupled hardware accelerator system).

Table 7.4 and Table 7.5 show the qualitative and quantitative results respectively of our method, making use of the behavioral templates to speed up the simulation, for the exhaustive search exploration method (BFT) and for the simulated annealing based exploration method with exact SA_{exact} and approximate SA_{BT}) versions. No results are show for the exhaustive search explorer with exact slaves in Table 7.4, as this is the reference front and hence has an $ADRS$=0 and a Pareto dominance 100%.

Different conclusions can be drawn from the results shown in Table 7.4 and Table

Table 7.4: Experimental Results

		Exhaustive Search		*Simulated Annealing*			
		BF_{BT}		SA_{BT}		SA_{exact}	
Bench	Masters	ADRS[%]	Dom[%]	ADRS[%]	Dom[%]	ADRS[%]	Dom[%]
S1	M=1	0.0	100	0.0	100	0	100
	M=2	0.0	100	1.1	75	4.3	75
	M=3	0.0	100	1.8	75	6.5	70
S2	M=1	0.0	100	2.2	90	3.8	50
	M=2	0.0	100	4.1	66	7.7	33
	M=3	0.0	100	5.3	66	7.8	43
S3	M=1	0.0	100	1.4	75	3.4	50
	M=2	0.0	100	3.7	75	8.4	33
	M=3	0.0	100	1.4	75	8.6	25
S4	M=1	0.0	100	3.2	100	7.6	41
	M=2	0.0	100	0.0	100	8.6	50
	M=3	0.0	100	3.2	75	7.9	33
S5	M=1	0.0	100	0.0	100	0.0	100
	M=2	0.0	100	0.0	100	1.5	50
	M=3	1.2	75	1.2	75	3.5	50
S6	M=1	3.5	50	3.5	50	7.8	75
	M=2	6.7	50	10.6	33	3.5	75
	M=3	5.7	50	5.7	50	6.5	50
S7	M=1	6.5	80	6.5	80	0.0	100
	M=2	9.6	80	9.6	80	10.4	25
	M=3	8.1	80	8.1	80	12.6	33
S8	M=1	8.5	75	10.2	75	11.3	25
	M=2	9.3	75	9.3	75	6.7	25
	M=3	10.4	75	12.8	75	15.5	25
S10	M=1	1.3	90	2.6	75	0.5	90
	M=2	2.4	85	2.4	85	2.1	90
	M=3	2.7	85	2.7	85	5.8	25
S10	M=1	3.1	75	3.7	50	6.4	33
	M=2	3.6	75	3.6	50	8.1	25
	M=3	4.1	66	4.5	50	8.7	25
S11	M=1	4.3	66	4.3	50	10.2	50
	M=2	5.1	75	5.7	33	12.3	25
	M=3	4.9	75	4.9	33	10.1	33
S12	M=1	5.8	80	9.6	80	17.3	25
	M=2	6.7	75	9.4	66	19.5	15
	M=3	7.1	66	10.4	66	21.2	10
Avg. BT Type I only (S1-S4)		0.0	100	2.3	81	6.2	50
Avg. BT Type II only(S5-S8)		5.8	69	6.5	65	6.6	53
Avg. BT Type I+II (S9-S12)		4.3	76	5.3	67	10.2	37
Avg. Total		**3.4**	**82**	**4.7**	**72**	**7.7**	**47**

Table 7.5: Running time results [min]

	BF	$BF_{template}$	Comparison
Bench	Run[min]	Run[min]	Δ_{BF-BFT}
S1	50	9	5.6
S2	110	15	7.3
S3	55	8	6.9
S4	802	63	12.7
S5	386	204	1.9
S6	957	499	1.9
S7	1,741	801	2.2
S8	1,478	1,205	1.2
S9	1,045	487	2.1
S10	4877	980	5.0
S11	12,554	2,591	4.8
S12	13,904	3,184	4.4
Avg. BT Type I (S1-S4)			8.12
Avg.BT Type II (S5-S8)			1.80
Avg. BT Type I+II (S9-S12)			4.73
Total Avg. (S1-S12)			**4.67**

7.5. In general the proposed method works well as shown by the low average ADRS and dominance. The exhaustive search (BF_{BT}) is only 3.4% worse for the ADRS and 82% for the dominance. The results have to anyway be analyzed carefully. When the accelerators have no data dependencies and thus, BTs of type I can be used for all of them, the method works extremely well matching the exact solution results in all cases (S1-S4 systems). In the case that data dependencies exists and BT of type II need to be used, the method still works well, but the quality of the solution degrades to an ADRS of 5.8% and dominance of 69%. Finally when systems mixed with both types of accelerators the average ADRS is 4.3% and average dominance is 76%. This implies that the QoR degrades depending on the type of accelerator, but the results are still very good.

The main benefit of the proposed method is shown in Table 7.4. Given a maximum amount of time to produce an exploration result, our proposed method can generate more combinations and hence leads to better results. In this case, given only 10% of the runtime taken by the exhaustive search, the average *ADRS* and dominance values are still very good (4.7% and 72%), while using the exact slaves lead to much worse results with an average *ADRS*=7.7% and an average Pareto dominance of 47%. On an average, this is an ADRS of 39% and a Pareto dominance decreased by 34%. The large ADRS value means that the exact method misses complete sections of the trade-off curve, which can be extremely important for system designers. Upon investigation of this results, it was found that the main culprit for the discrepancies in results is that the HLS tool used in this work, would often not report the accurate latency values for each benchmark. One way to increase the accuracy, would be to perform a cycle-accurate simulation of every accelerator in order to accurately annotate the latency instead of relying on the latency reported right after HLS.

Table 7.5 shows the quantitative by comparing the running times of the exhaustive search of both methods. Results shows that the template based method is on average 4.67 $\times$ faster further indicating the effectiveness of our method. Here again, speedup difference can be observed depending on the type of templates used. When the accelerator can be fully abstracted away speedups of on average 8.12$\times$ are obtained(S1-S4). In the case that every accelerator contains DDs (S5 to S8), speedups of on average 1.8$\times$ are obtained and finally in mixed systems (S9 to S12) average speedups of 4.73$\times$ are reached.

7.5 Summary

This chapter describes the application of the proposed translation framework *VeriIntel2C* for generating behavioral templates for accelerating system-level simulations. The rule-based algorithm successfully identifies the constructs which are control-dependent and impact the overall circuit's latency indirectly affecting the system simulations. The main objective of the *VeriIntel2C* is to identify those critical sections of RTL IPs for simulation and translate them into behavioral templates. The scenarios commonly identified as loops with conditional constructs having *break* and *continue* keywords are successfully identified and extracted solely into the behavioral templates.

[0]This work is accepted for publication in Integration the VLSI Journal, 2018.

Chapter 8

Conclusion

In summary, this thesis enables and improves the Design Space Exploration (DSE) and simulation of heterogenous SoC platforms by unifying all HW accelerators to the behavioral level using *VeriIntel2C*.

The thesis introduces High Level Synthesis, describes its methodology and advantages of being able to perform Design Space Exploration (DSE) without modifying the behavioral description to be explored. Further, the thesis presents the first open source benchmark suite in SystemC (language common in all HLS tools), *S2CBench*. These benchmarks were used to advance the state-of-the-art in HLS DSE by proposing a mixed static-dynamic method for DSE which uses machine learning and simulated annealing.

The main contribution of this thesis can be found in chapter 5, where an abstraction framework is introduced called *VeriIntel2C* to translate RTL IPs given in synthesizable Verilog to C programs to maximize HLS DSE. For this purpose, applications of Petri Net model in graph modeling applications were studied, given at the RT-Level and its advantages in the RTL to C translation. Using the advantages of Petri Net graphs, a graph-based translation framework is developed that extracts the

CDFG using the model of Petri Net graph. The Petri Net graph is modeled from an Abstract Syntax tree (AST) by parsing the RTL design. The generated CDFG contains distinct patterns and structures which are identified by the proposed rule-based algorithms for graph traversal.

Furthermore, the thesis investigates the shortcomings of *VeriIntel2C* by performing extensive experiments on the exploration of the generated designs. It was identified that the designs having multi-dimensional arrays and nested loops, upon exploration, produce more number of pareto-optimal design points compared with that of designs with read-only one-dimensional arrays. As a result, optimization algorithms were proposed and developed using loop fusion methods in order to merge loops and array merging methods. Upon experiments, it was observed that the proposed optimization methods expanded the design space for certain RTL designs and the results were compared with that of the base method, *VeriIntel2C*.

Finally, in order to demonstrate an application of *VeriIntel2C*, a method was developed to accelerate cycle accurate simulations of hardware accelerators in heterogenous SoCs. This method generates Behavioral Templates and uses *VeriIntel2C* to create the templates for RTL designs. Experiments were also performed on system-level exploration of the heterogenous SoC and prove improvement on the run time of the exploration.

Chapter 9

Future Work

Our future research direction is outlined as follows. In the era of heterogenous SoCs with IPs both in HLS and RTL, it is imperative to translate RTL towards HLS for expanding the DSE for RTL IPs with the objective of generating different forms of loops and arrays. The proposed translation method, *VeriIntel2C* and its optimizations *VeriIntel2C-opt* have aimed to translate synthesizable RTL designs into HLS descriptions that expands DSE of RTL designs. However, in lieu of the limitations of our method described in previous chapters, the thesis aims to extend this work beyond graph and rule-base and create an intelligent tool using deep learning. This would enable the method to evolve on learning the functionalities of RTL designs written in different design styles. Another feature to be added would be the abstraction of modules or operations into functions in C level. Functions are also one amongst the *explorable* constructs that contribute towards expanding the design space. Thus, we will aim to extend *VeriIntel2C* using these proposed features to effectively expand the design space of different types of HW accelerators. The measure, for original C-based designs, can be used as a reference to reach the optimal quality of DSE.

Bibliography

[1] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers, Principles, Techniques*. Addison wesley Boston, 1986.

[2] aldec. Aldec simulation. URL `https://www.aldec.com/en/solutions`.

[3] Giuseppe Ascia, Vincenzo Catania, Alessandro G. Di Nuovo, Maurizio Palesi, and Davide Patti. Efficient design space exploration for application specific systems-on-a-chip. *J. Syst. Archit.*, 53(10):733–750, October 2007. ISSN 1383-7621. doi: 10.1016/j.sysarc.2007.01.004. URL `http://dx.doi.org/10.1016/j.sysarc.2007.01.004`.

[4] IEEE Standards Association et al. Ieee standard for standard systemc language reference manual. *IEEE Computer Society*, 2012.

[5] David Atienza, Jose M Mendias, Stylianos Mamagkakis, Dimitrios Soudris, and Francky Catthoor. Systematic dynamic memory management design methodology for reduced memory footprint. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 11(2):465–489, 2006.

[6] Andrey Ayupov, Serif Yesil, Muhammet Mustafa Ozdal, Taemin Kim, Steven Burns, and Ozcan Ozturk. A template-based design methodology for graph-parallel hardware accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(2):420–430, 2018.

[7] Giovanni Beltrame, Luca Fossati, and Donatella Sciuto. Decision-theoretic design space exploration of multiprocessor platforms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(7):1083–1095, 2010.

[8] Luca Benini, Davide Bertozzi, Davide Bruni, Nicola Drago, Franco Fummi, and Massimo Poncino. Legacy systemc co-simulation of multi-processor systems-on-chip. In *Computer Design: VLSI in Computers and Processors, 2002. Proceedings. 2002 IEEE International Conference on*, pages 494–499. IEEE, 2002.

[9] N. Bombieri, Hung-Yi Liu, F. Fummi, and L. Carloni. A method to abstract rtl ip blocks into c++ code and enable high-level synthesis. In *Design Automation Conference (DAC), 2013 50th ACM / EDAC / IEEE*, pages 1–9, May 2013.

[10] Nicola Bombieri, Franco Fummi, and Graziano Pravadelli. Abstraction of RTL IPs into Embedded Software. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 24–29, 2010.

[11] Nicola Bombieri, Franco Fummi, and Sara Vinco. A methodology to recover rtl ip functionality for automatic generation of sw applications. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 20(3):36, 2015.

[12] Calypto. Catapult hls tool. URL `http://www.calypto.com`.

[13] Keith Campbell, Wei Zuo, and Deming Chen. New advances of high-level synthesis for efficient and reliable hardware design. *Integration, the VLSI Journal*, 58:189–214, 2017.

[14] Raul Camposano. Path-based scheduling for synthesis. *IEEE transactions on computer-Aided Design of Integrated Circuits and Systems*, 10(1):85–93, 1991.

[15] carbon. Carbon model studio. URL `http://www.carbondesignsystems.com/`.

[16] B. Carrion Schafer. Probabilistic multi-knob high-level synthesis design space exploration acceleration. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, PP(99):1, 2015.

[17] Alessandro Cilardo and Luca Gallo. Interplay of loop unrolling and multidimensional memory partitioning in hls. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE '15, pages 163–168, San Jose, CA, USA, 2015. EDA Consortium. ISBN 978-3-9815370-4-8.

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

[19] Jason Cong and Zhiru Zhang. An efficient and versatile scheduling algorithm based on sdc formulation. In *Design Automation Conference, 2006 43rd ACM/IEEE*, pages 433–438. IEEE, 2006.

[20] Philippe Coussy and Adam Morawiec. *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer Publishing Company, Incorporated, 1st edition, 2008. ISBN 1402085877, 9781402085871.

[21] cwb. Cyberworkbench. URL `www.cyberworkbench.com`.

[22] Kaivalya M Dixit. The spec benchmarks. *Parallel computing*, 17(10-11):1195–1209, 1991.

[23] F. Ferrandi, P. L. Lanzi, D. Loiacono, C. Pilato, and D. Sciuto. A Multi-objective Genetic Algorithm for Design Space Exploration in High-Level Synthesis. In *2008 IEEE Computer Society Annual Symposium on VLSI*, pages 417–422, April 2008.

[24] Forte. Cynthesizer tool. URL `http://www.forteds.com/`.

[25] Dan Gajski, Todd Austin, and Steve Svoboda. What input-language is the best choice for high level synthesis (hls)? In *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, pages 857–858. IEEE, 2010.

[26] Quentin Gautier, Alric Althoff, Pingfan Meng, and Ryan Kastner. Spector: An opencl fpga benchmark suite. In *Field-Programmable Technology (FPT), 2016 International Conference on*, pages 141–148. IEEE, 2016.

[27] Catherine H Gebotys and Mohamed I Elmasry. Global optimization approach for architectural synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(9):1266–1278, 1993.

[28] Tony Givargis, Frank Vahid, and Jörg Henkel. System-level exploration for pareto-optimal configurations in parameterized system-on-a-chip. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 10(4):416–422, 2002.

[29] Sumit Gupta and Lubomir Bic. Distributed adaptive simulated annealing for synthesis design space exploration. 1999.

[30] Sumit Gupta, Nicolae Savoiu, Nikil Dutt, Rajesh Gupta, and Alexandru Nicolau. Using global code motions to improve the quality of results for high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(2):302–312, 2004.

[31] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3–14. IEEE, 2001.

[32] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, Hiroaki Takada, and Katsuya

Ishii. Chstone: A benchmark program suite for practical c-based high-level synthesis. In *Circuits and Systems, 2008. ISCAS 2008. IEEE International Symposium on*, pages 1192–1195. IEEE, 2008.

[33] Steve Haynal. Automata-based symbolic scheduling. 2000.

[34] HLSBenchmark. S2cbench open source. URL `https://sourceforge.net/projects/s2cbench/`.

[35] C-T Hwang, J-H Lee, and Y-C Hsu. A formal approach to the scheduling problem in high level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 10(4):464–475, 1991.

[36] ITRS. International technology roadmap for semiconductors. URL `http://www.itrs.net/reports.html`.

[37] Rajiv Jain, Ashutosh Mujumdar, Alok Sharma, and Hueymin Wang. Empirical evaluation of some high-level synthesis scheduling heuristics. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, pages 686–689. ACM, 1991.

[38] V. Kianzad and S.S. Bhattacharyya. CHARMED: a multi-objective co-synthesis framework for multi-mode embedded systems. In *IEEE Application-Specific Systems, Architectures and Processors*, pages 28–40, Sept 2004.

[39] Scott Kirkpatrick. Optimization by simulated annealing: Quantitative studies. *Journal of statistical physics*, 34(5):975–986, 1984.

[40] David Ku and Giovanni De Micheli. Relative scheduling under timing constraints. In *Design Automation Conference, 1990. Proceedings., 27th ACM/IEEE*, pages 59–64. IEEE, 1990.

[41] Krzysztof Kuchcinski. Constraints-driven scheduling and resource assignment. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 8 (3):355–383, 2003.

[42] Chunho Lee, Miodrag Potkonjak, and William H Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 330–335. IEEE Computer Society, 1997.

[43] Bin Lin and Fei Xie. Scbench: a benchmark design suite for systemc verification and validation. In *Proceedings of the 23rd Asia and South Pacific Design Automation Conference*, pages 440–445. IEEE Press, 2018.

[44] Bin Lin, Zhenkun Yang, Kai Cong, and Fei Xie. Generating high coverage tests for systemc designs using symbolic execution. In *Design Automation Conference (ASP-DAC), 2016 21st Asia and South Pacific*, pages 166–171. IEEE, 2016.

[45] Hung-Yi Liu and L.P. Carloni. On learning-based methods for design-space exploration with high-level synthesis. In *DAC*, pages 1–7, May 2013.

[46] Tai Ly, David Knapp, Ron Miller, and Don MacMillen. Scheduling using behavioral templates. In *Proceedings of the 32nd annual ACM/IEEE Design Automation Conference*, pages 101–106. ACM, 1995.

[47] A. Mahapatra and B.C. Schafer. Machine-learning based simulated annealer method for high level synthesis design space exploration. In *Electronic System Level Synthesis Conference (ESLsyn), Proceedings of the 2014*, pages 1–6, May 2014.

[48] Anushree Mahapatra and Benjamin Schafer. Optimizing rtl to c abstraction methodology to improve hls design space exploration. *IEEE Embedded System Letters*.

[49] Anushree Mahapatra and Benjamin Schafer. Veriintel2c: Abstracting rtl to c to maximize high level synthesis design space exploration. *Integration, The VLSI journal*, 2018.

[50] Norian Marranghello, Jaroslaw Mirkowski, and Krzysztof Bilinski. *Synthesis of Synchronous Digital Systems Specified by Petri Nets*. Springer US, 2000. ISBN 978-1-4419-4969-1.

[51] Michael C McFarland. Using bottom-up design techniques in the synthesis of digital hardware from abstract behavioral descriptions. In *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, pages 474–480. IEEE Press, 1986.

[52] Wim Meeus, Kristof Van Beeck, Toon Goedemé, Jan Meel, and Dirk Stroobandt. An overview of todayŠs high-level synthesis tools. *Design Automation for Embedded Systems*, 16(3):31–51, 2012.

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

[54] T.M. Mitchell. *Machine Learning*. McGraw-Hill International Editions. McGraw-Hill, 1997. ISBN 9780071154673. URL `https://books.google.com.hk/books?id=EoYBngEACAAJ`.

[55] Sumio Morioka, Toshiyuki Isshiki, Satoshi Obana, Yuichi Nakamura, and Kazue Sako. Flexible architecture optimization and asic implementation of group signature algorithm using a customized hls methodology. In *Hardware-Oriented Security and Trust (HOST), 2011 IEEE International Symposium on*, pages 57–62. IEEE, 2011.

[56] Ravi Namballa, Nagarajan Ranganathan, and Abdel Ejnioui. Control and data flow graph extraction for high-level synthesis. In *VLSI, 2004. Proceedings. IEEE Computer society Annual Symposium on*, pages 187–192. IEEE, 2004.

[57] nangatecell. Nangate. URL `www.nangate.com`.

[58] openRTL. open cores. URL `http://opencores.org/`.

[59] G. Palermo, C. Silvano, and V. Zaccaria. Respir: A response surface-based pareto iterative refinement for application-specific design space exploration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28 (12):1816–1829, Dec 2009. ISSN 0278-0070. doi: 10.1109/TCAD.2009.2028681.

[60] Preeti R Panda and Nikil D Dutt. 1995 high level synthesis design repository. In *Proceedings of the 8th international symposium on System synthesis*, pages 170–174. ACM, 1995.

[61] Barry M Pangrle. Splicer: A heuristic approach to connectivity binding. In *Design Automation Conference, 1988. Proceedings., 25th ACM/IEEE*, pages 536–541. IEEE, 1988.

[62] Alice C Parker, Jorge T Pizarro, and Mitch Mlinar. Maha: A program for datapath synthesis. In *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, pages 461–466. IEEE Press, 1986.

[63] Pierre G Paulin and John P Knight. Force-directed scheduling for the behavioral synthesis of asics. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(6):661–679, 1989.

[64] Zebo Peng and K. Kuchcinski. Automated transformation of algorithms into register-transfer level implementations. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 13(2):150–166, Feb 1994.

[65] N. K. Pham, A. K. Singh, A. Kumar, and M. M. A. Khin. Exploiting loop-array dependencies to accelerate the design space exploration with high level synthesis. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, New York, NY, USA, 2014. ACM.

[66] Miodrag Potkonjak and Jan Rabaey. A scheduling and resource allocation algorithm for hierarchical signal flow graphs. In *Design Automation, 1989. 26th Conference on*, pages 7–12. IEEE, 1989.

[67] Dave Prothero. Modelling and implementation of petri nets using vhdl. In *Hardware Design and Petri Nets*, pages 223–236. Springer, 2000.

[68] Vijay K Raj. Another automated data path designer. *ICCAD-86*, pages 214–217, 1986.

[69] Nagarajan Ranganathan, Ravi Namballa, and Narender Hanchate. Chess: a comprehensive tool for cdfg extraction and synthesis of low power designs from vhdl. In *Emerging VLSI Technologies and Architectures, 2006. IEEE Computer Society Annual Symposium on*, pages 6–pp. IEEE, 2006.

[70] Patrik Rokyta, Wolfgang Fengler, and Thorsten Hummel. Electronic system design automation using high level petri nets. In *Hardware Design and Petri Nets*, pages 193–204. Springer, 2000.

[71] C. Rust, A. Rettberg, and K. Gossens. From high-level petri nets to systemc. In *Systems, Man and Cybernetics, 2003. IEEE International Conference on*, volume 2, pages 1032–1038 vol.2, Oct 2003. doi: 10.1109/ICSMC.2003.1244548.

[72] B. Carrion Schafer and K. Wakabayashi. Machine learning predictive modelling high-level synthesis design space exploration. *Computers Digital Techniques, IET*, 6(3):153–159, 2012.

[73] B. Carrion Schafer and Kazutoshi Wakabayashi. Design space exploration acceleration through operation clustering. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(1):153–157, Jan 2010.

[74] B. Carrion Schafer and Kazutoshi Wakabayashi. Divide and conquer high-level synthesis design space exploration. *ACM Trans. Des. Autom. Electron. Syst.*, 17 (3):29:1–29:19, July 2012. ISSN 1084-4309.

[75] B. Carrion Schafer, A. Trambadia, and K. Wakabayashi. Design of complex image processing systems in esl. In *2010 15th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 809–814, Jan 2010.

[76] Benjamin Carrion Schafer and Anushree Mahapatra. S2cbench: Synthesizable systemc benchmark suite for high-level synthesis. *IEEE Embedded Systems Letters*, 6(3):53–56, 2014.

[77] Benjamin Carrion Schafer, Takashi Takenaka, and Kazutoshi Wakabayashi. Adaptive simulated annealer for high level synthesis design space exploration. In *VLSI Design, Automation and Test, 2009. VLSI-DAT'09. International Symposium on*, pages 106–109. IEEE, 2009.

[78] Wei Song, Jim Garside, and Doug Edwards. Automatic data path extraction in large-scale register-transfer level designs. In *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 377–380. IEEE, 2014.

[79] W. Stoye, D. Greaves, N. Richards, and J. Green. Using rtl-to-c++ translation for large soc concurrent engineering: a case study. *Electronics Systems and Software*, 1(1):20–25, Feb 2003.

[80] sysAcc. Systemc synthesizable subset version 1.4 draft. URL `https://workspace.accellera.org/downloads/drafts_review/SystemC_Synthesis_Subset_Draft_1_4.pdf`.

[81] Chia-Jeng Tseng and Daniel P Siewiorek. Automated synthesis of data paths in digital systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 5(3):379–395, 1986.

[82] Verific. Verific parser platform. URL `http://www.verific.com/products.html`.

[83] Verilator. Verilator. URL `http://www.veripool.org/wiki/verilator`.

[84] Aravind Vijayakumar and Forrest Brewer. Weighted control scheduling. In *Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design*, pages 777–783. IEEE Computer Society, 2005.

[85] S. Xydis, C. Skouroumounis, K. Pekmestzi, D. Soudris, and G. Economakos. Efficient High Level Synthesis Exploration Methodology Combining Exhaustive and Gradient-Based Pruned Searching. In *ISVLSI*, pages 104–109, July 2010.

[86] S. Xydis, G. Palermo, V. Zaccaria, and C. Silvano. Spirit: Spectral-aware pareto iterative refinement optimization for supervised high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34 (1):155–159, Jan 2015. ISSN 0278-0070. doi: 10.1109/TCAD.2014.2363392.

[87] Alexandre V Yakovlev and Albert M Koelmans. Petri nets and digital hardware design. In *Lectures on Petri Nets II: Applications*, pages 154–236. Springer, 1998.

[88] Yuan Zhou, Udit Gupta, Steve Dai, Ritchie Zhao, Nitish Srivastava, Hanchen Jin, Joseph Featherston, Yi-Hsiang Lai, Gai Liu, Gustavo Angarita Velasquez, et al. Rosetta: A realistic high-level synthesis benchmark suite for software programmable fpgas. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 269–278. ACM, 2018.

[89] Eckart Zitzler, Lothar Thiele, Marco Laumanns, Carlos M Fonseca, and Viviane Grunert Da Fonseca. Performance assessment of multiobjective optimizers: an analysis and review. *IEEE transactions on evolutionary computation*, 7(2): 117–132, 2003.