EFFECTIVE HIGH-LEVEL SYNTHESIS DESIGN SPACE EXPLORATION THROUGH A NOVEL COST FUNCTION FORMULATION

by

Yiheng Gao

APPROVED BY SUPERVISORY COMMITTEE:

____

Benjamin Schaefer, Chair

____

Mehrdad Nourani

____

Yang Hu

Copyright © 2020

Yiheng Gao

All rights reserved

Dedicated to my family and teachers.

EFFECTIVE HIGH-LEVEL SYNTHESIS DESIGN SPACE EXPLORATION THROUGH A NOVEL COST FUNCTION FORMULATION

by

YIHENG GAO, BE

THESIS

Presented to the Faculty of

The University of Texas at Dallas

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER IN

COMPUTER ENGINEERING

THE UNIVERSITY OF TEXAS AT DALLAS

August 2020

ACKNOWLEDGMENTS

I would like to thank my thesis advisor, Dr. Benjamin Carrion Schafer, for his constant support, help and faith in my research work. His dedication towards his research inspired and motivated me to do more in my research work. He always helped me and guided me to the right direction when I made mistakes in my research. He always encouraged me to learn more, and I appreciate his patience and enthusiasm in helping and guiding the students under his supervision. Thank you, Professor, for your faith in me and my work.

I would also like to thank Dr. Mehrdad Nourani and Dr. Yang Hu for being a part of my committee and reviewing my work. I would like to thank the Electrical Engineering Department and Design Automation and Reconfigurable Computing Lab (DARCLab) for creating an inspiring research environment and providing me with the necessary resources. I would like to heartily thank all my lab mates, Jianqi, Zi, Bo and Zhiqi for their support and help. I would like thank the Office of Graduate Studies for reviewing my work and giving their suggestions. My humble thanks to my parents who supported me throughout my studies and encouraged me to pursue my dreams. I would lastly like to thank Jianqi for his help, love and encouragement.

May 2020

EFFECTIVE HIGH-LEVEL SYNTHESIS DESIGN SPACE EXPLORATION THROUGH A NOVEL COST FUNCTION FORMULATION

Yiheng Gao, MS
The University of Texas at Dallas, 2020

Supervising Professor: Benjamin Schaefer, Chair

In the last few decades, Integrated Circuits (IC) designers have had to manually translate behavioral description into Register-Transfer Level (RTL) code (e.g. Verilog or VHDL). High-Level Synthesis (HLS) automates this process. HLS has many advantages as compared to specifying hardware at the RT-Level. One big advantages is that the behavioral description only needs to be designed and verified once, but allows to generate RTLs with different characteristics by simply specifying different synthesis options. This opens the door to perform a fully automatic Design Space Exploration (DSE). The main goal in HLS DSE is to find Pareto-optimal micro-architectures for the given untimed behavioral description. For large untimed descriptions an exhaustive enumeration of all possible synthesis options combinations is not possible, hence heuristics are required. This work presents three meta-heuristic algorithms to address this issue: Simulated Annealing (SA), Genetic Algorithm (GA) and Ant Colony Optimization (ACO). These algorithms are originally used to solve Single-Objective (SO) problems whereas DSE is Multi-Objective (MO), i.e. area vs. performance. To convert the MO problem into a SO, this work proposes a new method called ξ-constraint to do the conversion, and compares the result with the traditional method (weighted sum as cost function) for all three algorithms.

TABLE OF CONTENTS

ACKNOWLEDGMENTS . . . v

ABSTRACT . . . vi

LIST OF FIGURES . . . ix

LIST OF TABLES . . . x

LIST OF ABBREVIATIONS . . . xi

CHAPTER 1 INTRODUCTION . . . 1

CHAPTER 2 HIGH-LEVEL SYNTHESIS . . . 3

2.1 Overview . . . 3

2.2 History of HLS Tools . . . 4

2.3 HLS Process . . . 5

2.4 Benefits of HLS . . . 9

CHAPTER 3 DESIGN SPACE EXPLORATION . . . 12

3.1 Overview . . . 12

3.2 Knobs in DSE . . . 13

3.3 Challenge in DSE . . . 15

3.4 Previous Work in DSE . . . 16

3.5 DSE Metrics . . . 18

CHAPTER 4 META-HEURISTIC METHODS . . . 20

4.1 Simulated Annealing . . . 20

4.1.1 Overview . . . 20

4.1.2 SO-SA . . . 21

4.1.3 Weighted Sum SA . . . 23

4.1.4 ξ-Constraint SA . . . 25

4.1.5 Parameters in SA . . . 27

4.2 Genetic Algorithm . . . 27

4.2.1 Overview . . . 27

4.2.2 SO-GA 30
4.2.3 Weighted Sum GA 32
4.2.4 ξ-Constraint GA 33
4.2.5 Parameters in GA 34
4.3 Ant Colony Optimization 35
4.3.1 Overview 35
4.3.2 SO-ACO 36
4.3.3 Weighted Sum ACO 39
4.3.4 ξ-Constraint ACO 40
4.3.5 Parameters in ACO 40
CHAPTER 5 EXPERIMENTAL RESULTS 42
5.1 Experimental Setup 42
5.2 Results and Analysis 44
5.2.1 Weighted Sum SA vs. ξ-Constraint SA 45
5.2.2 Weighted Sum GA vs. ξ-Constraint GA 46
5.2.3 Weighted Sum ACO vs. ξ-Constraint ACO 48
5.2.4 SA vs. GA vs. ACO 49
CHAPTER 6 CONCLUSION AND FUTURE WORK 50
6.1 Conclusion 50
6.2 Future Work 51
REFERENCES 52
BIOGRAPHICAL SKETCH 54
CURRICULUM VITAE

LIST OF FIGURES

2.1 (a) RTL design flow vs. (b) HLS design flow. 4

2.2 Allocation, scheduling and binding in high-level synthesis. 6

2.3 Synthesis result: datapath and FSM. 8

2.4 Number of gates vs. number of lines of C/Verilog code. [16] 10

3.1 Knobs Overview in DSE. [14] 13

3.2 Ave16 Source Code 14

3.3 DSE flows overview proposed to date (a) synthesis based, (b) supervised learning based and (c) graph based [14] 17

3.4 Distance between d_{Nk} and d_{M1} 19

4.1 Solution Representation in Search Space 21

4.2 Escape from Local Minimum Solution 23

4.3 Pareto Front with Different α 24

4.4 General Idea for Genetic Algorithm 29

4.5 Gene, Individual Representation for ave16 31

4.6 Reproduction Process 32

4.7 Graphs for ave16 Benchmark 36

5.1 Experimental Setup 43

LIST OF TABLES

3.1 Attributes Used 15
5.1 Benchmark Overview 43
5.2 Parameters Setting 44
5.3 Selected Parameter Value 45
5.4 ξ-Constraint SA vs. Weighted Sum SA 46
5.5 ξ-Constraint GA vs. Weighted Sum GA 47
5.6 ACO Result 48
5.7 Results Comparison among SA, GA and ACO 49

LIST OF ABBREVIATIONS

ACO	Ant Colony Optimization
ADRS	Average Distance from Reference Set
ALAP	As Late As Possible
ASAP	As Soon As Possible
ASIC	Application Specific Integrated Circuit
BF	Brute Force Algorithm
CDFG	Control Data Flow Graph
CWB	CyberWorkBench
DSE	Design Space Exploration
EDA	Electronic Design Automation
FCNT	Functional Unit Constraint File
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
FU	Functional Unit
GA	Genetic Algorithm
HDL	Hardware Description Language
HLS	High-Level Synthesis
IC	Integrated Circuit
MO	Multi Objective
QoR	Quality of Result
RTL	Register-Transfer Level
SA	Simulated Annealing
SO	Single Objective
SoC	System on Chip

CHAPTER 1

INTRODUCTION

The advancement of technology has also forced to raise the level of VLSI design abstraction from the register-transfer level (RTL) to the behavioral level. This new abstraction level allows IC designers to implement their hardware design using high-level language such as C/C++ instead of Hardware Description Language (HDL) such as Verilog and VHDL. This method dramatically increases the design efficiency reducing the Turn around Time (TAT). Typically system architecture designers or algorithm designers develop their design as a behavioral description which then needs to be converted by the VLSI IC designer engineers into an efficient hardware design that can implement that description. This is a tedious, error-prone work that heavily depends on the individual skill of the VLSI designer.

One of the biggest advantage for HLS is that it allows generation of different micro-architectures (designs) without changing the source code. This process is called Design Space Exploration (DSE) and can be done by automatically by changing different types of synthesis options (called knobs). With a proper heuristic algorithm, the DSE process can return a set of Pareto-optimal Designs that from the Pareto-front such that each design in Pareto-front is not dominated by any other design. The simplest way to address HLS DSE is by enumerating all possible synthesis options (exhaustive search). This exhaustive search method is guaranteed to find the optimal solution. Unfortunately, the complexity of the problems grows exponentially with the number of synthesis options, and hence, an exhaustive search is not possible for larger designs. Hence, some intelligent heuristics are needed.

This work implemented several well-known meta-heuristic algorithms to solve HLS DSE problem and compare their quality or results. In summary, the contribution of this work are:

- Implement multiple meta-heuristic algorithms which includes Simulated Annealing (SA), Genetic Algorithm (GA) and Ant Colony Optimization (ACO) and investigate if any of these meta-heuristcs is superior than the others for HLS DSE.

- Propose a new method called ξ-constraint to improve the quality of Pareto-front and apply it to all three meta-heuristics comparing the results with a traditional method (Weighted sum as cost function).

- Provides comprehensive experimental results to validate that the proposed method leads to better results.

CHAPTER 2

HIGH-LEVEL SYNTHESIS

2.1 Overview

Integrated circuits(ICs) with billions of transistors are commonly found in daily used electronic devices. The design and verification of these complex ICs has become extremely hard. To address this, the industry has started to raise the level of VLSI design abstraction from the RT-Level to the behavioral level. This methodology shift improves the productivity and reduces the TAT.

At the very beginning ICs were designed at transistor level, which required designers to draw the layout of every transistor in the chip. As technology advanced, the size of transistor became smaller and the number of transistors per chip increased dramatically. This increased the complexity of IC design and made transistor-level design impractical. This, digital IC designers raised the abstraction level and started working on gate level. In this level, digital IC designers focused on designing logic gates and used layout tools to generate the layout of all transistors. Later on, in 1980s the semiconductor industry began to adopt Register-Transfer Level (RTL) synthesis [6], where the designers wrote the RTL code in hardware description languages (HDL) (e.g. Verilog or VHDL),and synthesized it with logic synthesis tools (e.g. Synopsys Design Compiler) and get the gate netlist. In RTL, designers do not need to know about every gate in the circuit, only the detailed micro-architecture needs to be specified with a HDL.

Finally, the industry is moving one step further by synthesizing untimed behavioral description, e.g. ANSI-C or C++ to generate the RTL code automatically [13]. In the traditional RTL design flow, as shown in Fig. 2.1(a), the RTL code of the circuit is obtained by translating the behavioural model manually. This manual translation takes extremely long time even for highly skilled RTL designers. To improve the productivity, an automatic

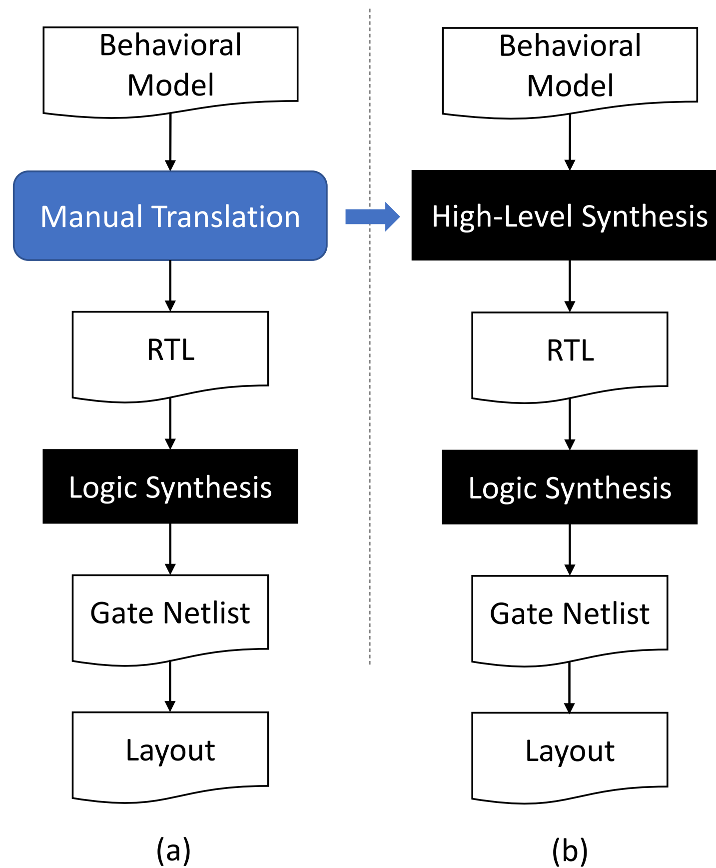

Figure 2.1. (a) RTL design flow vs. (b) HLS design flow.

C/C++ to RTL conversion method for IC/FPGA called high-level synthesis (HLS) was created. The overall new VLSI design flow using HLS is shown in Fig. 2.1(b), where the manual translation of the behavioral model to HDL is replaced by HLS. With the help of HLS, IC designers are able to only focus on the behavioral description of an algorithm and leave the implementation details to the synthesis tool.

2.2 History of HLS Tools

The earliest research work on HLS can be trace back to the 1970s, but the technology was mainly develoepd in the 1980s and 1990s. The major EDA companies also developed their first generation HLS tools in 1990s [6]. However, those works were commercial failure because

of domain specification, wrong input language, steep learning curve, poor quality of results, etc [6].

From the early 2000s, a new generation of commercial HLS tools were developed by many EDA companies that were gradually adopted by IC design companies. Most of the modern HLS tools use C/C++/SystemC as input language and produce better quality of results than its previous generation. Some popular commercial HLS tools include: Catapult [7] from Mentor Graphics, Stratus [1] from Cadence, Vivado HLS [18] from Xilinx, HLS compiler [5] from Intel and CyberWorkBench [10] from NEC. These HLS tools all accept C as input language, while some of them also use C++ and SystemC.

From the commercial HLS tools just mentioned, some mainly target the ASIC market while others mainly target FPGAs. The HLS tools from major EDA companies (Catapult, Stratus) are mainly focused on ASIC design, and the ones from FPGA venders (Vivado HLS, Intel HLS compiler) are designed for their FPGA devices. The High-level Synthesis tool used in this work is CyberWorkBench (CWB for short) [10] targets both ASICs and FPGAs.

2.3 HLS Process

HLS is a technique that automatically converts behavioral, untimed descriptions into efficient hardware that implements that behavior. More specifically, HLS tools typically take a behavioral description (C/C++/SystemC for example) as input and generate RTL code (Verilog/VHDL). The HLS flow is shown in Fig. 2.2. In particular, the input code is firstly parsed to create an intermediate representation, then allocation, scheduling and binding are performed to get the structural RTL design. After these steps, the Verilog/VHDL code is generated.

Besides the source code (behavioral description), HLS tool also needs a target synthesis frequency and functional unit (FU) library as inputs (Fig. 2.2). The target frequency will affect the scheduling process because it determines the maximum possible critical path delay

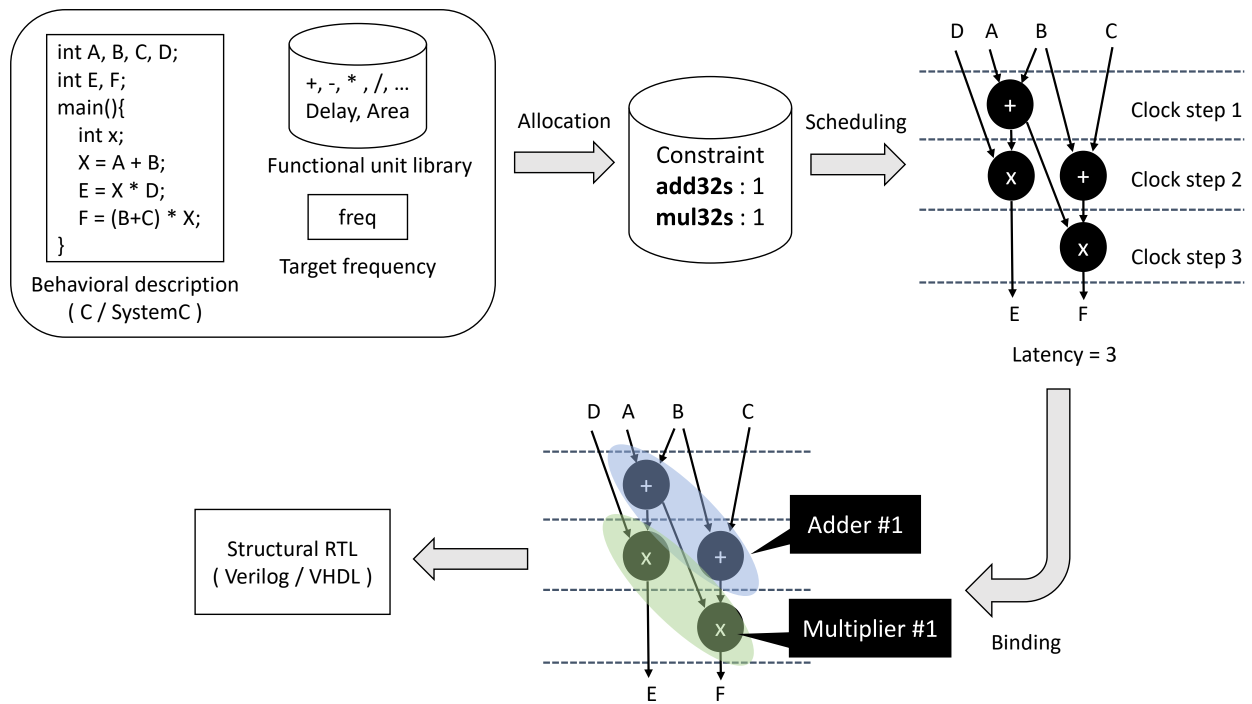

Figure 2.2. Allocation, scheduling and binding in high-level synthesis.

as the scheduling process will ensure the delay of all the paths does not exceed the clock period. The FU library contains pre-characterized functional units like adders, subtractors, multipliers, comparators, etc.., for different bitwidths. Given a new ASIC technology or FPGA device, a new FU library needs to be characterized before executing the HLS process. For this purpose, commercial HLS tools typically provide a built-in library characterizer to automatically generate the synthesis libraries. This tool will synthesize different types of FUs with various bitwidth using a logic synthesis tools (e.g. Design Compiler), and record the area and delay of each FU.

The allocation, scheduling and binding process are performed once the design has been parsed. Allocation specify the necessary hardware resources such as adders, multipliers, etc. Scheduling assigns every operation into a clock step based on its timing constraint. Binding provides a mapping from each operation to a specific FU and from each variable to a register.

Allocation: During the allocation stage, all the operations in the behavioral code are found to form a data flow graph (DFG). For example, in Fig. 2.2 there are 2 additions and 2 multiplications in the DFG for the given C code. The DFG shows the maximum possible FU usage (2 adders and 2 multipliers in the example) and contains no timing information. A FU constraint file (FCNT) will be generated after allocation, in which the maximum possible number of each FU is specified. Users can later modify the FU constraints which will lead to a different scheduling result. As shown in Fig. 2.2 the FU constraint file is configured such that both the number of adders and the number of multipliers in the final circuit cannot exceed 1.

Scheduling: The scheduling step decides in which clock step every operation is executed. The edges in the DFG indicate the data dependencies. An operation cannot be placed in the clock steps before its input data is ready. If two connected operations are in different clock step, the data will be stored in a registers, while operations that can be executed in the same clock cyles will lead to FU chaining. The chain of FUs has to respect the fact that path delay is smaller than the target clock period. Moreover, the number of a certain kind of operation in a clock step cannot exceed the limit specified in the FU constraint file, because a given FU can only be assigned to one operation in a clock step, while it can be assigned to multiple operations in different clock steps through FU sharing.

There are many scheduling algorithms, among which As Soon As Possible (ASAP) and As Late As Possible (ALAP) are the simplest solutions. One main goal of most scheduling algorithms is to shorten the latency, which is the total number of clock steps. In the example shown in Fig. 2.2 the latency of the synthesized circuit is 3.

Binding: The last step in the HLS process assigns the different operations in the scheduled DFG to specific FUs and registers in the constraint file. Different FU binding results in different bitwidth of FUs, different bitwidth and number of data inputs of multiplexers

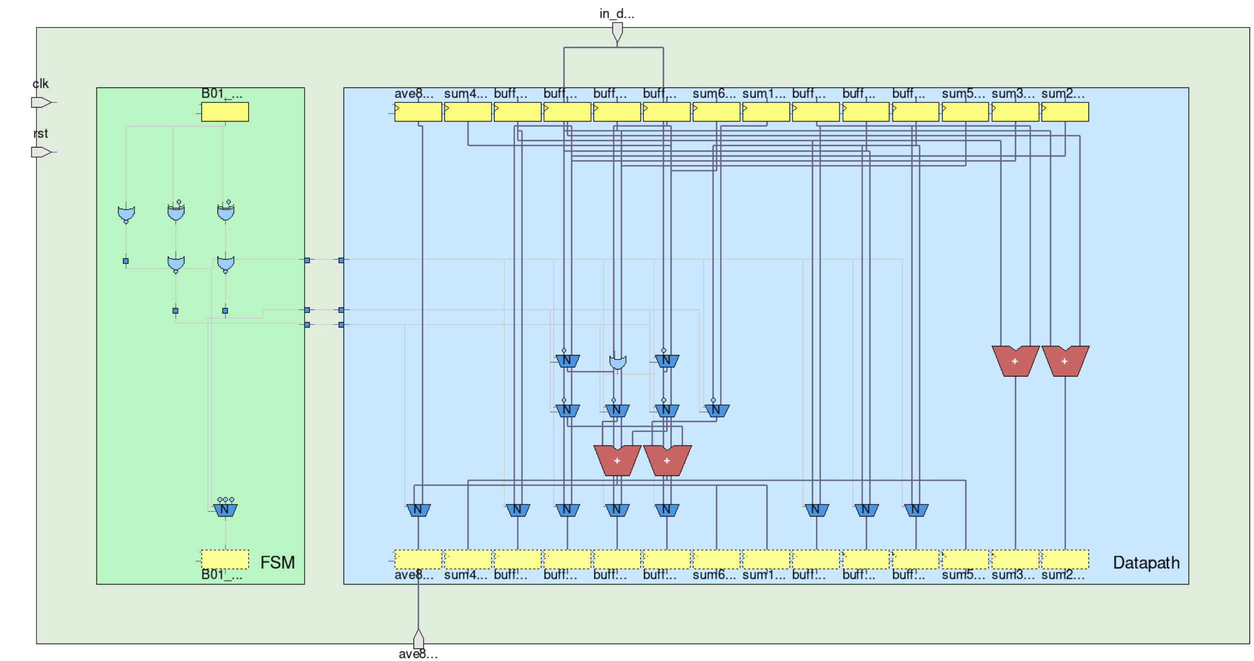

Figure 2.3. Synthesis result: datapath and FSM.

(MUXes), and hence different total area and critical path delay. There is only possible FU binding in the example shown in Fig. 2.2, however the number of possible FU bindings is very large in larger designs. One simple register binding method is to create a register for each edge that crosses clock steps in the DFG. In that case, most registers are not always in use in all clock steps, so these registers can be shared with the registers that do not have overlapping lifetimes. Finding the register binding that use the least number of registers can be reduced to graph coloring problem [8].

Generating RTL Design: After binding, the HDL code can be generated. This typically includes a datapath and a finite state machine (FSM), as shown in Fig. 2.3. The datapath is composed of FUs, MUXes, registers, etc., and takes in the primary inputs as well as generates the primary outputs. The MUXes in the datapath get control signals from the FSM to select correct data signals for the FUs and registers, and the FSM sends different control signals to the datapath in each of its states. It enables the datapath to accomplish

the task in every clock step of the DFG such that the data is *steered* correctly through all the FUs. The FSM state encoding affects the area and delay of the final circuit, for example binary or gray code saves the area of state register and logic in the FSM but increases delay as decoders are required to decode the state code, whereas one-hot encoding uses larger area but is faster.

2.4 Benefits of HLS

One of the biggest advantage of using HLS is the improvement of productivity. As Fig. 2.1(b) shows, with the help of HLS, the generation of RTL code becomes much easier. The adoption of HLS frees IC designers from manually translating behavioral description into HDL and thus, enables designers to work directly at the behavioral level.

In addition, HLS technology makes source code more compact and reduces the verification time. Every line of behavioral C/C++/SystemC code has been shown to generate, on average, 7× more gates than a single line of HDL code. As shown in Fig. 2.4, on average 4.2033 gates are generated per line for Verilog, while 31.306 gates are generated per line for C, which means on average C code is 7.4× smaller than Verilog code for the same number of gates [16].

HLS also makes verification much easier and faster. As the complexity of IC design increases, manual RTL coding tends to result in more errors. HLS allows designer to debug and verify designs at behavioral level instead of RTL level. To verify a design, HLS tools provide simulation environment of cycle accurate model, which is as accurate as HDL simulation while being more efficient. Take CWB as an example. According to [16], the cycle accurate simulation in CWB is 52 times faster than VCS for module simulation, and 234 times faster for entire SoC simulation [16].

Many fields can benefit from HLS. Hardware/software co-design is one example. Nowadays, almost every SoC include at least one embedded processors. In general, embedded

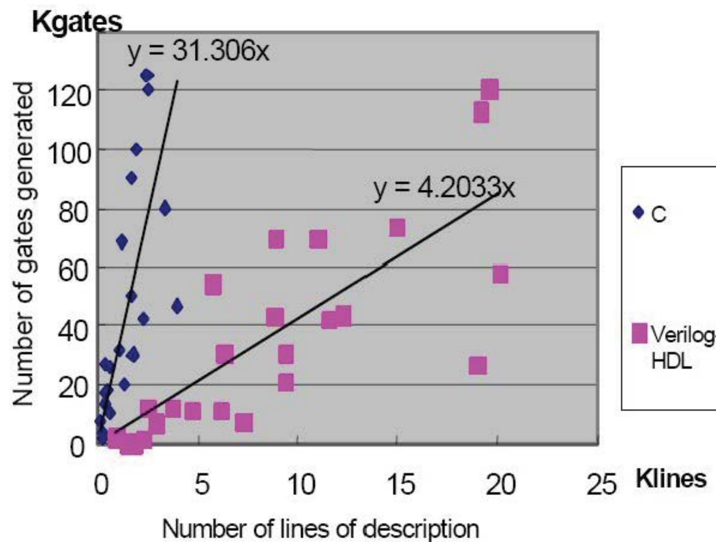

Figure 2.4. Number of gates vs. number of lines of C/Verilog code. [16]

software runs on the processors typically written in C/C++. These high-level languages are also the input language of the HLS tools. This fact enables IC designers to speed up software/hardware co-design and co-verification through HLS, since they can model the whole embedded system in C/C++ and use HLS tools to map the computational intensive parts of any application onto dedicated hardware accelerators. Besides, designers can also quickly explore area/performance/power tradeoffs with different software/hardware boundary [2].

Another example that benefits from HLS is FPGA design. With the slow down of Moore's law and advances in heterogeneous computing, many companies (e.g. Microsoft and Amazon) have put FPGAs in their servers and data centers to improve performance and power consumption. Ideally, the software engineers working for those companies need be able to program for both processors and FPGAs, but it is often very hard for software engineers to work on FPGAs because RTL design is unnatural for them. On the contrary, HLS accepts C/C++ as input and generates RTL that can in turn be mapped to an FPGA. With the help from HLS tools, software engineers can now map their algorithm to FPGA without a solid knowledge of hardware.

One more advantage, which is more related to this work, is that HLS makes automatic micro-architectural design space exploration (DSE) possible. Most HLS tools have the ability to generate a large number of different RTL designs from the same behavioral functionality by simply setting different synthesis options. By quickly exploring a design space, HLS enables the generation of Pareto-optimal designs with different trade-offs (Area vs. Latency as an example).

Compared with a traditional RTL design flow, HLS expands the search space so that more designs can be explored. The higher level of abstraction is, the larger the design space is. At gate level, when a fixed gate netlist is given, the only thing designers can tune is in the placement and routing of the design. At RTL, given a RTL design, the different logic synthesis options lead to gate netlists with different area/delay/power trade-offs, but the mirco-architecture remains the same. At behavioral level, more options can be choosen. With different target frequencies, functional unit constraints and attributes HLS can generate different micro-architectures with various area/latency/delay/power. For example, designers can specify whether the arrays are synthesized as registers or RAM, loops unrolled or not, etc. By changing different options, HLS can automatically explore different micro-architectures, which is impossible for RTL designers because it requires to conduct manually writing tens of thousands of RTL designs.

CHAPTER 3

DESIGN SPACE EXPLORATION

3.1 Overview

HLS DSE is a technique to explore different micro-architectures without modifying behavioral descriptions. These different micro-architectures (also known as designs) have different area (A) and latencies (L). Ideally, an IC designer would like to make the area (A) as small as possible and the latency (L), in clock cycles, also as low as possible [14]. These objectives are nevertheless conflicting, which means that reducing one increase of other conflicting parameter. This fact makes DSE a typical Multi-Objective (MO) Optimization Problem. Namely, in DSE, there is no unique design that leads to minimizing all of the objectives; rather, there is a set of Pareto-optimal designs that can be presented to users to choose from. These acceptable designs form a trade-off curve called Pareto frontier. The purpose for DSE problem is to discover the entire Pareto frontier[14]:

$$\textbf{Find} \quad P = \{d_1, d_2, \cdots, d_n\} \tag{3.1}$$

such that any design $d_i \in P$ cannot be dominated by others.

This work mainly focus on two parameters: A and L. These parameters represent the cost and speed of an IC respectively and are conflicting. Decreasing latency may requires more functional units to be placed on a chip, thus increase the area of the IC. Considering A and L, the set of dominating design P can be defined as follows. For any $d_i = (A_i, L_i) \in P, \forall d_j = (A_j, L_j)$ found in DSE satisfies:

$$\begin{cases} A(i) \leq A(j) \\ L(i) \leq L(j) \end{cases} \tag{3.2}$$

3.2 Knobs in DSE

As stated in Chapter 2, HLS allows to generate different designs without modifying behavioral description by setting different synthesis options. These synthesis options are also known as knobs (K). There are mainly three different exploration knobs: global synthesis options (K_{opt}), function unit constraint (K_{fus}) and local synthesis directives (K_{attr})[14]. Combining these knobs leads to a more extensive design space. Fig. 3.1 shows how the different knobs (K) impact with DSE.

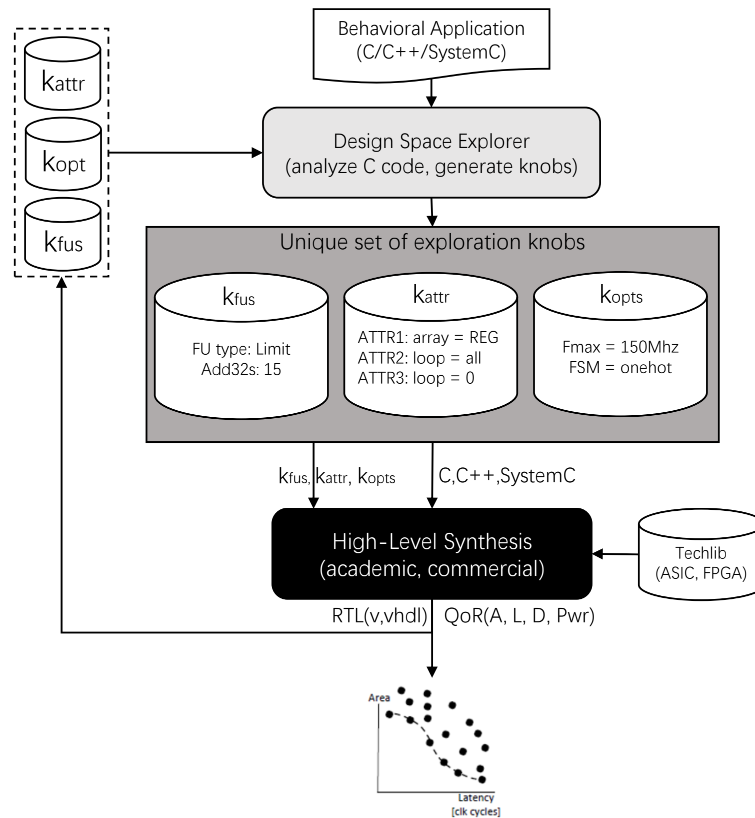

Figure 3.1. Knobs Overview in DSE. [14]

```
int data[16]    /* Cyber array = REG*/;
/* Cyber function = inline */
int ave16(int data_new){
   int sum = 0,x;
   /* Cyber unroll = all */
   for(x=15;x>0;x--)
      data[x] = data[x-1];
   data[0] = data_new;
   /* Cyber unroll = 0 */
   for(x=0;x<16;x++)
      sum = sum + data[x];
   return(sum/16);
}
```

Figure 3.2. Ave16 Source Code

Global Synthesis Options (K_{opts}). This knob usually specified when the HLS tool is invoked serves as a global options to be applied to the entire behavioral description. Take loop as an example, in K_{opts}, all loops in source code must be rolled or unrolled. Most of the time, however, the designer might want to unroll some of the loops and not others. Hence, Global Synthesis Options might not be flexible enough for some applications.

Function Unit Constraint (K_{fus}). K_{fus} affects the synthesis result by restricting the maximum number of FUs. A widely used optimization method called resource sharing[12] can be applied to reduce area[14]. In resource sharing, FUs are shared among different operations executed in different clock steps. This method is more efficient in Application Specific Intergrated Circuits (ASICs) while resource sharing typically result in larger synthesis result in FPGA due to the extra costs of the muxes required to share a single FUs [4].

Local Synthesis Directives (K_{attr}). Unlike K_{opts}, local synthesis directives are attribute options specified as pragmas directlty at the behavioral source code. These pragmas tell the HLS tools how to synthesize different operations. For example, the code snippet in Fig. 3.2 calculates average of 16 numbers and includes two for-loops. With K_{opts}, all loops

Table 3.1. Attributes Used

Type	Attribute	Description
Loop	unroll=0	do not unroll the loop
	unroll=all	unroll the loop completely
	unroll=x	unroll the loop x times
Function	func=inline	inline the function
	func=goto	function only instantiated once
Array	array=REG	map arrays to register
	array=RAM	implemented as RAM
	array=ROM	implemented as ROM
	array=EXPAND	expand arrays to variables
	array=LOGIC	implemented as logic

must be unrolled or not unrolled. In contrast, K_{attr} enables loop 1 to be completely unrolled while loop 2 not unrolled through these pragmas. This knob, hence, allows more finer controlability. Most commercial HLS tools make extensive use of these pragmas. CWB, e.g. has over 200 different attributes[13]. Some attributes are used in this work and summarized as Tab. 3.1. This work mainly focus on these pragmas for the DSE.

3.3 Challenge in DSE

The most simplest and straightforward algorithm to find the Pareto-optimal designs that form the Pareto-front is a brute force enumeration of all combinations (BF). This is an exhaustive search method which means that it is required to synthesize every possible micro-architecture based on different attribute combination (also known as solution). It then compares the cost of each design, and finally reports the Pareto-optimal design. The time complexity of BF is

$$f(t) = O(M^N) \tag{3.3}$$

where M is number of all possible attribute value for a pragma, N is total number of pragmas inserted in the source code.

As the complexity of behavioral description increases, the total number of lines of source code increases as well, which leads to increase of number of pragmas and number of possible attribute values. From (3.3) we can see that the running time of BF increases exponentially when M, N increases. This makes an exhaustive search algorithm impossible within a reasonable time. Therefore, new methods based on heuristics are needed. This work studies the use of some well-known meta-heuristic algorithms to explore the search space efficiently.

3.4 Previous Work in DSE

DSE techniques proposed so far can be classified into two different categories: synthesis based and model based[14]. These can be further divided into four subcategories: Meta-heuristics, Dedicated Heuristics, Supervised Learning and Graph Based[14]. The authors in [14] reviewed the state-of-the-art techniques in this domain.

Meta-heuristics such as SA, GA and ACO are often naturally inspired and typically lead to a very good resul. Dedicated heuristics are methods that have been developed to specifically addresses the uniqueness of the DSE optimization problem [14]. Both of them belong to synthesis based techniques which generate knob settings and then call the HLS tool in order to evaluate the effect of these new settings on the qualify of results (QoR). Fig. 3.3(a) shows the flow of this kind of DSE techniques. The program first reads the source code and extracts the operations that can be controlled through pragmas (mainly loops, arrays and functions). Then it generates a particular knob setting combination, calls the HLS tool and evaluates the QoR for this unique combination. Based on the results, the program modifies the knob settings such that the synthesized design moves toward the one with minimum cost. These steps continue until a given exit condition occurs.

Supervised learning method is the mixture of synthesis and model based and therefore can be divided into two steps. The first step is similar to the synthesis based techniques. The explorer parses the behavioral code, generates a certain number of knob settings, synthesizing

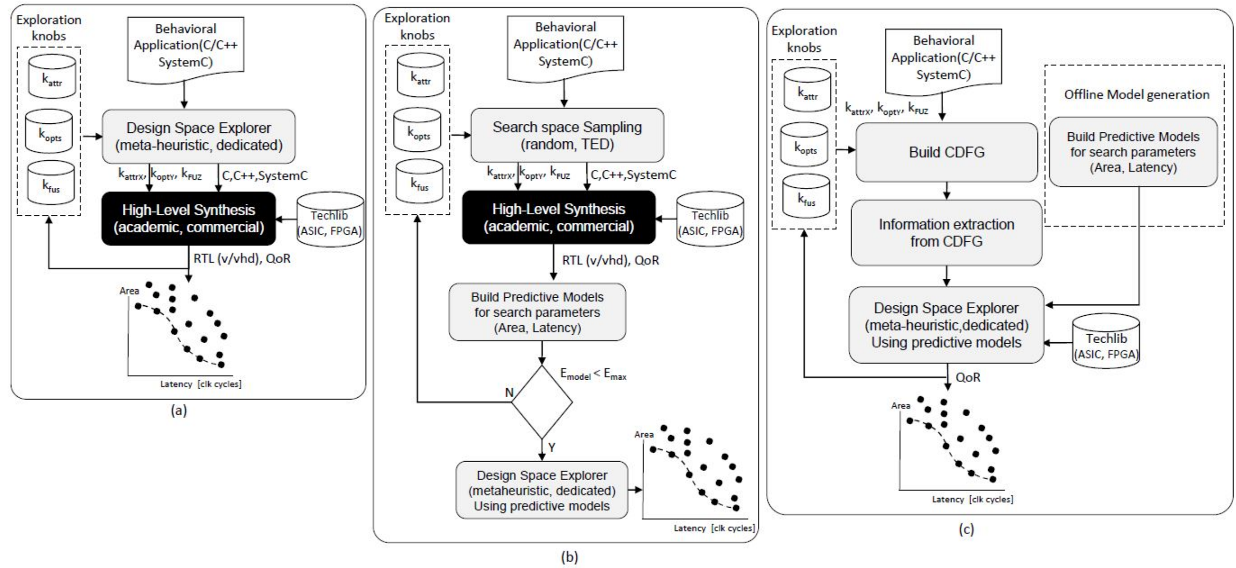

Figure 3.3. DSE flows overview proposed to date (a) synthesis based, (b) supervised learning based and (c) graph based [14]

these by calling HLS tool and continuously builds a predictive model based on the synthesis result until the model is stable (the error between the model and the synthesized designs is smaller than a certain maximum error threshold)[14]. In the second step, the explorer does not invoke HLS tools. Instead, it uses the predictive model to evaluate new knob settings. Fig. 3.3(b) is the flow for supervised learning based method.

Fig. 3.3(c) is the typical flow of graph based method. This method never invokes HLS tool but builds a predictive model offline and use the model based on CDFG techniques. After parsing the behavioral description, the explorer builds a CDFG and continues to extract information from the graph. Combined these information and the predictive models, the explorer estimates the synthesis result. Then the search space can be explored using the predictive model instead of calling HLS tools.

3.5 DSE Metrics

To measure the quality of different algorithms, some quality indicators are needed. These indicators can be used to measure how good different methods are. In [19], the authors proposed Dominance and Average Distance from Reference Set (ADRS) as indicators.

Dominance. This indicator is calculated as follows:

$$Dominance = \frac{P_M \cap P_N}{P_N} \tag{3.4}$$

where P_M is the Pareto-optimal designs obtained from the heuristic algorithm, P_N is the Pareto-optimal designs from the base line. This formula calculates the ratio between the total number of Pareto-optimal designs found by a heuristic algorithm given in reference Pareto set. The higher the value is the better results a heuristic algorithm gets. For small benchmarks, the reference set can be generated by a BF method, whereas for large benchmark this is typically obtained by combining the results from all the heuristic methods.

ADRS. This indicator measures how close a Pareto set obtained by a heuristic algorithm is to the reference set. Heuristic algorithms enable HLS tools to generate a trade-off curve within acceptable time. The designs on this trade-off curve might nevertheless not be the Pareto-optimal designs. Let the reference Pareto set $P_N = \{d_{N1}, d_{N2}, \cdots, d_{Ni}\}$, Pareto frontier under evaluation $P_M = \{d_{M1}, d_{M2}, \cdots, d_{Mj}\}$, ADRS is calculated as follows:

$$ADRS = \frac{\sum_{k=1}^{j} Dist(d_{Mk})}{|P_M|} \tag{3.5}$$

where $Dist(d_{Mk})$ is the minimum distance for design d_{Mk} to all reference points as Fig. 3.4 shows. This indicator allows designers to measure how good these results are. The smaller the value is, the better. $ADRS = 0$ indicates $P_M = P_N$ which is what designers expected.

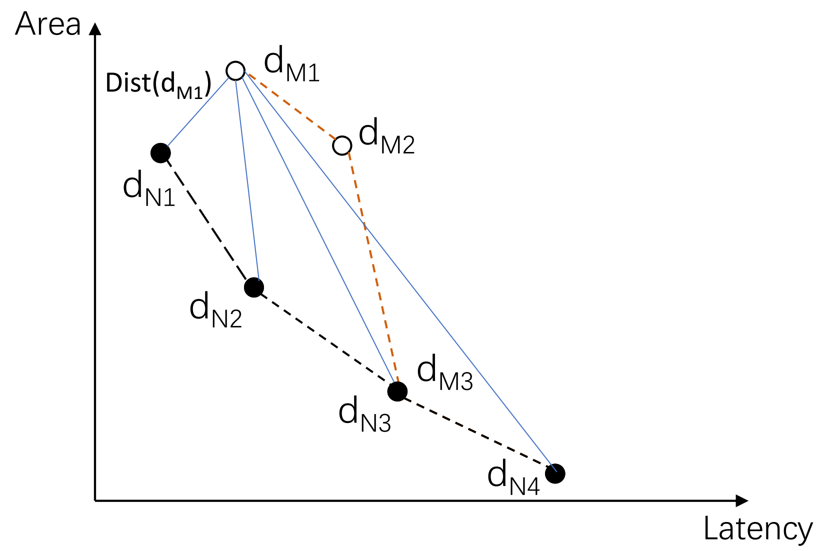

Figure 3.4. Distance between d_{Nk} and d_{M1}

CHAPTER 4

META-HEURISTIC METHODS

4.1 Simulated Annealing

4.1.1 Overview

Simulated Annealing (SA) is a meta-heuristic algorithm that is based on a probabilistic technique which is widely applied in searching approximate optimal solution in an extremely large search space within a reasonable time. It is inspired from the annealing process in metallurgy, a technology that heats material to extremely high temperature followed by the slowly cooling down to increase the size of material's crystal and reduce the defects. These two attributes are both related with thermodynamic free energy of atoms in material. The theory behind annealing process is the fact that atoms in material will stay in place where free energy is local minimum. By heating the material, atoms will leave their original position and move randomly to other positions. When the cooling rate is slow, atoms are more likely to find positions that have lower free energy than its original position. Because atoms finally find a position with lower thermodynamic free energy, the size of material's crystal and defect rate is improved.

In SA, similar to the annealing phenomenon in metallurgy, each point in a large search space can be imagined as an atom, these points have its thermodynamic free energy which is called `cost` and can be calculated by `cost function`. The algorithm starts at a random position. At each step, SA consider some neighboring state S_n of the current state S_c and decide whether move the system to S_n or staying in S_c. This step repeated until termination condition occurs. SA is typically used to find the minimum or maximum cost.

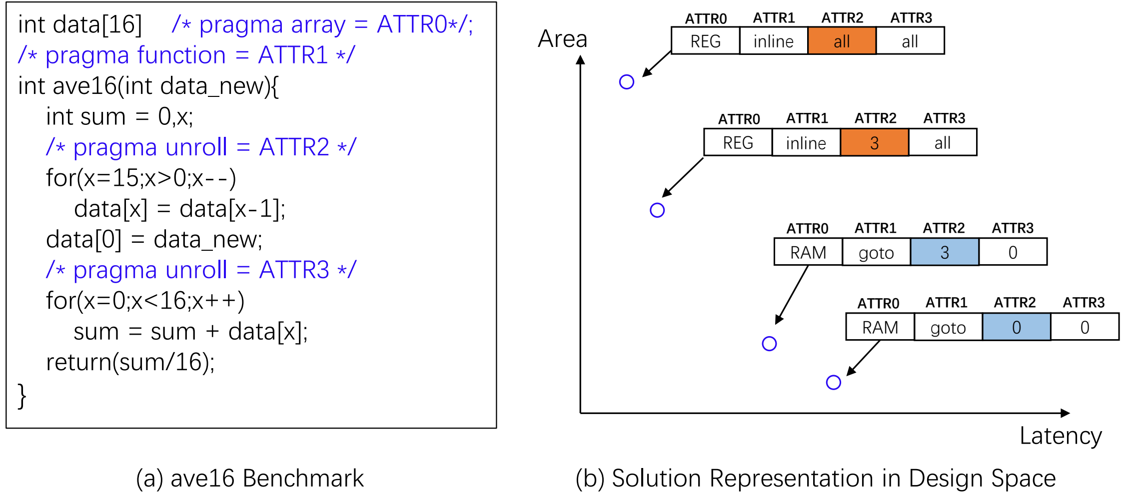

Figure 4.1. Solution Representation in Search Space

4.1.2 SO-SA

As mentioned in 4.1.1, SA is an algorithm to solve Single-Objective(SO) problems, which means searching the position in a system that has `minimum or maximum cost`. In the DSE casae, the target will be set as

$$\textbf{Minimize } Cost \tag{4.1}$$

where `Cost` is given by `Cost Function` which will be discussed in the following section. The state S is represented by a combination of values of attributes (also known as `solution`) as shown in Fig. 4.1.

Informally, SA works as follows. An initial temperature T_{ini} will be set as current temperature T_c, and a random solution will be generated. After synthesis, the cost of that solution can be calculated with `cost function`. At each iteration, the algorithm generates a neighbor solution according to `neighbours generating rule`, evaluates the neighbor cost and decides moving the system to the neighbor solution or staying in current solution according to `acceptance rule`. The process is then iterated with a reasonable `temperature reduced mechanism` until termination condition occurs. The pseudo-code for SO-SA is shown in Algorithm 1.

Algorithm 1: The SO-SA Algorithm

```
1  T_c ← T_ini
2  Generate a random solution
3  C_c ← cost of the random solution
4  while termination condition does not occur do
5  |  for i = 1 to n_T do
6  |  |  Generate a neighbor solution
7  |  |  C_n ← neighbor cost
8  |  |  ΔC = C_n − C_c
9  |  |  /* Acceptance Rule */
10 |  |  if ΔC < 0 or Random(0,1) < e^(−ΔC/T_c) then
11 |  |  |  C_c ← C_n
12 |  |  end
13 |  end
14 |  T_c ← γ · T_c
15 end
```

Neighbors Generating Rule. Solution in this work is defined as a set of attribute knobs (K_{attr}) in a behavior description as equation 4.2 indicates.

$$S = [Attr_1, Attr_2, \cdots, Attr_n] \tag{4.2}$$

where $Attr_i \in \Omega_i$, Ω_i is the set of all possible value for $Attr_i$. A neighbor is a new solution in which a certain number of attributes of the current solution is changed as Fig. 4.1(b) shows. The number of attributes changed ($N_{neighbor}$) when generating a neighbour solution is a parameter that can be modified by users.

Acceptance Rule. This rule is applied when the system decides if S_c moves to its neighbor solution S_n or stays at the current solution. Intuitively, if S_n has lower cost, the system should accept neighbor solution and vice versa. However, constantly accepting smaller cost solutions may cause the system to move toward a local optimal solution, not the global optimal. For example, as Fig. 4.2 shows, if a system starts at point A and only accepts smaller cost solutinos at every step, the system will finally move to point B, which is not the global minimum.

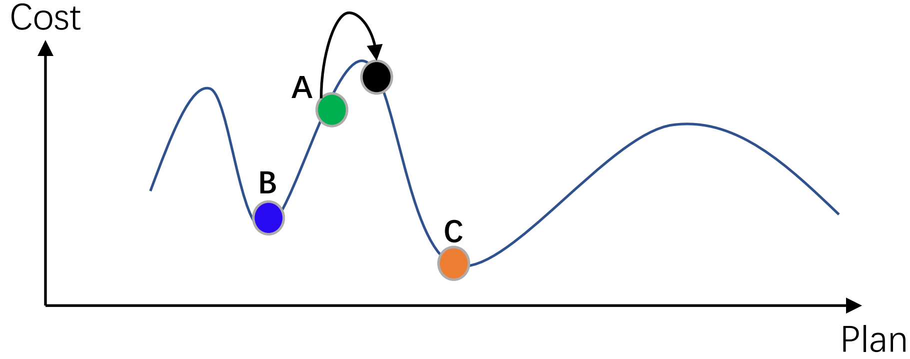

Figure 4.2. Escape from Local Minimum Solution

To overcome this problem, SA provide a mechanism call `escape possibility` to randomly accept some worse solution to allow a more extensive search for the global minimum solution. Similar with annealing in metallurgy, the probability is related with the system's current temperature T_c. As T_c decreased with time, the probability of accepting worse solution reduced as well. In this way, the result found by SA will converge to global minimum solution. The acceptance rule is given by equation 4.3.

$$S_c = \begin{cases} S_n, & \text{if } \Delta C < 0 \text{ or } Random(0,1) < e^{\frac{-\Delta C}{T_c}} \\ S_c, & \text{otherwise} \end{cases} \tag{4.3}$$

where $\Delta C = C_n - C_c$ is the difference between the cost of a neighbor solution C_n and the cost of the current solution C_c. T_c is the current temperature of a system that slowly reduces over time.

4.1.3 Weighted Sum SA

As mentioned in 4.1.1, SA is classified as SO algorithm. DSE problem, however, belongs to Multi Objective (MO) problem. The most important objectives of DSE are `Area` and `Latency` which reflect the size and the speed of a design respectively. Therefore, some methods should be applied to convert MO to SO so that the problem can be solved by

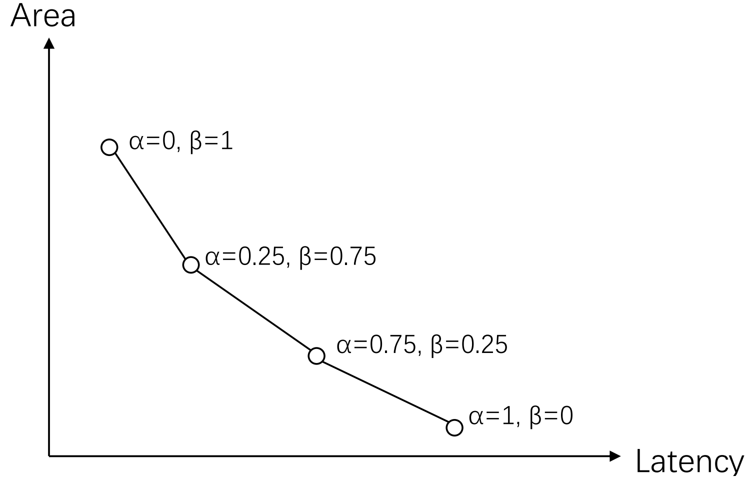

Figure 4.3. Pareto Front with Different α

repeatedly calling SA with different cost function. The authors in [9] developed a way to use SA to solve MO problem without converting MO into SO, but it lacks of mathematical proof. One traditional way to convert of MO into an SO function is aggregating all objectives in a weighted function [11] [15]. The simplest and most popular aggregation function is a linear combination of all objectives.

With weighted sum method, the `cost function` can then be set as

$$C(S) = \alpha \cdot \frac{Area}{A_{max}} + \beta \cdot \frac{Latency}{L_{max}} \tag{4.4}$$

where $\alpha + \beta = 1$, S is given in (4.2). $Area$, $Latency$ are divided by A_{max}, L_{max} respectively to unify different units. α, β indicate the relative importance of $Area$ and $Latency$. By assigning different α, SA is be able to find different compromised solution as Fig. 4.3 shows. The pseudo-code for `Weighted Sum SA` is shown in Algorithm 2.

This is a relatively simple technique. With each value of α, β, the problem is solved, and thus, a set of solutions is generated. However, this method does not always lead to a good solution, and also misses some dominating points. As Fig. 4.3 shows, usually Pareto-optimal designs on the trade-off curve are found when different cost functions are used, since

Algorithm 2: SA-Weighted Sum

```
1  α ← 1.0
2  β ← 0.0
3  while α ≥ 0 do
4      T_c ← T_ini
5      Generate a random solution S_c
6      C_c ← α · A_c/A_max + β · L_c/L_max
7      while termination condition does not occur do
8          for i = 1 to n_T do
9              S_n ←neighbor solution
10             C_n ← α · A_n/A_max + β · L_n/L_max
11             ΔC = C_n − C_c if ΔC < 0 or Random(0,1) < e^(−ΔC/T_c) then
12                 C_c ← C_n
13             end
14         end
15         T_c ← reduced temperature
16     end
17     α ← α −step
18     β = 1 − α
19 end
```

with fixed α and β SA searches torwards one direction on the area-latency plane by nature. There will be easily Pareto-optimal solutions missed if *step* for α and β is relatively large. Furthermore, considering the situation that the design space has two points $A_1 = 500$ and $A_2 = 100$ with the same Latency $L_{min} = 10$. when set $\alpha = 0$ and $\beta = 1$, the cost function C cannot guarantee to find the non-dominated point because these two points has the same cost but actually $P_2 = (100, 10)$ dominates $P_1 = (500, 10)$.

4.1.4 ξ-Constraint SA

To search Pareto-optimum solution more effectively and precisely, a new method called ξ-`Constraint` is proposed. This method solves MO problem by finding optimization solution with one object and setting other objectives as constraints bounded by some range ξ_i. By constantly shrinking the space which is bounded by ξ, MO problems can be solved more accurately.

Algorithm 3: SA-ξ-Constraint

```
1  L_constraint ← ∞
2  while L_constraint ≥ L_min do
3  |  T_c ← T_ini
4  |  Generate a random solution
5  |  C_c ← A_c
6  |  while termination condition does not occur do
7  |  |  for i = 1 to n_T do
8  |  |  |  S_n ←neighbour solution
9  |  |  |  C_n ← A_n if L_n < L_constraint, ∞ ortherwise
10 |  |  |  ΔC = C_n − C_c
11 |  |  |  if ΔC < 0 or Random(0,1) < e^(−ΔC/T_c) then
12 |  |  |  |  C_c ← C_n
13 |  |  |  end
14 |  |  end
15 |  |  T_c ← γ · T_c
16 |  end
17 |  (A_min, L_i) ← global minimum solution
18 |  L_constraint ← L_i
19 end
```

In ξ-`Constraint` SA, DSE problem is transformed as

$$\begin{aligned} &\textbf{Minimize } f_1(S) = A(S), \\ &\textbf{Subject to } f_2(S) = L(S) \leq \xi_i \end{aligned} \tag{4.5}$$

where S is given in (4.2). The cost function C is now simplified to

$$C = f(S) = Area \tag{4.6}$$

where S is given in (4.2).

Typically the ξ-`Constraint SA` works as follows. Initially, the latency constraint $L_{constraint}$ is set to ∞ so that SA can explore the whole design space to discover (A_{min}, L_1). After that, shrinking the space by setting $L_{constraint}$ to L_1 to discover (A_{min2}, L_2), where $A_{min2} > A_{min}$ and $L_2 < L_1$. The process then iterated until $L_{constraint} = L_{min}$. The pseudo-code for ξ-Constraint SA is reported in Algorithm 3.

4.1.5 Parameters in SA

In this work, four parameters are set in the SA process: n_T, γ, $N_{neighbour}$ and N_{exit}.

n_T, γ. The temperature reduction rate. This parameter will affect the quality of the SA result. If the temperature is dropped too quickly, SA might not have enough chance to move to desired solution, however, if the temperature decreases too slowly, the running time will increased dramatically. Therefore, the temperature is important in SA. There are two parameters related with temperature: n_T and γ. n_T indicates after exploring n_T neighbours, the current temperature will decrease by the factor γ

$$T_c \leftarrow \gamma \cdot T_c \tag{4.7}$$

$N_{neighbor}$. In HLS DSE, the most common and convenient knob to explore different micro-architectures is K_{attrs}. With the K_{attrs} knob, each solution is represented by a set of attributes specified in the source code in the form of pragmas. Exploring neighbours in SA implies changing one or more attributes every time. $N_{neighbor}$ is the number of attributes changed.

N_{exit}. This last parameter is the termination condition for SA. In HLS DSE , there is no obvious exit condition because the more solution are explored, the higher chance to a smaller cost solution. Therefore, SA will be terminated when no better solution is found after N_{exit} consecutive solutions.

4.2 Genetic Algorithm

4.2.1 Overview

Genetic Algorithm (GA) is a search based algorithm used to solve optimization problem in computational mathematics. It can be classified as a type of Evolutionary Algorithms.

These algorithms were originally motivated from some phenomena in evolutionary biology which includes cross-breeding, mutation, natural selection and hereditary.

GA is usually implemented by computer simulations. For an optimization problem, a certain number of candidate solutions, which are called chormesomes, can be abstracted and represented with a different combination of genes. An chromosome is typically expressed as a simple string, but other representations are also acceptable depending on a specific problem. This process is called coding. The algorithm initially generates a certain number of chromosomes as the first generation. In this step, algorithm designers occasionally intervene with chromosome generation to improve the quality of initial population. The next step for GA is generating next generation of chromosomes and forming a new population. This process is accomplished through reproduction and selection. The reproduction process includes crossover and mutation. The selection process selects chromosomes for breeding and is based on chromosome's fitness value which can be calculated by a given fitness function: the higher fitness value is more chance of being selected. This is a compromised strategy because simply selecting individuals with high fitness may cause precocity which means algorithm quickly converge to the local optimal solution instead of the global optimal solution. These selected chromosomes begin reproduction. Typically there is a parameter called crossover possibility, which represents the chance for selected chromosomes crossover their genes and generate offspring. The first half genes of offspring comes from one of parents while the second half genes of offspring inherited from the other parent. The half here is not the real half. It is determined by the crossover point randomly generated by algorithm. Mutation is introduced after crossover to provide a chance for offspring to obtain better fitness value. In general, a very small number (0.1 for example) will be set in this step to reflect the possibility of mutation. Based on the mutation possibility, offspring's chromosome is randomly mutated. This mutated gene is not from its parents. After a serial of reproduction process, generated new chromosomes constitutes a new generation which usually has better fitness values

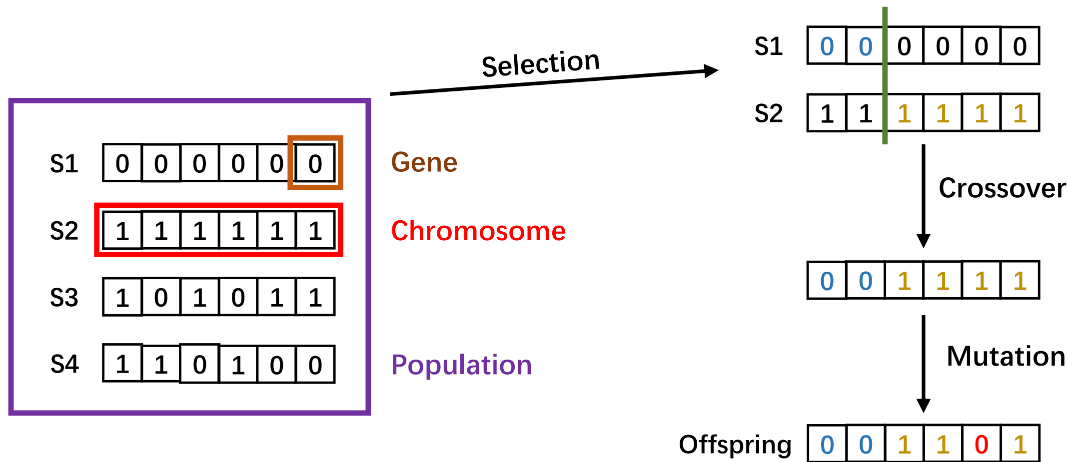

Figure 4.4. General Idea for Genetic Algorithm

than the original generation. By iteratively repeating the reproduction process until the termination condition occurs, generations developed toward the direction of improvement of overall fitness, because GA intends to select individuals with better fitness value to produce the next generation. Therefore, individuals with low fitness value are gradually eliminated. In general, termination condition can be set as one or some combinations of the following conditions:

- Reached maximum allowable number of generations.
- Exceed the maximum allowable time or memory usage.
- Continuing evolution does not produce individuals with better fitness value.

In summary, for implementation of GA solutions are traditionally represented in binary, which is a string of 0's and 1's as Fig. 4.4. The algorithm starts with a population that contains completely random chromosomes, evaluates the fitness with a specific fitness function of the initial population, selects multiple chromosomes from the current population based on the fitness value, and generates next generation of chromosomes through cross-breeding

and mutation. This generation then becomes the initial population of next iteration. GA iteratively executes that process to select individuals with better fitness value which is similar with nature selection in evolutionary biology.

4.2.2 SO-GA

Back to DSE problem, chromosomes can be represented by a series of knobs (K_{attr}) settings, all possible values for K_{attr} is its unique genes. Fig. 4.5 shows an example of chromosomes and genes for benchmark ave16. The target for the DSE problem is set to find the population that has the lowest cost C which calculated by the cost function (which serves as the fitness function mentioned above, but unlike fitness, cost is the smaller the better). The cost function depends on the method used. More details about cost function will be discussed in later section.

Informally, GA works as follows. The algorithm initialize an original population by generating several chromosomes with ramdom attribute values. RTL designs will be synthesized with the attributes in the chromosomes. The cost of the design will be calculated using its HLS report by the cost function. Then GA constantly select two chromosmes (designs) from original population as parents to reproduce until termination condition occurs. Parents generate offsprings after crossover and mutation process. The algorithm then calculated children's cost value and compared them with the parents'. If the children has better fitness value then replace the parents with the children to continue cross-breeding process until no better child generated for consecutive N_{pare} times. After that the parents are put into the population and the process continues. The pseudo-code is shown in Algorithm 4.

Breeding Rule. Similar with evolutionary biology that some individuals are not be able to breed, some individuals selected from population are not allowed to produce children by chance. This possibility is called crossover possibility (P_{cros}). Individuals can produce

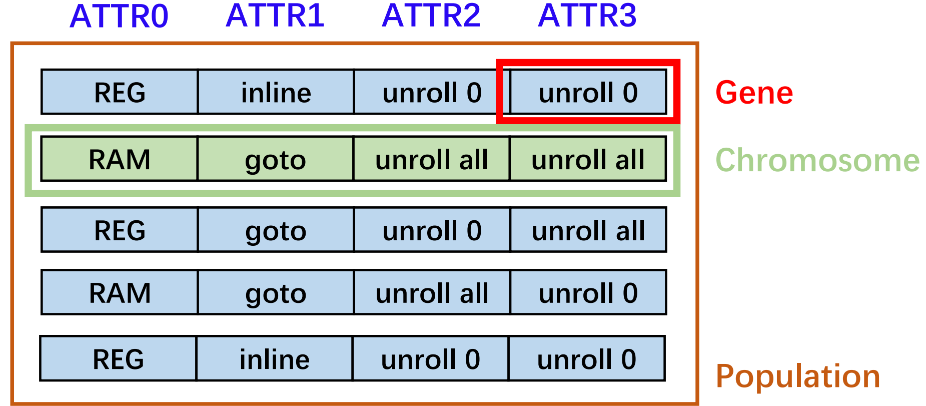

Figure 4.5. Gene, Individual Representation for ave16

Algorithm 4: The SO-GA Algorithm

```
1  Generate a random original population
2  while termination condition does not occur do
3      Select S_m, S_f randomly from population
4      C_m ← Cost(S_m), C_f ← Cost(S_f)
5      /* Breeding Rule */
6      while better child generated within consecutive N_pare iterations do
7          if random(0,1) < P_cros then
8              Randomly generate crossover point
9              S_c1, S_c2 ← Crossover and Mutate S_m, S_f
10             C_c1 ← Cost(S_c1), C_c2 ← Cost(S_c2)
11             if C_c1 or C_c2 better than C_m or C_f then
12                 Replace parents
13             end
14         end
15     end
16     Put S_m, S_f into population
17 end
```

offspring if and only if the following inequality is true:

$$Random(0,1) < P_{cros} \tag{4.8}$$

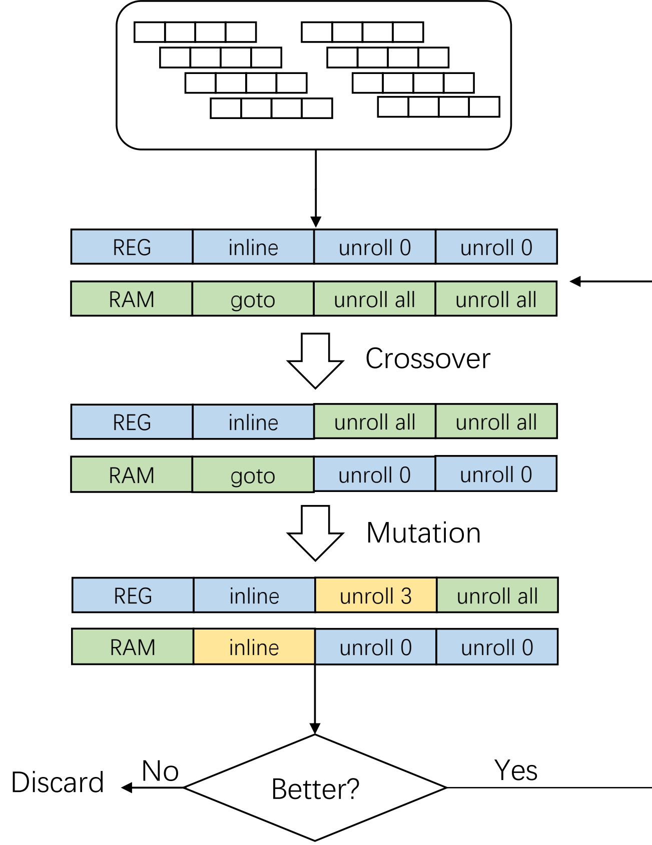

Figure 4.6. Reproduction Process

where $P_{cros} \in [0, 1]$ is a parameter. If the individuals are able to reproduce, the crossover point will be randomly generated and a very small mutation rate will be applied. Two children will be generated and evaluated their cost C. This breeding process is shown in Fig. 4.6.

4.2.3 Weighted Sum GA

Similar to Weighted Sum SA (Sec 4.1.2), traditional way to solve MO problems in GA is assigning every objective a weight and sum them up as cost function. In this work we only focus on A and L, therefore the cost function is given:

$$C(S) = \alpha \cdot \frac{Area}{A_{max}} + \beta \cdot \frac{Latency}{L_{max}} \tag{4.9}$$

where $\alpha + \beta = 1$, $\alpha \in [0, 1]$, $\beta \in [0, 1]$. The cost function dynamically changes every time SO-GA executed. Initially α. β are set 1.0, 0.0 respectively. After executing first round (SO-GA) the result with smallest area A is obtained. Then we reduce α, β to form a new cost function and get the population that has the lowest cost. These steps repeat until exit condition occurs. The pseudo code is shown in Algorithm 4.

The disadvantage for weighted sum GA is that without information of search space, it is hard to set the step for α and β. With inappropriate step setting, some Pareto points cannot be easily found by these α and β. For example, in small benchmark ave16, base line results are obtained with BF. By analysing the base line, we realized that there always has a Pareto point will be missed even the step is set to 0.1. Meanwhile, the step cannot be set too small because it will increase the computational burden for GA and hence increase the runtime. Under these circumstances, we applied ξ-constraint method to implement GA.

4.2.4 ξ-Constraint GA

ξ-Constraint is a different method from weighted sum to solve MO problem with SO algorithms. This method sets one objective as the target to optimize while other important objectives are used for setting boundary constraint ξ. More specifically, for the DSE problem we worked on, the cost function is set as

$$C(S) = A \tag{4.10}$$

whereas the latency obtained at each round will be set as constraint. At the beginning, the constraint ξ is set to ∞ such that SO-GA searches the whole design space to find out designs with (A_{min}, L_1). After first round, the constraint ξ is set to L_1 so that the search space shrinks. SO-GA search A_{min} in that shrunk space. This process continued until constraint hits the smallest search space, namely, ξ equals L_{min}. The pseudo-code for this method is shown in Algorithm 6.

Algorithm 5: Weighted Sum GA

```
1  α ← 1.0
2  β ← 0.0
3  while α ≥ 0 do
4      Generate original population
5      while termination condition does not occur do
6          Select S_m, S_f randomly from population
7          C_m ← Cost(S_m), C_f ← Cost(S_f)
8          /* Breeding Rule */
9          while better child generated within consecutive N_pare iterations do
10             if random(0,1) < P_cros then
11                 Randomly generate crossover point
12                 S_c1, S_c2 ← Crossover and Mutate S_m, S_f
13                 C_c1 ← α · A_n(S_c1) + β · L_n(S_c1)
14                 C_c2 ← α · A_n(S_c2) + β · L_n(S_c2)
15                 if C_c1 or C_c2 better than C_m or C_f then
16                     Replace parents
17                 end
18             end
19         end
20         Put S_m, S_f into population
21     end
22     α ← α − step
23     β = 1 − α
24 end
```

4.2.5 Parameters in GA

There are four parameters in GA: P_{cros}, R_{muta}, N_{pare} and N_{exit}. P_{cros} is the crossover possibility. In GA there exists the situation that parents cannot reproduce. This chance is determined by crossover possibility P_{cros}. R_{muta} is the mutation rate and should be set to a small number. N_{pare} indicates when to select new parents from population: no better design obtained after consecutive N_{pare} children. The termination condition is similar to SA: after consecutive N_{exit} parents being put in the population, the design with lowest cost has not been updated, then the GA algorithm ends and move into the next round.

Algorithm 6: ξ-Constraint GA

```
1  L_constraint ← ∞
2  while L_constraint ≥ L_min do
3      Generate original population
4      while termination condition does not occur do
5          Select S_m, S_f randomly from population
6          C_m ←Cost(S_m), C_f ←Cost(S_f)
7          /* Breeding Rule */
8          while better child generated within consecutive N_pare iterations do
9              if random(0,1) < P_cros then
10                 Randomly generate crossover point
11                 S_c1, S_c2 ← Crossover and Mutate S_m, S_f
12                 C_c1 ← A(S_c1) if L(S_c1) < L_constraint, ∞ otherwise
13                 C_c2 ← A(S_c2) if L(S_c2) < L_constraint, ∞ otherwise
14                 if C_c1 or C_c2 better than C_m or C_f then
15                     Replace parents
16                 end
17             end
18         end
19         Put S_m, S_f into population
20     end
21     (A_min, L_i) ← global minimum solution
22     L_constraint ← L_i
23 end
```

4.3 Ant Colony Optimization

4.3.1 Overview

Ant Colony Optimization(ACO) is a graph-based optimization algorithm inspired by the behavior of ants to find shortest path in the process of finding food. Researchers realized that the behavior of an ant is relatively simple, the ant colony as a whole, however, shows some sophisticated behaviors. For example, ant colonies can always find the shortest path to food sources in different environments. This is due to the fact that ants release a substance called pheromone on the way they passed. Ants have the ability to perceive pheromone and will select a path based on pheromone concentration: the more pheromone a path has, the

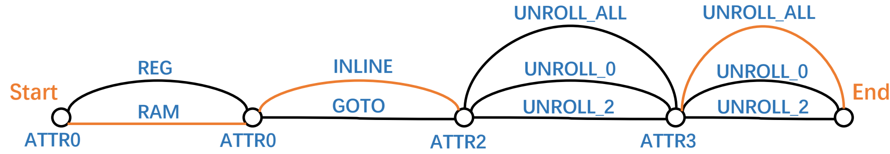

Figure 4.7. Graphs for ave16 Benchmark

more likely the ants will follow the path. Compared with longer path, shorter path would have more ants pass within a certain time. The more ants have passed a path, the higher concentration of pheromone the path has. This is a positive feedback mechanism. After a period of time, the entire ant colony will go along the path with highest pheromone which is the shortest path to the food source.

ACO was originally introduced by Dr.Marco Dorigo in his doctoral dissertation in 1992. It was used to solve Travelling Sales Problem(TSP) which is the problem of finding a closed tour that visit each city once with minimal cost [3]. In that ACO algorithm, ants start at a random city and perform a closed tour. While performing a tour, an ant select a city by a rule that an edge with lower cost, which means higher pheromone concentration, has higher possibility to be chosen. Once all ants complete their tour, the pheromone will be updated: a portion of pheromone evaporates on all edges, then each ant deposits an amount of pheromone on the edges along its tour. In this way, edges which belong to shorter tour are receiving higher pheromone concentration. Evaporating pheromone provides a mechanism for ACO to avoid converging to the solution that has local minimal cost.

4.3.2 SO-ACO

As a graph-based optimization algorithm, a graph must be generated to solve the DSE problem. One way to build the graph is to take the attributes K_{attr} specified as pragamas in the source code as vertices, and all possible values for each K_{attr} as as edges. Fig. 4.7 shows the graph for benchmark `ave16`.

Informally, ACO works as follows. First of all, build a graph based on attributes that a benchmark has. At every iteration, m ants start from *Start* point and go through each

Algorithm 7: The ACO Algorithm

```
1  Generating graph based on attributes
2  A ← {Ant_1, Ant_2, ..., Ant_m}
3  /* at this level each loop is called an iteration */
4  while Termination condition not occur do
5  |   /* at this level each loop is called a step */
6  |   for Ant_k ∈ A do
7  |   |   Select a tour with edge selection rule
8  |   |   Synthesis
9  |   |   Decide pheromone should be deposited on each edge
10 |   end
11 |   Global pheromone update
12 end
```

vertex $ATTR$ until End point. The edge an ant choose from $ATTR_i$ to $ATTR_j$ is based on `edge selection rule`. Ants prefer to select an edge with a high volume of pheromone, and the pheromone concentration reflects the minimum cost that an edge has. Once an ant has chosen its tour, HLS tools is excuted and the cost is calculated. The cost will deposited on the edges as pheromone according to `heuristic determine rule`. After all ants finished their tours, a `global pheromone updating rule` will be applied: a portion of the pheromone evaporates on all edges, and then each ant deposits some pheromone on the edges it has gone through. The process is iterated until termination condition occurs. Pseudo-code for ACO is shown in Algorithm 7.

Edge Selection Rule. There are two cases for an ant to select edges. Before each ant starts its journey, a random number $q \in [0, 1]$ and a predefined parameter $q_0 \in [0, 1]$ are used to decide which case the ant is in. If $q \leq q_0$, the ant is in the first case called `exploitation`, and the path with the lowest cost (C_{min}) from history will be mutated one edge randomly and assigned to that ant. In the second case called `exploration`, where $q > q_0$, the ant will select the edges based on the pheromone concentration. The ant goes through every vertex and chooses the edge between vertices one by one. Suppose ant k is at vertex $ATTR_i$, it will

choose the edge E_x to move to $ATTR_j$ according to the possibility

$$p_k(E_x) = \begin{cases} \frac{\tau(E_x)}{\sum_{E_u \in J_k(E_r)} \tau(E_u)}, & \text{if } E_x \in J_k(E_r) \\ 0, & \text{otherwise} \end{cases} \tag{4.11}$$

where τ is the pheromone concentration of E_x, $J_k(E_r)$ is the set of edges that exit between $ATTR_i$ and $ATTR_j$.

The edge selection rule provides a mechanism called `exploitation` to allow ant colony to explore a path which is slightly different from the one with C_{min} so far. `exploitation` mechanism is adopted due to the fact that, unlike TSP, in the HLS DSE problem the 'best' edge on some paths may not be the best choice in the path with lowest cost, which means some edges on the path with C_{min} may have relatively low pheromone. In our problem, it is not known how much impact of each edge has on the cost of the path unless the HLS tool is called, while on the contrary, in TSP, the length of each edge is fixed. In the worst case, some edges on the best path do not have pheromone on it, because no ant has explored the path with minimal cost and those edges are 'bad' for many other paths. With `exploitation` rule, ants are able to explore paths with smaller cost so that the result of ACO could improve. The parameter q_0 decides the relative importance of exploitation versus exploration. As mentioned before if $q \leq q_0$ then select the minimum cost edges with slightly change, otherwise, an edge is selected based on pheromone level according to 4.11.

Global Pheromone Updating Rule. In ACO, the global pheromone updating rule is implemented as 4.12 shows.

$$\tau(E_x) \leftarrow (1 - \alpha) \cdot \tau(E_x) + \sum_{k=1}^{m} \Delta T_k(E_x) \tag{4.12}$$

where

$$\Delta T_k(E_x) = \begin{cases} \frac{1}{C_{E_x}}, & \text{if } (E_x) \in \text{ tour done by ant } k \\ 0, & \text{otherwise} \end{cases} \tag{4.13}$$

Algorithm 8: ACO-Weighted Sum

```
1  Generating graph based on attributes
2  A ← {Ant_1, Ant_2, ..., Ant_m}
3  α ← 1.0
4  β ← 0.0
5  while α ≥ 0 do
6      while Termination condition not occur do
7          for Ant_k ∈ A do
8              Select a tour with edge selection rule
9              Synthesis
10             Decide pheromone should be deposited on each edge
11         end
12         Global pheromone update
13     end
14     α ← α −step
15     β = 1 − α
16 end
```

$\alpha \in [0, 1]$ is a pheromone decay parameter, C_{E_x} is the minimum cost of the edge that can be find so far performed by ant k, and m is the number of ants, which is another important parameter.

4.3.3 Weighted Sum ACO

Similar to previous weighted sum methods (4.1.3, 4.2.3), the weighted sum ACO sets the target as

$$\textbf{Minimize} \quad C \tag{4.14}$$

where

$$C = \alpha \cdot \frac{Area}{A_{max}} + \beta \cdot \frac{Latency}{L_{max}} \tag{4.15}$$

By tuning α, β, the cost function can be dynamically changed to reflect relative importance of *Area* and *Latency*. By iterating this step, a set of Pareto-optimal solution can be found. Pseudo-code for weighted sum ACO is shown in Algorithm 8.

Algorithm 9: ACO ξ-Constraint

```
1  L_constraint ← ∞
2  Generating graph based on attributes
3  A ← {Ant_1, Ant_2, ..., Ant_m}
4  while L_constraint ≥ L_min do
5      while termination condition does not occur do
6          for Ant_k ∈ A do
7              Select a tour with edge selection rule
8              Synthesis
9              Decide pheromone should be deposited on each edge
10         end
11         Global pheromone update
12     end
13     (A_min, L_i) ← global minimum solution
14     L_constraint ← L_i
15 end
```

4.3.4 ξ-Constraint ACO

Different from weighted sum method, ξ-constraint method set cost function as

$$\begin{aligned} &\textbf{Minimize } C = Area, \\ &\textbf{Subject to } Latency < \xi_i \end{aligned} \tag{4.16}$$

where ξ_i is the *Latency* find at $(i-1)$ round.

At each round, ant colony searching for the path which has A_{min} with constraint condition $L < \xi_i$. In the first round, ξ_1 is set to ∞ so that the algorithm would search the whole design space. Once the (A_{min}, L_1) is found, the algorithm move to next round. ξ_2 updates to L_1 and resume searching A_{min} with the new constraint condition. This step iterated until ξ_i equals to L_{min}. Algorithm 9 shows the pseudo-code for ξ-Constraint ACO.

4.3.5 Parameters in ACO

There are 4 parameters need to be set in ACO: m, α, q_0 and *Exit Condition*. m indicates the number of ants in ant colony system that start from *Start* point in each iteration. α is a

pheromone decay parameter to indicate how fast the pheromone evaporates over time. The value range for α is $\alpha \in [0, 1]$. q_0 indicates the importance of exploitation versus exploration in `Edge Selection Rule`. The range of q_0 is $[0, 1]$. The exit condition is set as follows: if the ant colony does not find smaller cost for consecutive N iterations, the algorithm will terminate.

CHAPTER 5

EXPERIMENTAL RESULTS

There are two subsections in this chapter. In the first subsection introduces the experimental setup used to evaluate our proposed exploration method. The second, shows the results obtained, together with the explanation and analysis of the data.

5.1 Experimental Setup

In this work, we use CyberWorkBench (CWB) as HLS tool, which supports C/C++ and systemC as input language, and select 7 different behavioral description that are written in C as benchmarks as Tab 5.1 shows. In this table, the first column is the benchmark name. The next three columns indicate benchmark's knob K_{attr} numbers for arrays, loops and functions respectively. The last column shows the total number of designs in the design space which is calculated by multiplying the number of all possible values of each attribute. The last four columns show that the larger a benchmark, the more knobs K_{attr} it has and more designs can be explored.

The complete experimental procedure is shown in Fig. 5.1. This procedure is executed for every algorithm and every benchmark. In this work, we compared our proposed ξ-constraint method with the traditional weighted-sum method for both SA (ξ-constraint SA vs. weighted sum SA), GA (ξ-constraint GA vs. weighted sum GA) and ACO (ξ-constraint ACO vs. weighted sum ACO). For the weighted-sum method, the step for changing α, β was set to 0.5.

To compare the result, we selected *Runtime* and *ADRS* as quality indicators. We also generated various combinations of parameters for SA, GA and ACO and used them for both ξ-constraint and weighted sum method. Tab 5.2 summarized all parameters in SA, GA and ACO and their value range when generating various parameter combinations.

Table 5.1. Benchmark Overview

Benchmark	#Arrays	#Loops	#Func	#Design
adpcm	1	1	0	44
ave16	1	2	0	147
kasumi	2	1	3	1400
interpolation	0	5	0	1024
gfilter	5	0	5	7776
aes	6	5	0	20480
snow3G	2	1	10	230400

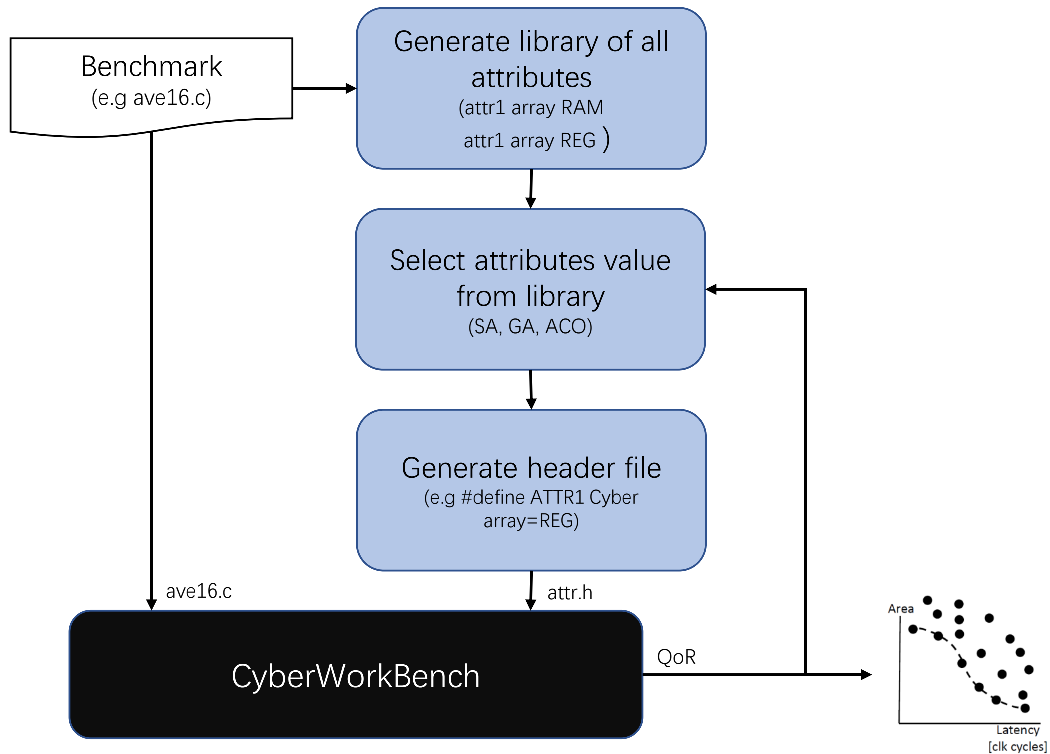

Figure 5.1. Experimental Setup

Table 5.2. Parameters Setting

Algorithm	Paramters	Range	Describe
SA	$N_{neighbor}$	1-2	Number of attributes replaced when generate neighbours.
	T_n	10-55	Temperate reduce interval, after n solutions reduce temperature.
	γ	0.1-0.95	Temperate reduce rate, $T_c = \gamma \cdot T_c$
	N_{exit}	5-50	Exit condition, after N solution not improve cost, exit SO-SA.
GA	P_{cros}	0.5-0.95	Crossover possibility.
	R_{muta}	0.11-2.0	Mutation Rate.
	N_{pare}	2-4	After N times breeding not improve fitness value, select new parents.
	N_{exit}	5-50	Exit condition, after N solution not improve fitness value, exit SO-GA.
ACO	m	10-19	Number of ants in the colony.
	α	0.11-2.0	Pheromone reducing rate, $Phero = (1-\alpha) \cdot Phero$
	q	0.4-0.85	Exploration possibility.
	N_{exit}	5-50	Exit condition, after N solution not improve cost, exit SO-ACO.

5.2 Results and Analysis

Among results of various parameters settings, we selected the best settings that lead to the smaller *ADRS* and shorter *Runtime* for each method and for each benchmark, and summarized those parameter settings in Tab 5.3. The first column is the name of benchmark. The next two columns are the SA parameter settings that are used to compare the performance of ξ-constraint SA versus weighted sum SA, e.g. 1-45-50-0.9 means in SA $N_{neighbor} = 1$, $N_{exit} = 45$, $T_n = 50$, $\gamma = 0.9$. SA-C represents ξ-constraint SA, and SA-W represents weighed sum SA. Similarly, column 4 and 5 are the GA parameter settings that are selected to do performance comparison. Finally, the last two columns are parameter settings for ACO. This table reveals the parameter settings which lead to good synthesis result are different for each benchmark. Therefore, setting a fixed parameter value as most of HLS DSE frameworks did is not the optimal strategy. Dynamically setting parameter for every

Table 5.3. Selected Parameter Value

Benchmark	$N_{neighbour}$-N_{exit}-T_n-γ		P_{cros}-R_{muta}-N_{pare}-N_{exit}		N_{exit}-m-α-q	
	SA-C	**SA-W**	**GA-C**	**GA-W**	**ACO-C**	**ACO-W**
adpcm	1-45-50-0.9	1-15-20-0.3	0.65-0.17-4-35	0.95-0.11-2-5	50-19-2.0-0.85	30-15-0.16-0.65
ave16	1-20-35-0.5	2-50-55-0.95	0.95-0.11-2-5	0.8-0.14-3-20	35-16-0.17-0.7	15-12-0.13-0.5
kasumi	1-15-40-0.75	1-20-25-0.4	0.9-0.12-2-10	0.55-0.19-4-45	30-15-0.16-0.65	50-19-2.0-0.85
interpolation	1-25-30-0.8	1-15-20-0.3	0.5-2.0-4-50	0.9-0.12-2-10	10-11-0.12-0.45	35-16-0.17-0.7
gfilter	2-50-50-0.95	2-35-40-0.7	0.9-0.12-2-10	0.5-2.0-4-50	25-14-0.15-0.6	20-13-0.14-0.55
aes	1-30-35-0.6	1-15-20-0.3	0.95-0.11-2-5	0.85-0.13-2-15	5-10-0.11-0.4	20-13-0.14-0.55
snow3G	1-30-55-0.95	1-25-30-0.5	0.55-0.19-4-45	0.75-0.15-3-25	15-12-0.13-0.5	25-14-0.15-0.6

meta-heuristic algorithm is a better way. The authors in [17] adopted machine learning to achieve that goal.

The following paragraphs will discuss the results of different parameter settings listed in Table 5.3. The proposed method ξ-constraint will be compared with weighted sum method for SA, GA and ACO respectively. Finally, results for all three algorithm will be put together to compare if there exists the best algorithm.

5.2.1 Weighted Sum SA vs. ξ-Constraint SA

Table 5.4 shows the *ADRS* and *Runtime* of the weighted-sum SA and ξ-constraint SA. In this table, the first column is benchmark's name. The next two columns are the values of *ADRS* and *Runtime* for ξ-constraint SA, and the last two columns are the values of *ADRS* and *Runtime* for weighted sum SA. *ADRS* value for each benchmark is calculated by the formula (3.5) and the *Runtime* is given in minutes. We also calculated the arithmetic average of *ADRS* and geometric mean of *Runtime* as row 8 and 9 shows. The last row shows the

Table 5.4. ξ-Constraint SA vs. Weighted Sum SA

Benchmark	**ξ-Constraint SA**		**Weighted Sum SA**	
	ADRS(%)	**Runtime(min)**	**ADRS(%)**	**Runtime(min)**
adpcm	0.0000	0.83	0.0000	0.76
ave16	0.0000	3.43	3.1646	3.16
kasumi	6.3526	19.93	13.9364	15.17
interpolation	0.0000	6.87	0.0000	6.41
gfilter	14.9797	57.26	18.6415	50.06
aes	0.2491	125.96	8.6528	86.86
snow3G	0.0000	56.22	0.0000	45.38
Avg.	3.0831		6.3422	
Geomean		14.83		12.44
Δ	51.39%	-19.25%		

difference of *ADRS* and *Runtime* for the ξ-Constraint method compared with the weighted sum method.

From Table 5.4, we can observed that in 4 out of 7 benchmarks all Pareto-optimal solutions are found with ξ-Constraint method whereas all Pareto-optimal solutions are found in only 3 benchmarks for weighted sum method. In addition, the *ADRS* obtained by ξ-Constraint SA is smaller than weighted sum SA for all benchmarks, but *Runtime* of ξ-Constraint SA is higher. On average, ξ-constraint SA decreases *ADRS* from 6.3422 to 3.0831 which is 51.39% *ADRS* reduced, but increases *Runtime* from 12.44 min to 14.83 min which is 19.25% *Runtime* increased. It can be concluded that ξ-Constraint SA can provide more accurate Pareto frontier and smaller *ADRS* compared with weighted sum SA, but there will be increase of *Runtime* as penalty.

5.2.2 Weighted Sum GA vs. ξ-Constraint GA

Table 5.5 shows the *ADRS* and *Runtime* of weighted sum GA and ξ-constraint GA. In this table, the first column is benchmark's name. The next two columns are the values of *ADRS* and *Runtime* for ξ-constraint GA, and the last two columns are the values of

Table 5.5. ξ-Constraint GA vs. Weighted Sum GA

Benchmark	**ξ-Constraint GA**		**Weighted Sum GA**	
	ADRS(%)	**Runtime(min)**	**ADRS(%)**	**Runtime(min)**
adpcm	0.0000	1.01	0.0000	0.71
ave16	1.2874	2.87	2.4803	2.58
kasumi	0.0000	18.32	16.9305	12.35
interpolation	0.0000	4.02	0.0000	4.10
gfilter	0.0000	60.40	3.0037	40.87
aes	1.6880	70.02	8.4791	67.15
snow3G	0.0000	40.03	0.0000	33.84
Avg.	0.4215		4.4134	
Geomean		12.01		9.79
Δ	90.37%	-22.74%		

ADRS and *Runtime* for weighted sum GA. *ADRS* value for each benchmark is calculated by the formula (3.5) and the *Runtime* is given in minutes. We also calculated the arithmetic average of *ADRS* and geometric mean of *Runtime* as row 8 and 9 shows. The last row shows the difference of *ADRS* and *Runtime* for the ξ-Constraint method compared with the weighted sum method.

From Table 5.5 we can observe that in 5 out of 7 benchmarks all Pareto-optimal solutions are found with ξ-Constraint method whereas all Pareto-optimal solutions are found in only 3 benchmarks for weighted sum method. In addition, the *ADRS* obtained by ξ-Constraint SA is smaller than weighted sum SA for all benchmarks, but *Runtime* of ξ-Constraint SA is higher for most benchmarks. On average, ξ-constraint GA decreases *ADRS* from 4.4134 to 0.4215 which is 90.37% *ADRS* reduced, but increased *Runtime* from 9.79 min to 12.01 min which is 22.74% *Runtime* increased. It can be concluded that ξ-Constraint GA can provide more accurate Pareto frontier and smaller *ADRS* compared with weighted sum GA, but there will be increase of *Runtime* as penalty.

Table 5.6. ACO Result

Benchmark	ξ-Constraint ACO		Weighted Sum ACO	
	ADRS(%)	Runtime(min)	ADRS(%)	Runtime(min)
adpcm	0.0000	1.03	0.0000	0.80
ave16	0.5866	3.93	2.4603	2.85
kasumi	1.1472	21.97	16.9305	18.88
interpolation	0.0000	11.86	0.0000	6.79
gfilter	0.0000	70.45	3.6619	57.57
aes	2.8469	142.54	9.1234	117.20
snow3G	0.0000	57.18	0.0000	50.62
Avg.	0.6544		4.5966	
Geomean		17.97		13.89
Δ	85.76%	-29.38%		

5.2.3 Weighted Sum ACO vs. ξ-Constraint ACO

Tab 5.6 shows the *ADRS* and *Runtime* of weighted sum ACO and ξ-constraint ACO. In this table, the first column is benchmark's name. The next two columns are the values of *ADRS* and *Runtime* for ξ-constraint ACO, and the last two columns are the values of *ADRS* and *Runtime* for weighted sum ACO. *ADRS* value for each benchmark is calculated by the formula (3.5) and the *Runtime* is given in minutes. We also calculated the arithmetic average of *ADRS* and the geometric mean *Runtime* as row 8 and 9 shows. The last row shows the difference of *ADRS* and *Runtime* for the ξ-Constraint method compared with the weighted sum method.

From Tab 5.6, we can observed that in 4 out of 7 benchmarks all Pareto-optimal solutions are found with ξ-Constraint method whereas all Pareto-optimal solutions are found in only 3 benchmarks with weighted sum method.In addition, the *ADRS* obtained by ξ-Constraint SA is smaller than weighted sum SA for all benchmarks, but *Runtime* of ξ-Constraint SA is higher. On average, ξ-constraint ACO decreased *ADRS* from 4.5966 to 0.6544 which is 85.76% *ADRS* reduced, but increased *Runtime* from 13.89 min to 17.97 min which is 29.38% *Runtime* increased. It can be concluded that ξ-Constraint ACO can provide more accurate

Table 5.7. Results Comparison among SA, GA and ACO

Method	Algorithms	ADRS (%) Avg.	Runtime (min) Geomean
ξ-Constraint	SA	3.0831	14.83
	GA	0.4251	12.01
	ACO	0.6544	17.97
Weighted Sum	SA	6.3422	12.44
	GA	4.4134	9.79
	ACO	4.5966	13.89

Pareto frontier and smaller *ADRS* compared with Weighted Sum ACO, but there will be increase of *Runtime* as penalty.

5.2.4 SA vs. GA vs. ACO

In the last step, we put all the geometric means listed in Tab 5.4, Tab 5.5 and Tab 5.6 into a single table (Table 5.7). In this table, the first two columns are the method and algorithm used respectively. The last two columns are the QoR indicators *ADRS* and *Runtime*. We group ξ-Constraint SA, GA and ACO as Group 1 (row 2 to 4), and Weighted Sum SA, GA and ACO as Group 2 (last 3 rows). By comparing *ADRS* and *Runtime* in Group 1 and 2 respectively, we discovered that among all three algorithms, on average GA yields smallest *ADRS* and shortest *Runtime*, therefore, we concluded that GA is the best meta-heuristic algorithms among the algorithms implemented in this work.

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 Conclusion

This work has studied the use of multiple meta-heuristic algorithms to address the HLS DSE problem. These algorithms include Simulated Annealing (SA), Genetic Algorithm (GA) and Ant Colony Optimization (ACO) which are algorithms typically used to solve Single Objective (SO)optimization problems. However, the HLS DSE problem is a Multi Objective (MO) Optimization problem. Therefore, these heuristics where modified to convert the SO formulation into an MO formulation.

The traditional conversion method is a weighted sum which assigns all objects a different weight and sum them up to constitute a single cost function. This method is relatively simple and easy to implement. As we applied it to the HLS DSE problem, the quality of result is often not good. Therefore this work proposed a new method called ξ-constraint to improve the quality of the Pareto-frontier found.

In the experiments, we selected 7 different benchmarks written in C and ran multiple different parameter settings for all algorithms with both methods. To compare the result among different methods and algorithms, we selected the results with smallest ADRS and shortest runtime for all benchmarks. As for the result, on average, ξ-constraint SA reduced ADRS by 51.39% at the cost of taking 19.25% longer to run. In the GA case, the ξ-constraint method could on average reduced the ADRS by 90.37% with the penalty of 22.74% longer runtime. In the ACO case, the ξ-constraint method improved the ADRS by 85.76% while increasing the runtime by 29.38%. Based on these results, we can conclude that ξ-constraint is able to improve the quality of Pareto frontier found by the meta-heuristic algorithms with small penalties of the runtime. In addition, if we compare performance of these algorithms,

it can be observed that GA in general leads to smaller ADRS and shorter runtime, and therefore is the best algorithm.

6.2 Future Work

To further improve QoR generated by meta-heuristic algorithms, as mentioned in Section **??**, a method to dynamically set the meta-heuristics parameters based on characteristics of a specific benchmark is one direction of future work. Typically, these meta-heuristics require a set of parameters to obtain good results. However, fixed parameter setting might not be the best for all of the benchmarks because benchmarks. Machine Learning could be used to help choosing the best set of parameters, to improved the QoR. [17] adopted SK-Learn Linear Tree Regressor which is a machine learning model to achieve this goal, more machine learning model can be explored in the future.

In addition, except for algorithms implemented in this work, there are a variety of other meta-heuristic algorithms that can be applied in DSE such as other Evolutionary Algorithms, Particle Swarm Optimization, etc..It would be interesting to study these too in the context of HLS DSE.

REFERENCES

[1] Cadence. Stratus high-level synthesis, available at: https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html.

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

[3] Dorigo, M. and L. M. Gambardella (1997, April). Ant colony system: a cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation 1*(1), 53–66.

[4] Hadjis, S., A. Canis, J. H. Anderson, J. Choi, K. Nam, S. Brown, and T. Czajkowski (2012). Impact of fpga architecture on resource sharing in high-level synthesis. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '12, New York, NY, USA, pp. 111–114. Association for Computing Machinery.

[5] Intel. High-level synthesis compiler, available at: https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html.

[6] Martin, G. and G. Smith (2009, July). High-level synthesis: Past, present, and future. *IEEE Design Test of Computers 26*(4), 18–25.

[7] MentorGraphics. Catapult high-level synthesis, available at: https://www.mentor.com/hls-lp/catapult-high-level-synthesis/.

[8] Micheli, G. D. (1994). *Synthesis and optimization of digital circuits*. McGraw-Hill Higher Education.

[9] Nam, D. and C. H. Park (2000). Multiobjective simulated annealing: A comparative study to evolutionary algorithms. *International Journal of Fuzzy Systems 2*(2), 87–97.

[10] NEC. Cyberworkbench, available at: https://www.nec.com/en/global/prod/cwb/index.html.

[11] Ngatchou, P., A. Zarei, and A. El-Sharkawi (2005, Nov). Pareto multi objective optimization. In *Proceedings of the 13th International Conference on, Intelligent Systems Application to Power Systems*, pp. 84–91.

[12] Raje and Bergamaschi (1997). Generalized resource sharing. In *1997 Proceedings of IEEE International Conference on Computer Aided Design (ICCAD)*, pp. 326–332.

[13] Schafer, B. C., Takashi Takenaka, and Kazutoshi Wakabayashi (2009). Adaptive simulated annealer for high level synthesis design space exploration. In *2009 International Symposium on VLSI Design, Automation and Test*, pp. 106–109.

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

[15] Smith, K. I., R. M. Everson, J. E. Fieldsend, C. Murphy, and R. Misra (2008). Dominance-based multiobjective simulated annealing. *IEEE Transactions on Evolutionary Computation 12*(3), 323–342.

[16] Wakabayashi, K. (2004). C-based behavioral synthesis and verification analysis on industrial design examples. In *Proceedings of the 2004 Asia and South Pacific Design Automation Conference*, ASP-DAC '04, pp. 344–348. IEEE Press.

[17] Wang, Z. and B. C. Schafer (2020). Machine learning to set meta-heuristic specific parameters for high-level synthesis design space exploration. In *Design Automation Conference (DAC)*, pp. 1–6.

[18] Xilinx. Vivado high-level synthesis, available at: `https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html`.

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

BIOGRAPHICAL SKETCH

Yiheng Gao was born in Guangchang, Jiangxi, China on 7th January 1994. She completed her Bachelors in Automation in the College of Mechatronics and Control Engineering from Shenzhen University, Shenzhen, China. She joined The University of Texas at Dallas to pursue her Masters in Computer Engineering in August 2017. She further joined the Design Automation and Reconfigurable Computing Lab (DARC lab) under the guidance of Dr. Benjamin Carrion Schafer in January 2019 to pursue research on design space exploration with high level synthesis. Yiheng's research interests include high level synthesis, algorithms, machine learning, computer architecture, and reconfigurable systems.

CURRICULUM VITAE

Yiheng Gao

May, 2020

Contact Information:

Department of Computer Engineering
The University of Texas at Dallas
800 W. Campbell Rd.
Richardson, TX 75080-3021, U.S.A.

Educational History:

B.E., Automation, Shenzhen University, Shenzhen, China, 2015
M.S., Electrical Engineering, The University of Texas at Dallas, 2020

Meta-Heuristic Algorithms in High-Level Synthesis Design Space Exploration
M.S. Thesis
Computer Engineering Department, The University of Texas at Dallas
Advisors: Dr. Benjamin Carrion Schafer

Employment History:

Associate Engineer, Caterpillar, Houston, TX, August 2019 – Dec 2019
Software Developer, Panasonic, Shenzhen, China, July 2015 – July 2016