Circuit Clustering for Cluster-based FPGAs Using Novel Multiobjective Genetic Algorithms

Yuan Wang

Ph.D

University of York
Electronics

September 2015

Abstract

Circuit clustering is one of the most crucial steps in a post-synthesis FPGA CAD flow. It attempts to efficiently fit synthesised logic functions into FPGA logic clusters. On a FPGA, different clusterings result in different circuit mappings, which affect FPGA utilisation, routability and timing, and therefore impact the circuit performance. This research proposes the use of a Multi Objective Genetic Algorithm (MOGA) as a methodology to solve the cluster-based FPGA circuit clustering problem.

Four alternative approaches based on MOGA methods are proposed in this research: RVPack is inspired by the stochastic feature that exists in Evolutionary Algorithms (EAs). GGAPack, GGAPack2, DBPack and HYPack, T-HYPack (Timing-driven HYPack) are then proposed and developed, which are fully customised MOGA-based circuit clustering methods. GGAPack clusters a circuit using a top-down perspective, and DBPack uses a new bottom-up perspective. HYPack combines GGAPack and HYPack – a hybrid method. According to experimental results, a few conclusions are drawn: It is possible to improve the performance of the greedy algorithm based circuit clustering methods by incorporating randomness. The performance of MOGA based top-down clustering is poor; however, using MOGA to cluster a circuit from a bottom-up perspective can produce better solutions. T-HYPack clustered circuit has the best timing performance compared with state-of-the-art methods. The experimental results also reflect a wider potential for using GAs to solve FPGA circuit mapping problems.

Contents

List of Tables

List of Figures

Acknowledgement

I would like to first thank the EPSRC (Engineering and Physical Sciences Research Council, UK) for support in funding this research. I would like also thank to my supervisors and group members, Prof. Andy Tyrrell, Dr. James Alfred Walker, Dr. Martin Albrecht Trefzer and Dr. Simon Bale for their guidance and support throughout my Ph.D journey.

A special thanks should also go to my parents for their love and support; without them I would not have been able to come to the UK to receive a higher-standard education. I also need to thank my girlfriend, Miss Zhen Qiu, for three years of her love and patience.

There are a number of other people I would like to thank: Mr. Yunfeng Ma – for spending a lot of time discussing algorithm-related questions. Mr. Jianxin Zhao – my undergraduate supervisor, who always encourages me when I run into trouble. Mrs. Emily Gaspar-Philpott, who has helped with thesis editing and formatting.

Finally, I would like to give my special thanks to Mr. Yunfeng Ma and Yunfeng's wife, Miss. Xiaoyuan Chen - thanks to them for taking great care of our rented property. Mr. Jingbo Gao - thanks for the times we have worked together in the university library, finishing our thesis writings.

Declaration

I, Yuan Wang, declare that this thesis is a presentation of original work and I am the sole author. This work has not previously been presented for an award at this, or any other, University. All sources are acknowledged as References. My research is part of PAnDA (Programmable Analog and Digital Array) project, and it is funded by the EPSRC. Some (partial) methods and results of this thesis, in collaboration with other authors, are published in the following papers:

Methods and partial results of DBPack (Chapter 7) have been published in the following paper:

- Y. Wang, S. J. Bale, J. A. Walker, M. A. Trefzer, and A. M. Tyrrell, "Multiobjective Genetic Algorithm for Routability-Driven Circuit Clustering on FPGAs", in Proceeding, SSCI2014. IEEE, 2014, pp. 109-116

Methods and partial results of HYPack (Chapter 8) have been published in the following paper:

- Y. Wang, J. A. Walker, S. J. Bale, M. A. Trefzer, and A. M. Tyrrell, "Two-Phase Multiobjective Genetic Algorithm for Constrained Circuit Clustering on FPGAs", in Proceeding, CEC2015. IEEE, 2015

Chapter 1

Introduction

1.1 Background

Modern electronics can be dated back to the innovation of solid transistors (Edgar, 1930), and the Integrated Circuit (IC) (Jacobi, 1952). In earlier times, electronic systems were built on discrete transistors. Although these transistors have a reduced size and lower power consumption compared with valves (Fleming, 1905), the relatively large size, high power consumption, and the discrete characteristic distribution of transistors are still issues when implementing complex electronic systems. The IC first appeared in 1949 (patent was published on 15 May 1952) where an integrated-circuit-like amplifying device was successfully developed and implemented (Jacobi, 1952). Since IC, the device size, power consumption are gradually decreased, and the component coherence issues are progressively solved. IC allows a number of components; this including transistors, resistors or even capacitors and inductors, implemented on a single silicon chip, which produce a circuit that is powerful, stable, efficient and portable.

Application-Specific Integrated Circuits (ASICs) have become more popular since the IC was born – for instance, the 7400 series digital ICs (Lancaster,

1974). Though an IC can replace a number of discrete transistors or a block of circuits, the IC integration level, in early times, is still low. The industry cannot be satisfied with these still-small-scale ICs, and it continues to be developed. Very-Large-Scale Integrated (VLSI) circuits were invented in the 1970s (Mead and Conway, 1978); today, a very-large-scale IC can contain billions of components. Rather than implementing a simple application, for example an amplifier or a logic gate, these ICs enable the possibility to implement higher performance applications, such as a Central Processing Unit (CPU), Digital Signal Processor (DSP), mixed-signal system, and ultimately producing an entire electronic system on a single silicon chip, which is known as SoC (System on a Chip).

With the rapidly increasing needs of complex electronic designs, the flexibility of application-oriented ICs is extremely limited, and the weaknesses of these ICs are exposed. These include the time consuming design cycles, high fabrication and testing costs. In addition, there are also requirements in testing and research, which need fully customised devices. Obviously, application-oriented ICs are not able to meet these requirements. In this case, reconfigurable devices were proposed. These devices are stand-alone ICs, but integrate a number of reconfigurable resources which allow a designer to further configure these ICs on the circuit level so different functions can be realised. Since these reconfigurable ICs can be easily configured, it means that when producing a design using this type of ICs, it largely decreases the design-to-market time, and reduces the engineering costs at the microelectronic level.

FPGA, short for Field Programmable Gate Array, is an important digital reconfigurable device, usually based on Look-Up Tables (LUTs – normally a LUT is composed of a set of Static Random Access Memory cells– SRAMs) and programmable Flip-Flops (FFs), after PLD (Programmable Logic Device, it contains a "fixed-OR, programmable-AND" plane plus memory, the plane can be used to implement "sum-of-products" binary logic equations. If a device does not have the memory, it refers to PLA™–Programmable Logic

Array.) and CPLD (Complex Programmable Logic Device, it might contain many PLAs or PLDs that are linked by programmable interconnections on a single CPLD), and it was initially launched on the market by Xilinx in 1985. FPGAs were then quickly adapted into the electronic industry. Currently, FPGA is a common device to be utilised in for example high performance computing, signal processing, and communication, load-adaptive and fault-tolerant systems. To get a FPGA properly configured, a Computer Aided Design (CAD) flow is used. This flow contains a few key steps, which are circuit synthesis, circuit clustering, clustered circuit placement and routing. By using this flow, it allows a higher level abstraction design to be automatically processed and mapped onto targeted FPGAs.

Circuit synthesis – a HDL (Hardware Design Language) described circuit is automatically synthesised into logic gates (functions).

Circuit clustering – as modern FPGAs use logic clusters in their architectures, where a logic cluster is a larger logic macro containing a set of reconfigurable elements that can realise logic functions, synthesised logic gates (functions) have to be grouped, also referred to separate the synthesised circuit, into logic clusters to match a FPGA architecture, at meantime producing less clusters (groups) and cluster interconnects – circuit connections between logic clusters, where such clustered circuit uses less FPGA resources.

Placement – clustered logic clusters are placed onto a FPGA.

Routing – FPGA pre-defined routing resources are used to form connections between the logic clusters in order to form a complete circuit.

Circuit clustering is the first step in post-synthesis processes in FPGA CAD flow. Hence, the quality of a clustered circuit can significantly affect the circuit mapping on a FPGA. If a circuit is poorly clustered, the following CAD processes could not efficiently adjust the circuit and would result in the mapped circuit having low routability, low speed and high power consumption

on the FPGA. On the other hand, circuit clustering usually involves different clustering metrics and also has a number of constraints. As a result, circuit clustering is always an important and difficult process, and it is also the reason that circuit clustering is a hot topic in the area of academic research.

In the early stages, circuit clustering is facilitated by a number of greedy algorithms (Betz and Rose, 1997a; Marquardt et al., 1999; E.Bozorgzadeh et al., 2001; Singh and Marek-Sadowska, 2002; Cormen et al., 2009). They are simple and use a bottom-up clustering perspective, where they cluster a circuit from a local-optimal perspective. When attempts to find a global optimal solution of a problem, dealing with the problem from a local optimal perspective is normally difficult to get the global optimality. Therefore, using the bottom-up methods, the clustered circuit is often less optimal (Feng, 2012). At the same time, clustering metrics are also considered in these algorithms, which use a weighted approach. Although this weighted method can deal with multiple clustering objectives, a better trade off solution is usually difficult to find (Rajavel and Akoglu, 2011). Recently, it has been helpful to use the graph-partitioning-based methods in FPGA circuit clustering methods (Marrakchi et al., 2005; Feng, 2012; Feng et al., 2014a). The graph-partitioning-based methods can cluster a circuit from a global perspective – this is known as top-down clustering methods. These methods are considered to produce better solutions than the greedy algorithms since it uses the global clustering perspective. However, the graph-partitioning-based method mainly focuses on minimising circuit interconnects of partitioned circuits, where it might be difficult to control, for example, which connection is inside a logic cluster, or how many logic functions are in a logic cluster. This means that these types of methods cannot efficiently incorporate clustering metrics and constraints (Marrakchi et al., 2005; Feng, 2012).

1.2 Motivation

Charles Darwin indicated that the root of a large number of different species on the planet was based on the principle of mutation and natural selection, also called natural evolution, and this is the famous theory of Darwin's natural selection (Darwin, 1859). Darwin considers individuals that are best adapted to an environment can survive, and have chances to produce offspring. In contrast, the less adapted individuals gradually die off, and these individuals are replaced by better individuals. Darwin called this mechanism "survival of the fittest".

Genetic Algorithms (GAs) are a subset, or one dialect to be more precise, of Evolutionary Algorithms (EAs), and EAs are the algorithms that imitate the process of natural evolution, and use the natural evolutionary process as a model to solve actual problems (Holland, 1975). GAs are popular in a wide range of areas such as music generation, strategy planning, VLSI technology and machine learning. When a GA is utilised to solve a problem, it is normally not necessary to have specific knowledge about the target problem, which indicates that GAs, or EAs in general, are a model-free heuristic algorithm, and it is an automatic problem solver (Langeheine, 2005). The evolved solutions of a GA are usually useful as proved in the "no free lunch" theorem – "any two optimisation algorithms are equivalent when their performance is averaged across all possible problems" (Wolpert and Macready, 1997). GAs can be extended for supporting multiobjective problems, and this is different from the weighting approach which weights all objectives in a single function to score a solution. In Multi Objective GAs (MOGAs), the multiobjective mechanism is often based on Pareto optimality (Pareto, 1906). This means that MOGA can produce trade off solutions for multiobjective problems (Fonseca and Fleming, 1993).

Research-based FPGA circuit clustering methods have been developed for almost two decades since VPack (Betz and Rose, 1997a). It is notable that there are superior methods, for example T-VPack (Marquardt et al.,

1999) and iRAC (Singh and Marek-Sadowska, 2002). However, these are all bottom-up and greedy-algorithm-based methods, where these algorithms are limited by their "models" and the bottom-up clustering perspective (Singh and Marek-Sadowska, 2002). This implies that these "models" and the clustering perspective can be further enhanced, so the solutions produced by these methods can be improved. PPack (Feng, 2012; Feng et al., 2014a) is a new clustering method based on graph-partitioning methods and uses a top-down clustering perspective. PPack tests show that it can produce excellent circuit clustering solutions compared with previous greedy-algorithm-based methods, but unfortunately PPack cannot efficiently deal with complex clustering objectives and constraints.

As previously introduced, GAs are a model-free and automatic problem solver, and can also be applied to multiobjective problems, where the multiobjective features can efficiently incorporate targeted objectives of a problem and also problem constraints. Therefore, MOGA can be used to solve the FPGA circuit clustering problem. Using different MOGA designs, the MOGA can either solve the clustering problem from a global perspective, referred as the top-down clustering method, or a local-optimal perspective, the bottom-up clustering method.

1.3 Research hypothesis

1.3.1 Statement of hypothesis

The research hypothesis is as follows:

> The quality and performance of a multiobjective circuit mapped to a cluster based FPGA can be improved through the use of evolutionary algorithms during the circuit clustering stage of a FPGA computer aided design flow.

1.3.2 Analysis of hypothesis

Nowadays, FPGAs tend to group a number of basic logic blocks, a basic logic block can be as simple as one LUT plus one configurable FF, as logic clusters (Altera Corp., 2001, 2003b,a; Xilinx Inc., 1998, 2010, 2013, 2012b, 2014). This design can reduce the use of reconfigurable resources, and speed up a mapped circuit compared with designs that do not use logic clusters. From the FPGA chip implementation perspective, logic clusters are defined as a large identical logic macro, thus, a FPGA design can be implemented by simply repeating the placement of the macro. From the CAD perspective, when preferentially arranging circuits in logic clusters, it can reduce the difficulties in routing a circuit on a FPGA, as some connections can be formed within logic clusters.

Circuit clustering is a key step in a FPGA CAD flow, where a large synthesised circuit is separated into sub circuits. It has to guarantee that each sub circuit can be mapped onto a FPGA logic cluster, where each sub circuit meets the hardware constraints of the logic cluster. To increase the quality of a clustered circuit, a circuit clustering method is required to cluster more circuit connections in logic clusters, so it can therefore use fewer logic cluster interconnects. In this case, the final routing stage has fewer connections to route. At the same time, the clustering method has to maximise the usage within a logic cluster so fewer logic clusters can be used, which allows more logic to be mapped onto a FPGA. Clustered CLB number and CLB interconnect number can be considered as the routability of a clustered circuit.

Apart from the routability, a circuit clustering method is also required to optimise the performance of clustered circuits. Circuit speed, or timing, is a key factor that affects the performance of a clustered circuit. A circuit speed can be determined by the circuit's critical path delay. The more stages found in the critical path of a circuit, the lower the circuit speed will be, and the performance of the circuit will be decreased. An effective clustering method usually clusters more critical connections inside FPGA logic clusters, as the FPGA logic cluster has shorter wires and delays, while in the meantime

leaving equally critical or non-critical connections as FPGA logic cluster interconnects. When routing such circuits, circuit speed can be improved.

Genetic algorithms are one subset of evolutionary algorithms – the powerful model-free problem solvers, and can adapt different representations, genetic operations and selection mechanisms, which means genetic algorithms are effective for solving or optimising complex engineering problems. On the other hand, the selection mechanism can be extended to support multiobjective problems. This implies multiobjective genetic algorithms can be a suitable method for exploring the approach of solving the FPGA circuit clustering problem.

1.4 Novel contributions

This doctoral research focuses upon using MOGAs to solve the FPGA circuit clustering problem in a FPGA CAD flow. To achieve this target, a stochastic mechanism is first incorporated into a standard greedy-algorithm-based circuit clustering method. Subsequently, a set of fully customised MOGAs are developed, which represent complete program frameworks for using MOGAs to solve the FPGA circuit clustering problem, and also highlight which clustering perspective (top-down/bottom-up) is efficient for solving this problem. It is also shown which objectives are more effective at optimising the quality of a clustered circuit. In addition, this research also propose an on-line optimisation method to optimise the performance of clustered circuits. This thesis presents four major methods, which are listed as follows:

1) RVPack, short for Random VPack, FPGA circuit clustering method is proposed. This method is based on VPack (Betz and Rose, 1997a). Similar to a GA, which uses stochastic variations to drive evolutions, randomnesses are injected to the greedy-algorithm-based VPack algorithm. Although, in this case, RVPack might produce less optimised solutions, some superior solutions can be identified. This method and

its experiments indicate that it is possible to improve solution qualities by incorporating stochastic variations in classic-greedy-algorithm-based circuit-clustering methods.

2) GGAPack/GGAPack2, where GGAPack is short for Grouping Genetic Algorithm Pack, are fully customised and MOGA based FPGA circuit clustering methods. These methods cluster a circuit from a global perspective. Unfortunately, experimental results show that GGAPack is inefficient at producing highly optimised solutions. However, GGA-Pack provides a useful GA framework to solve the circuit clustering problem. GGAPack2 is based on GGAPack – the only difference is that GGAPack2 produces solutions based on RVPack solutions in stead of randomly initialise a population. Real mapping tests show that GGAPack2 solutions are not able to optimise circuit performances, but GGAPack2 can produce better solutions in terms of basic circuit clustering requirements. This means that it might be inefficient to use GAs to solve the FPGA circuit clustering problem from a global perspective.

3) DBPack, short for DataBase Pack, redesigns the MOGA-based circuit clustering method, GGAPack, and uses a new bottom-up clustering perspective, which directly searches a group of basic logic blocks to form a FPGA logic cluster. This method produces excellent solutions in the aspect of including circuit connections in logic clusters, and its solutions are better than iRAC (Singh and Marek-Sadowska, 2002), where iRAC was considered the state-of-the-art connection-absorption clustering method. This method indicates that clustering a circuit from this new bottom-up perspective, and using MOGA, allows the FPGA circuit clustering problem to be properly solved.

4) HYPack/T-HYPack are short for Hybrid Pack and Timing-driven Hybrid Pack. These approaches combine the methods of GGAPack and DBPack. According to HYPack testing results, HYPack produced solutions are further optimised compared with DBPack. T-HYPack

solves the FPGA circuit clustering problem by producing clustered solutions that are similar to the HYPack, but in addition, T-HYPack also speeds up the clustered circuits on FPGAs. This is facilitated by using an on-line optimisation method. Although T-HYPack does not take the circuit critical paths into account, where circuit critical path is often used in conventional methods, T-HYPack can actually improve the timing performance of the clustered circuits. These methods, HYPack/T-HYPack, suggest that bottom-up and top-down clustering methods can be combined, and an on-line optimisation can be used to further optimise the performance of clustered circuits.

1.5 Thesis structure

Chapter 1 introduces the background and motivation of this research, and highlights the research hypothesis and its novel contributions. This chapter also presents the structure of this thesis.

Chapter 2 first introduces reconfigurable devices, and clarifies their basic concepts. Then this chapter focuses on FPGAs (Field Programmable Gate Arrays), which are an important reconfigurable digital device. This includes FPGA programmable logic and routing architectures. This chapter emphasises that nowadays FPGAs are cluster-based, and routing architectures are usually island-styled. In order to provide a background for circuit clustering method research, it is defined as a cluster-based island style FPGA model.

Chapter 3 explains why Computer Aided Design (CAD) is important in FPGA design flow. A research based CAD flow is introduced. The definition, requirements and significances of circuit clustering are explained. The rest of this chapter reviews a number of state-of-the-art circuit clustering methods, and comments on their advantages and disadvantages.

Chapter 4 reviews the concept of Evolutionary Computing (EC), which

is based on Darwin's theory of natural selection. EC actually refers to a set of Evolutionary Algorithms (EAs), and the components of EA are introduced. The rest of this chapter focuses on Genetic Algorithms (GAs), and MultiObjective Genetic Algorithms (MOGAs). The MOGA is the major method that has been used to solve the FPGA circuit clustering problem in this research.

Chapter 5 introduces the Random VPack, RVPack, FPGA circuit clustering method. This chapter first reviews VPack algorithm in detail, and highlights how the randomnesses are injected in VPack to produce the RVPack. The experimental setups and result comparisons are presented in the rest of this chapter.

Chapter 6 presents Grouping Genetic Algorithm based GGAPack and GGAPack2 FPGA circuit clustering methods. These methods are top-down clustering methods. This chapter clarifies GA representations, genetic operations, fitness function designs and multiobjective selection schemes. For GGAPack2, it explains how the RVPack solutions are used in GGAPack2. The detailed experimental setups, results and result analysis are summarised.

Chapter 7 proposes a new MOGA-based FPGA circuit clustering method, the DBPack. This method fixes problems that are identified in Chapter 5 – RVPack. DBPack clusters a circuit using a new bottom-up perspective. Similar to GGAPack, it introduces GA representation, genetic operations, fitness function designs and the multiobjective selection scheme. The experimental setups, results and comparisons follow in this chapter.

Chapter 8 combines GGAPack and DBPack methods, and proposes HYPack, and T-HYPack – the hybrid FPGA circuit clustering methods. HYPack and T-HYPack are based on DBPack produced solutions, and use GGAPack method as a second optimiser. In T-HYPack, it also optimises the timing performance of a clustered circuit by incorporating a FPGA placement and routing. This work is carried out by an on-line optimisation approach, where a clustered circuit can be continuously optimised for the timing performance

on a targeted FPGA. The experimental setups, results and result analysis are included.

Chapter 9 summarises the findings of the proposed methods, and concludes this research. This chapter also highlights the future work.

Chapter 2

Reconfigurable Devices

2.1 Introduction to reconfigurable devices

With the continued rapidly increasing needs of complex electronic system design, the weaknesses of pre-defined Integrated Chips (ICs), or Application-Specific ICs (ASICs) are exposed, where these weaknesses include, for example, longer design-to-market time, higher testing cost and fixed function. It has to emphasise that the design, fabrication and testing of a new ASIC are the most expensive and crucial parts in the microelectronics industry. To meet many testing and research requirements, which required a large number of full-customisable devices, reconfigurable devices were appeared. Reconfigurable devices are normal ICs but these ICs supply with a number of configurable resources, which allow these devices to be configured, referred to as function updatable, as any type of circuit or for many applications, and therefore avoid reinvestments in design, fabrication and testing in the microelectronics.

The configurability of a digital system first appeared from the Programmable Read-Only Memory (PROM), and developed through many other logic devices such as the Programmable Logic Array (PLA^{TM}), Programmable Logic Device (PLD), and Field Programmable Gate Arrays (FPGA). (Brown

and Rose, 1996). Similar to basic electronic circuits, reconfigurable devices, also known as reconfigurable hardware, can be divided into analogue and digital types. A typical reconfigurable digital device is the Field Programmable Gate Arrays (FPGAs), its counterpart in the analogue domain being the Field Programmable Analogue Arrays (FPAAs), and Field Programmable Transistor Arrays (FPTAs). The typical FPAAs are the Zetex (Zetex Corp., 1999), Lattice ispPAC series (Lattice Corp., 2000, 2001a,b,c) and Anadigm AN221E04 (Anadigm Inc., 2003) FPAAs. These devices allow the analogue building blocks of a circuit to be configured, for example current sources and operational amplifiers (OPAMPs). Some reconfigurable analogue devices also provide lower level configurations – transistor levels, the FPTAs, for instance JPL FPTAs (Stoica et al., 2000) and Heidelberg FPTAs (Langeheine et al., 2001), are the typical devices. This thesis focuses on reconfigurable digital devices, in particular FPGAs.

Due to specific needs, the arrangements of interconnects and reconfigurable fabric structure, where the fabric is defined as a set of reconfigurable building blocks, and the arrangement refers to an architecture, in reconfigurable devices can be different. However, their architectures can still be classified as linear, array, mesh, crossbar, data-path, etc. The details of these architectures are well introduced in Trefzer and Tyrrell's book (Trefzer and Tyrrell, 2015), a book for reconfigurable hardware. On the configurable device, the configurable fabric structure is normally one of two types – homogeneous or heterogeneous structures. In a homogeneous structure, the configurable fabric is formed from identical configurable blocks, and these blocks are arranged in a regular fashion. In contrast, the heterogeneous structure means that, apart from some identical configurable blocks, the configurable fabric also contains a number of specialised blocks, known as hard macros. In addition to the reconfigurable fabric structures, another important parameter for the reconfigurable device is the granularity, which indicates the configurable level of the reconfigurable device. The granularity is usually defined at three levels. These are: fine-grained, medium-grained and coarse-grained. Table 2.1 shows that the configurable levels in digital and analogue configurable devices with

Table 2.1: The configurable levels in digital and analogue configurable devices with different granularities (Trefzer and Tyrrell, 2015)

Granularity	Digital Recfg. Device	Analogue Recfg. Device
Fine	Logic Gates, Loop-Up Tables, MUXs...	Transistors, Current Mirrors, Differential Pairs...
Medium	Flip-Flops, Memories, Multipliers...	OPAMPs, Comparators...
Coarse	ALUs, Processors...	Filters, ADCs, DACs...

Recfg. = Reconfigurable
MUXs = multiplexers
ALU = Arithmetic Logic Unit
ADC, DAC = Analog to Digital Converter, Digital to Analog Converter

different granularities.

2.2 Field Programmable Gate Array (FPGA)

2.2.1 Definition

The Field Programmable Gate Array, abbreviated to FPGA, is a pre-fabricated IC. It is a digital device which belongs to the category of reconfigurable digital devices, and has a number of pre-defined reconfigurable resources which allow the FPGA to be programmed or reprogrammed as any type of digital circuit or system after it has been fabricated (Brown and Rose, 1996).

It has been commonly considered that the modern FPGA era began with the first commercial FPGA introduced in 1985 – the Xilinx XC2064 FPGA, a static RAM, the SRAM (Pavlov and Sachdev, 2008), based FPGA that has 64 Configurable Logic Blocks (CLBs), 58 inputs and outputs (IOs), and

internal configurable fabric which is implemented as 4-input Look-Up Tables (LUTs), (Carter et al., 1986; Xilinx Inc., 1985; Kuon et al., 2007), which can be classified as fine-grained. Today, FPGAs have grown and developed significantly. A modern FPGA can contain more than 330,000 logic blocks, and have thousands of IOs (Kuon et al., 2007; Altera Corp., 2011b; Xilinx Inc., 2015a). As a result, FPGAs are widely utilised in digital systems, and used as a central configurable hub between different subsystems.

2.2.2 Applications

In the last two decades, by benefiting from the modern Very-Large-Scale Integration (VLSI) circuits technology (Weste and Harris, 2010), the FPGA scale has been increased significantly. FPGAs can be applied to a variety of applications. The applications of the FPGA can be summarised as follows:

1) From the hardware perspective, the FPGA can implement any logic circuits (Brown and Rose, 1996). In comparison with complex gate-level-ASIC-based digital PCBs (Printed Circuit Boards), the use of FPGAs can simplify the PCB, and lead to the FPGA being an all-in-one solution for digital logic. Moreover, modern FPGAs have a large number of configurable resources, including heterogeneous blocks; these enable the FPGA to build different digital subsystems, or even an entire system.

2) From the semiconductor industry perspective, due to the flexibility of the FPGA, FPGAs are widely used to verify new designs, or fast prototyping designs. For example, the FPGA can be utilised to investigate timing and logic verification of a new digital design.

3) From the product perspective, the FPGA can implement high-speed, high-precision applications. These applications include: ultra high-speed interfaces, sophisticated controllers, high-speed signal processors,

advance filters and complex communication systems. (Brown and Rose, 1996).

4) From the research and academic perspective, primitive concepts of reconfigurable computing were first convinced in Estrin and Viswanathan's paper (Estrin, 1960; Estrin and Viswanathan, 1962). Since the availability of the FPGA, the implementations of these concepts can finally be accomplished. FPGAs are also utilised in the areas of dynamic reconfigurable, working-load-adaptive, and fault-tolerant systems. On the other hand, the FPGA also enables research in FPGA architectures, and Computer-Aided Design (CAD) algorithms.

2.2.3 Advantages and disadvantages

When FPGAs merge with ASICs level fabrication, according to Gokhale and Graham's book (Gokhale and Graham, 2005), a number of advantages are notable:

1) Since the FPGA is programmable and offered as a standard product, for a system development, the design-to-market time can be significantly reduced.

2) Pre-fabricated FPGA products allow designers to focus on the higher level development, so by using the FPGA, there is no engineering cost at the microelectronic-level implementations, testings and fabrications.

3) FPGA manufacturers provide a number of different FPGAs, referred to commercial FPGA products, and these FPGAs are pre-tested by the FPGA manufacturer. Therefore, FPGA, or FPGA product, can be a distinct platform to verify new designs without considering the fault of microelectronic level.

4) When the FPGA is deployed for post-fabricated electronic systems, the reconfigurability of the FPGA is beneficial as it means that new

functions or changes can be added or made to the circuit level. This can be achieved remotely or by an on-line means.

In contrast, the negative effects of using FPGAs are also presented, and these are:

1) Modern FPGAs utilise SRAM (Pavlov and Sachdev, 2008) as the basic configurable elements, as well as deploying many pre-defined routing resources. This means that FPGAs use a greater area, have longer delays and consume more power compared with ASICs.

2) Although the revised Moore's law has predicted that the IC integrated transistor number is doubled every two years (Moore, 1965, 2006), area overhead is still the major issue in the FPGA and increases the fabrication cost; the cost is much higher than the ASIC when an ASIC is used in mass fabrication.

3) From the circuit performance perspective, when implementing the same application on both an ASIC and FPGA, Kuon and Rose point out that (Kuon and Rose, 2006), as reconfigurable resources exist in the FPGA, the FPGA can use up to 20 to 30 times more chip area. At the same time, the FPGA can be 3 to 4 times slower and consume 10 times more power than the ASIC. Although the fabrication technology is continuously developed, the circuit performance on FPGAs might still be lower than ASICs. This is due to massive reconfigurable resources in FPGAs.

4) Normally a design or application is mapped to the FPGA using the CAD flow, the results of of these CAD algorithms can significantly affect the performance of the design on FPGAs; this is because different CAD algorithms can produce different mappings and use different reconfigurable resources of a FPGA. As a result, design or application performances cannot be guaranteed as the mapping is not unique.

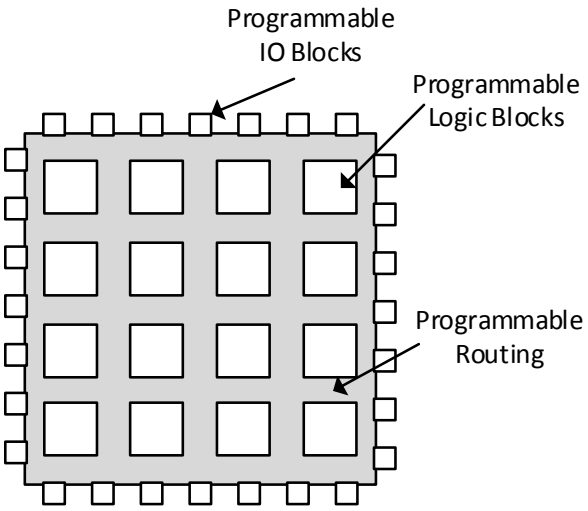


Figure 2.1: A generic FPGA architecture (Gokhale and Graham, 2005)

2.3 Overview of FPGA architectures

2.3.1 Basic architecture

In general, a FPGA contains at least three types of components, which are programmable logic block – also known as the basic logic element (BLE), programmable routing resources and programmable input-output (IO) blocks. These programmable logic blocks are the smallest logic elements, and can be connected to each other by using the routing resources to form a circuit. IO blocks are the interface to link to other circuits outside the FPGA. Figure 2.1 shows a generic FPGA architecture.

If these programmable logic blocks are all identical, this indicates that the FPGA is a homogeneous structure. However, to meet different application requirements, the FPGA can have a number of specialised blocks - this is referred to as a heterogeneous FPGA structure, an example of which is the SRAM block. The generic heterogeneous FPGA structure is shown in Figure A.1 in Appendices. How many detailed reconfigurations can be achieved

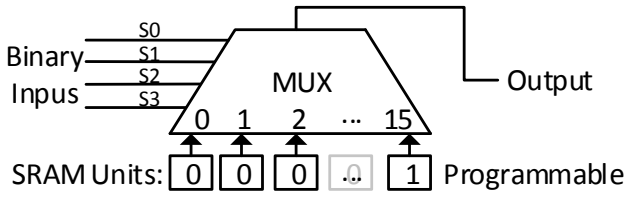


Figure 2.2: A generic 4-input Look-Up Table (LUT)

usually specifies the FPGA granularity. If the configurable logics are operated at the bit level, or small word which is less than 4 bits, the FPGA will be fine-grained (Trefzer and Tyrrell, 2015). As a fine-grained architecture has more routing resources, it uses a larger chip area, and further increases the delay or power of applications. The FPGA granularity is usually balanced, which is to retain the flexibility, and also keep costs low on the chip size.

2.3.2 Programmable logic architecture

Look-Up Table – LUT

Figure 2.2 shows a 4-input, where 4 indicates the LUT size, SRAM-based LUT, which has been widely utilised in commercial FPGAs for realising combinational logic. In fact, the LUT is a truth table of a logic. A LUT is generally comprised of a main multiplexer, and a number of SRAM cells, where a size k LUT operates as a k address lines memory block and has 2^k SRAM cells. In Figure 2.2, the left input pins are equivalent to the inputs of a logic gate. Below the multiplexer, it shows the SRAM cells with pre-loaded data. If the inputs of the multiplexer are logic "1, 1, 1, 1" to the pin $S0, S1, S2$ and $S3$, the multiplexer output will be logic "1", otherwise it will be logic "0" as all other SRAM cells have "0" stored in them. This configuration indicates that the LUT is representing the function of a 4-input "AND" gate. Although most FPGAs use LUTs for combinational logic, there are a few architectures, for example Actel ProASIC flash family (Actel Corp., 2009)

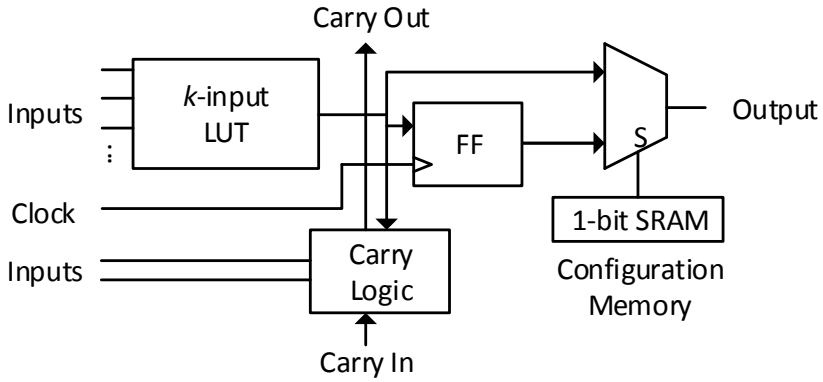


Figure 2.3: A generic FPGA programmable logic block (Gokhale and Graham, 2005)

and QuickLogic pASIC 1, 2, 3 series FPGAs (QuickLogic Corp., 1998, 2000, 2005), that realises the logic by standard logic gates and multiplexers. A reason of using the standard logic gate, instead of the LUT, is intended to achieve a more fine-grained FPGA. For instance, a master-slave FF can be produced by the standard gate in the Actel ProASIC Plus FPGA.

Programmable logic block (BLE)

In most FPGAs, the programmable logic block, or known as BLE used in Xilinx FPGAs, contains one or more LUTs and a configurable Flip-Flop (FF). In many devices, there are even more resources, for example, fast carry logic. The FPGA programmable logic block can be generalised in Figure 2.3. In this block, the LUT is for the implementation of combinational logic, and the configurable FF can be configured as a flip-flop or latch used for the implementation of sequential circuits. The outputs of the LUT and FF are selectable via a multiplexer controlled by a 1-bit SRAM cell – an example of the programmability of a FPGA. It is important to emphasise that the SRAM memory exists in the entire block, which enables it to configure the block functions in each part of the FPGA. Fast carry logic is usually in the block as well – it aims to reduce delays and configurable resource usages when

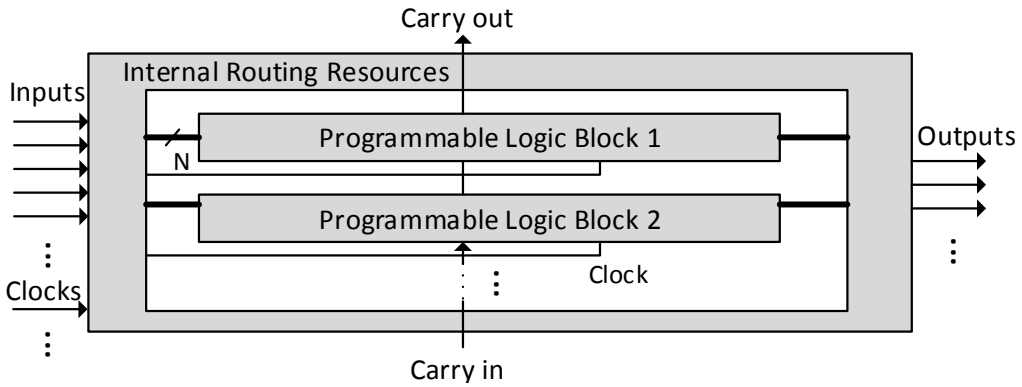


Figure 2.4: Generic programmable logic cluster, it contains a few logic programmable blocks, and internal routing resources. The internal routing resources can be configured to form connections for internal logic blocks. Moving some connections in the cluster can reduce the routing pressure of FPGA higher routing architecture.

performing carry logic operations.

The commercial FPGA programmable logic block is more complicated than the generic one illustrated here. Gokhale and Graham have commented that commercial FPGA programmable logic blocks are usually designed for maximising flexibility (Gokhale and Graham, 2005). For example, the configurable FF can be configured as combinations of asynchronous or synchronous sets and resets, and its trigger modes can be negative, positive edges, or the level trigger. In recent Xilinx and Altera FPGAs, their programmable logic blocks can support many additional functions. For instance, typical examples are that the carry logic in Xilinx Virtex FPGA can implement multiplication functions, and the carry logic in Altera Stratix II FPGA (Altera Corp., 2011a) is replaced with full adders.

Programmable logic cluster

Arranging a number of simple programmable logic blocks on a FPGA might use a enormous number of routing resources. A common solution in most

Table 2.2: Mainstream FPGA LUT sizes, larger logic block names, and larger logic block sizes over years

FPGA	Year	LUT Size	Name	Logic Block Size
Xilinx XC3000	1987	4	CLB	2
Xilinx XC4000	1990	3/4	CLB	1 (3-LUT)/2 (4-LUT)
Altera FLEX 8000	1992	4	LAB	8
Altera FLEX 10K	1995	4	LAB	8
Xilinx Virtex	1998	4	CLB	4
Altera Apex 20K	1998	4	LAB	10
Xilinx Virtex-II	2001	4	CLB	8
Altera Apex II	2001	4	LAB	10
Altera Stratix	2002	4	LAB	10
Altera Cyclone	2002	4	LAB	10
Xilinx Virtex-4	2004	4	CLB	8
Altera Stratic II	2004	3/4	CLB	24 (3-LUT)/16 (4-LUT)
Xilinx Spartan-3E	2005	4	CLB	8 (2 Slices)
Altera Cyclone II	2005	3/4	LAB	16
Xilinx Virtex-5	2006	5/6	CLB	8 (2 Slices)
Altera Cyclone III	2007	3/4	LAB	16
Xilinx Virtex-6	2009	5/6	CLB	8 (2 Slices)
Altera Stratix V	2010	3/4/5	LAB/MLAB	10
Xilinx Virtex-7	2011	5/6	CLB	8 (2 Slices)
Xilinx Zynq-7100	2013	5/6	CLB	8 (2 Slices)

MLAB = Memory Logic Array Block, it is similar to LAB.
Slice – recent Xilinx FPGAs contain 2 slices in each CLB.
Each slice contains 4 programmable logic blocks.
x-LUT = x-input LUT

commercial-LUT-based FPGAs is to group a few programmable logic blocks together as a programmable logic cluster as shown in Figure 2.4, and arrange these larger logic clusters on the FPGA. In a programmable logic cluster, it contains a few logic programmable blocks, and internal routing resources. In general, the internal routing resources can be configured to form connections for internal logic blocks. This design allows many programmable logic blocks to be routed within the clusters, so lower routing pressures are given to the FPGA higher level structure. Moreover, the use of the cluster can also

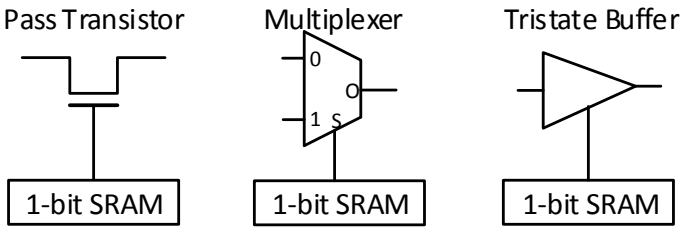


Figure 2.5: Routing-purpose switches used for FPGAs (Gokhale and Graham, 2005)

improve circuit performances as the cluster internal routing wire length is shorter. Current FPGAs tend to group more than 8 programmable logic blocks in a larger programmable logic cluster. In different brand FPGAs, the programmable logic cluster has different names. Xilinx refers to the programmable logic cluster as Configurable Logic Block, the CLB, and Altera calls it Logic Array Block, the LAB. Table 2.2 summarises the FPGA LUT and cluster size for mainstream FPGAs. Note that, in some FPGAs, the programmable logic cluster, or the programmable logic block, contains two different sized LUTs, and these LUTs can connect together for particular functions, increasing the flexibility.

2.3.3 Routing architecture

Routing infrastructure

FPGAs have a number of pre-defined wires and switches both in the programmable logic block, cluster and the higher level architecture for the purpose of routing. In terms of different FPGA architectures, the arrangement of these wires and switches is different. However, the routing-purpose switch can be generally classified into three types. These are pass transistor, multiplexer and tristate buffer, Figure 2.5 illustrates each of these. For each switch, there is a 1-bit programmable SRAM cell which controls the switch state. Among switch types, both the multiplexer and tristate buffer are active switches, and

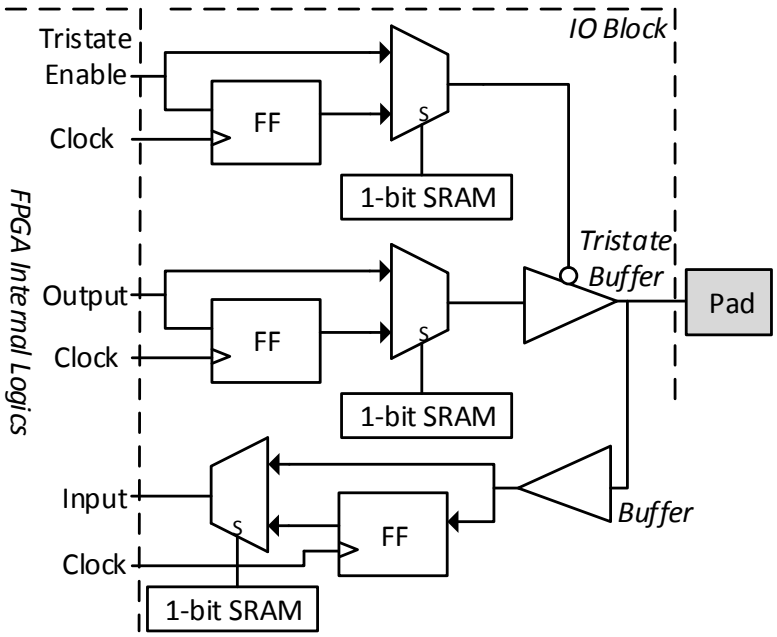


Figure 2.6: A generic FPGA programmable input and output block (Gokhale and Graham, 2005)

the pass transistor is a passive switch.

From the implementation area perspective, the multiplexer uses most chip area, and the area grows exponentially as the multiplexer input number increases. The pass transistor occupies the smallest area, as well as being lower power, but, due to the transistor gate capacitances and the transistor conducting resistance, the pass-transistor is also the "slowest" switch, which is only utilised on occasions when speed is not an issue. To compromise the area, FPGA architecture uses combinations of these switches as appropriate.

Programmable IO block

To connect other circuits, the FPGA deploys the programmable input-output (IO) blocks as the interface, and these blocks are around the FPGA chip die. In general, an IO block comprises multiplexers, tristate buffers and

flip-flops (FF), and the tristate buffer is the main component to control the IO signal flow. Figure 2.6 shows a generic FPGA programmable IO block. "Programmable" means that an IO pad is able to configure as a particular function based on circuit design requirements, for example, as an input or output pin, or a high impedance pin.

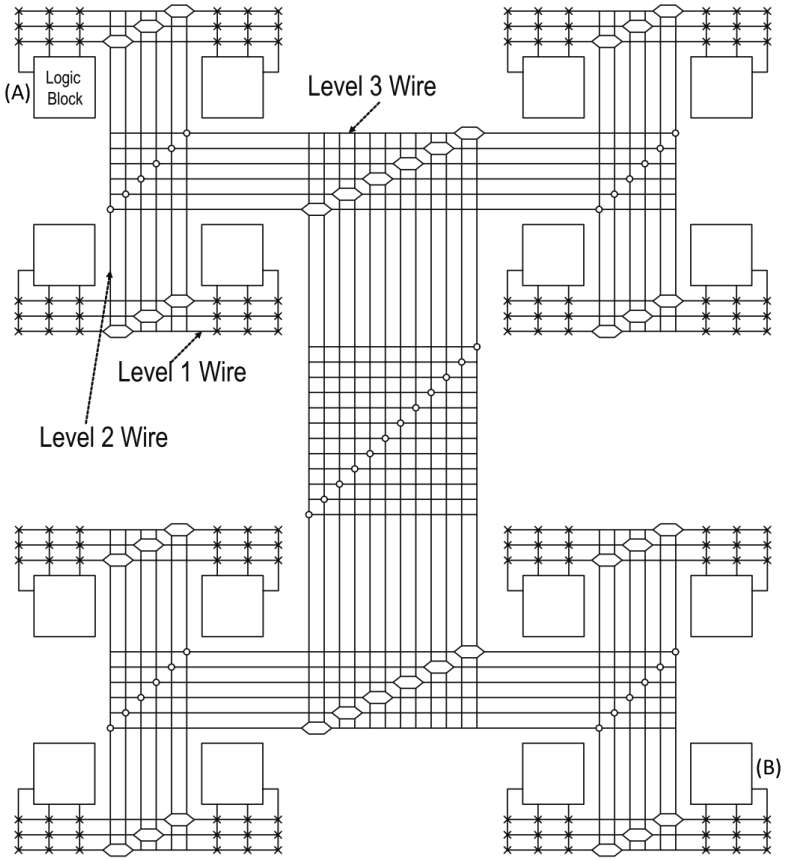


Figure 2.7: Hierarchical routing architecture FPGA example, if the upper left corner logic cluster (A) has a signal to the upper right corner logic cluster (B), a routing flow will be as follows: Level 1, Level 2, Level 3, and back to Level 2, Level 1 at the right of the figure. (Tsu et al., 1999)

Common FPGA routing architecture

The routing architecture refers to the position relationship between the FPGA routing channel, where a channel contains a number of routing-purpose wires, and the logic cluster or the logic block. It also indicates how the routing wire connects to the logic cluster, and how many wires and routing switches are used (Kuon et al., 2007). For most fine-grained commercial FPGAs, as discussed in Kuon's review, the routing architecture can be characterised as hierarchical (Aggarwal et al., 1994) or island-style (Betz et al., 1999; Brown et al., 1992a). Details of these two architectures are summarised as follows:

Hierarchical routing architecture

The hierarchical routing architecture is commonly used in the Altera FPGAs, and these FPGAs include the Flex10K (Altera Corp., 2003d), Apex (Altera Corp., 2003c) and Apex-II (Altera Corp., 2002) series. The main feature of the architecture is that it separates FPGA logic clusters into a number of distinct groups (Aggarwal et al., 1994; Tsu et al., 1999) as shown in Figure 2.7. In this design, the connections within the logic cluster can be made at the lowest routing hierarchy (Level 1) using shorter wires. In contrast, if a connection is between distant groups, the connection has to travel more routing hierarchies.

From the geometrical perspective, in this routing architecture, there is an interesting phenomenon; where even two logic blocks that are physically adjacent, their connections might be via different routing hierarchies, which when connecting to them, causes a large circuit delay. Aggarwal (Aggarwal et al., 1994) have indicated that this architecture can improve performances for certain circuits as the delay of the architecture is predictable. However, as the CMOS variability (Bowman et al., 2002) increases, the delay estimation is becoming more difficult. In theory, for these routing hierarchies, the same routing hierarchy should have the same propagation delay. However, the

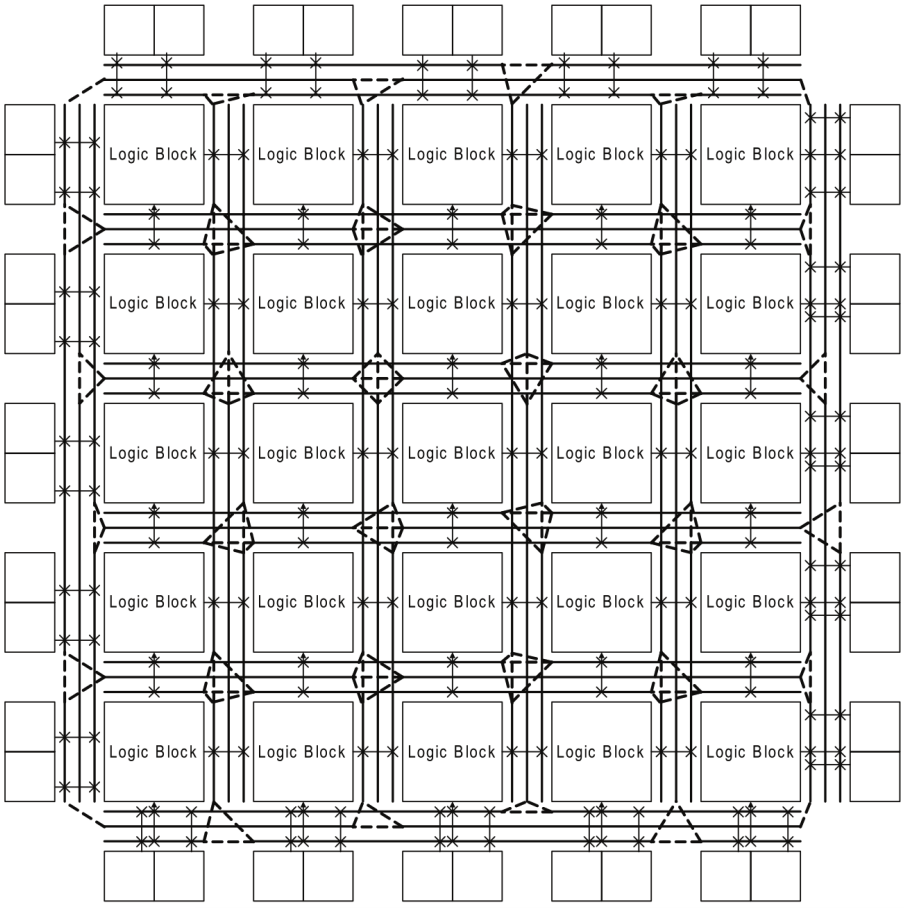


Figure 2.8: A generic island-style FPGA routing architecture (Kuon et al., 2007)

CMOS variability can significantly vary the delay. On the other hand, this architecture also applies pressures to the design mapping process. If a mapped circuit cannot match well the feature of the architecture, it will decrease the utilisation of the architecture. These are the reasons that modern FPGAs have skipped this type of routing architecture (Tsu et al., 1999).

Island-style routing architecture

The island-style routing architecture refers to a two-dimensional mesh architecture, in which the logic cluster is arranged as a mesh, and routing resources are integrated in the mesh (Kuon et al., 2007). In terms of a logic cluster, each side of the logic cluster has the routing channel accessible. Figure 2.8 shows a generic island-style FPGA routing architecture. Currently, most commercial SRAM-based FPGAs, such as Altera Stratix II (Altera Corp., 2011a), Stratix III (Altera Corp., 2011b) FPGAs, Lattice LatticeXP (Lattice Semi. Corp., 2007) FPGA and Xilinx Virtex-4 (Xilinx Inc., 2010), Virtex-5 (Xilinx Inc., 2012a), Virtex-6 (Xilinx Inc., 2012b), Virtex-7 (Xilinx Inc., 2014) FPGAs, etc., are based on this routing architecture.

In this architecture, the routing channels are pre-defined, and each channel has W routing wires or tracks – the channel width. Apart from the wires, a number of routing switch boxes are deployed next to logic blocks or logic clusters, and cut certain wires as a number of wire segments. This results in this unique architecture having a number of different length wire segments, which provide a relatively higher routing flexibility, where, to form a connection, a suitable-length wire can be utilised. Since there are wire segments surrounding the logic blocks, by staggering them, the start and end points of each wire can be suitably arranged. As well as this, these logic blocks and switch boxes are identical components. This means these wires and components are able to optimise to form a single tile, then this styled FPGA can be built by replicating the tile (Kuon et al., 2007). The island-style FPGA is the most common architecture which is used in circuit clustering methods research, and more details are introduced in Section 2.4.

2.3.4 Heterogeneous block

To provide more flexibility, and extend FPGA functions, FPGAs often integrate different types of heterogeneous blocks. Some of these block types are

summarised as follows:

Embedded memory block

Though the FPGA programmable logic block has a reconfigurable FF, the implementation of memory rich applications are extremely inefficient as the FF can only store 1-bit data, and it is difficult to configure into different geometrical sizes. Since the Xilinx XC4000 series FPGAs (Xilinx Inc., 1993), Xilinx has provided a new feature to its LUTs, which can be used as asynchronous RAMs. This feature has been developed, and currently, Xilinx FPGA LUT can be configured as synchronous RAMs, dual-ported RAMs and shift registers. Altera, was first to introduce on memory block in their FPGA – the Altera FLEX 10K series FPGA (Altera Corp., 2003d). Today, most FPGAs deploy the memory block in the FPGA architecture (Altera Corp., 2011b; Xilinx Inc., 2011; Altera Corp., 2012; Xilinx Inc., 2012a,b, 2014; Altera Corp., 2015).

Arithmetic logic block

It is costly to use programmable logic blocks to implement complex arithmetic operations; even the long routing wire can increase the circuit delay resulting in low performances. FPGAs often supply the arithmetic logic block, also known as the DSP (Digital Signal Processing) block. In most FPGAs, this block is implemented as an 18x18 multiplier (Xilinx Inc., 2008; Altera Corp., 2011a). The DSP block can be used in the following operations: addition, subtraction, multiplication and multiply-accumulate (MAC). The use of these blocks can significantly speed up applications for example the FFT (Fast Fourier Transform) and FIR (Finite-Impulse Response).

Embedded microprocessor block

Although most FPGA configurable logic fabrics can be used to implement soft cores for control-intensive applications and finally producing the FPGA as a SoC (System on a Chip), the FPGA manufacturers also place dedicated embedded microprocessor blocks as larger hardware macros in the FPGA architecture, and these microprocessors are mostly ARM-, or MIPS-based 32-bit RISC processors, for example, Xilinx Zynq-7000 series integrate high-performance ARM-cortex cores (Xilinx Inc., 2015b), and drives the FPGA development to the direction of "CPU + reconfigurable fabric". There are two major advantages of using the dedicated processor: Firstly, it saves the reconfigurable resources, and gives the design more flexibilities. Secondly, the dedicated processor tends to be power saving, and produce higher performances, as well as being more secure.

2.4 Cluster-based island-style FPGA and its model

Most circuit clustering methods research target the cluster-based island-style FPGA, and using this architecture as a FPGA model. The first reason is that the commercial FPGAs, for example the Altera FLEX6K, 8K, 10K series (Altera Corp., 2001, 2003b,a) and Xilinx XC5200, Virtex (Xilinx Inc., 1998, 2010, 2013, 2012b, 2014) series FPGAs , tend to group more programmable logic blocks in a larger logic cluster, which is known as "cluster-based", as well as these clusters are more generic, so research outcomes can be extended to different FPGA architectures. The other factor is that the implementation of this architecture can be easily carried out using a hierarchical construction concept if the FPGA layout tile is well arranged, as mentioned in the previous section. Therefore, the cluster-based island-style FPGA is popular for research, specially for the CAD algorithm research. This section introduces a simplified cluster-based island-style FPGA model, and clarifies the key parameters for

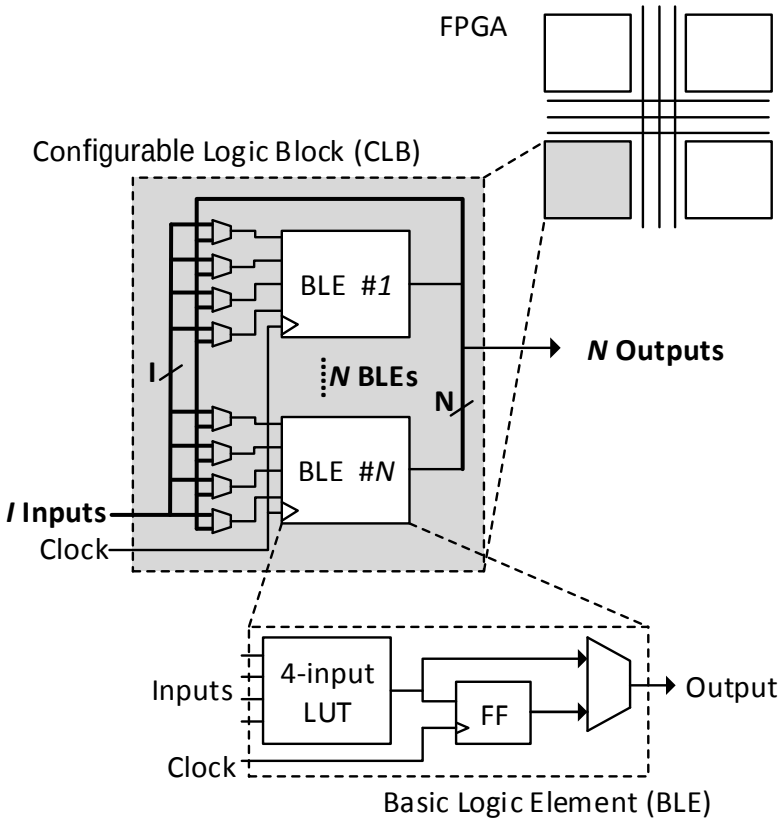


Figure 2.9: Cluster-based FPGA BLE and CLB internal structures (Betz et al., 1999)

this model.

2.4.1 CLB and BLE model

In this type of FPGAs, the larger programmable logic cluster, called CLB (Configurable Logic Block) or logic block, contains N programmable logic blocks, also named BLEs (Basic Logic Elements), and the BLE is the smallest reconfigurable logic element. The BLE includes a k-input LUT and a reconfigurable FF for realising both the combinational logic and sequential circuits. The parameters of the CLB (BLE) are summarised in Table 2.3. As discussed

Table 2.3: CLB (BLE) model parameters and meanings (Marquardt, 1999)

Parameter	Meaning
k	The number of inputs of the LUT (size)
N	The number of BLEs in the CLB
I	The number of inputs of the CLB, for LUTs
$Mclk$ (Clock)	The number of clock of the CLB, for FFs

previously, the LUT is based on SRAM and multiplexers - the more input the LUT has, the more area is taken, and even increasing the LUT size also requires more routing resources. Therefore, a trade off is usually established for balancing the LUT size. According to Betz, Rose and Marquardt's work (Betz, 1998; Marquardt, 1999), it is area-efficient when the LUT size in BLE is 4 ($k = 4$) and the CLB clock number is 1 ($Mclk = 1$), where the clock can meet most single clock driven circuits (benchmark circuits). Figure 2.9 shows the cluster-based island-style FPGA CLB and BLE models.

Inside the CLB, there are a number of BLE outputs that can connect to other BLE inputs. For any BLE, its output can connect to any other BLEs via the internal routing resources, also known as the crossbar (Kuon et al., 2007) or Input Interconnect Block (IIB) (Feng and Kaptanoglu, 2008), the left multiplexers of the CLB which are shown in Figure 2.9, this type of CLB will be named as a "full-connected" CLB, and it is the classic model used in circuit clustering method researches. The advantage of this CLB is that the number of inputs of the CLB can be less than the total number of BLE inputs – $k \times N$, which reduce the area of CLB internal routing resources. The additional benefit of using the full-connected CLB is that this CLB can simplify the CAD algorithm as these BLEs within the CLB are all identical to the routing resources (Marquardt, 1999).

Based on the same reasoning which improves the FPGA area efficiency, the number of BLEs which is the value of parameter N, within the CLB can be set from 1 to 10, but without the number 2 ($N = 2$). When $N = 2$, the FPGA area efficiency cannot be maintained (Betz, 1998; Marquardt, 1999).

The value of parameter I highlights the input bandwidth of the CLB. If I is equal to $k \times N$, this CLB will be known as input-bandwidth-free CLB, or stated that the CLB does not have input bandwidth. Otherwise, if the I is less than $k \times N$, the CLB will be known as input-bandwidth-constraint CLB (Feng, 2012). The input-bandwidth-constraint CLB is widely used, for example, in Altera Cyclone, Stratix (Leventis et al., 2003; Lewis, 2003) series FPGAs, and it is also the model used in VPR (Betz and Rose, 1997b), which is a research-based CAD tool. Note that, nowadays, although not all FPGAs have this constraint, the input-bandwidth-constraint CLB can actually represent a type of problem in the CAD algorithm research, especially in the research of circuit clustering methods. In order to assign a suitable value for I while optimising the logic utilisation in a CLB, Betz and Rose have suggested that I value is equal to $2N + 2$ (Betz and Rose, 1998). Subsequently, Ahmed and Rose's paper (Ahmed and Rose, 2000) has further generalised this relationship as shown in Equation 2.1.

$$I = (N + 1) \times \frac{k}{2} \tag{2.1}$$

2.4.2 Routing architecture model

Figure 2.10 shows the detailed island-style FPGA routing architecture (model), and it defines the interconnection method of the input or output of the logic block to the routing channel as well as the arrangement of the wire segment in the channel. In this architecture, here are X rows $\times$ Y columns array logic blocks ($X = Y$), and the logic block use the previous model, so this FPGA is a fine-grained and homogeneous architecture. For the logic block, its inputs and outputs are connected to the routing channel via the input and output connection blocks (Rose and Brown, 1991). To reflect the flexibility of the connection block, the fraction of wire segment width in the channel, which refers to the pre-defined channel width W, to the connection number of the input or output of the logic block is used. Based on the fraction, two

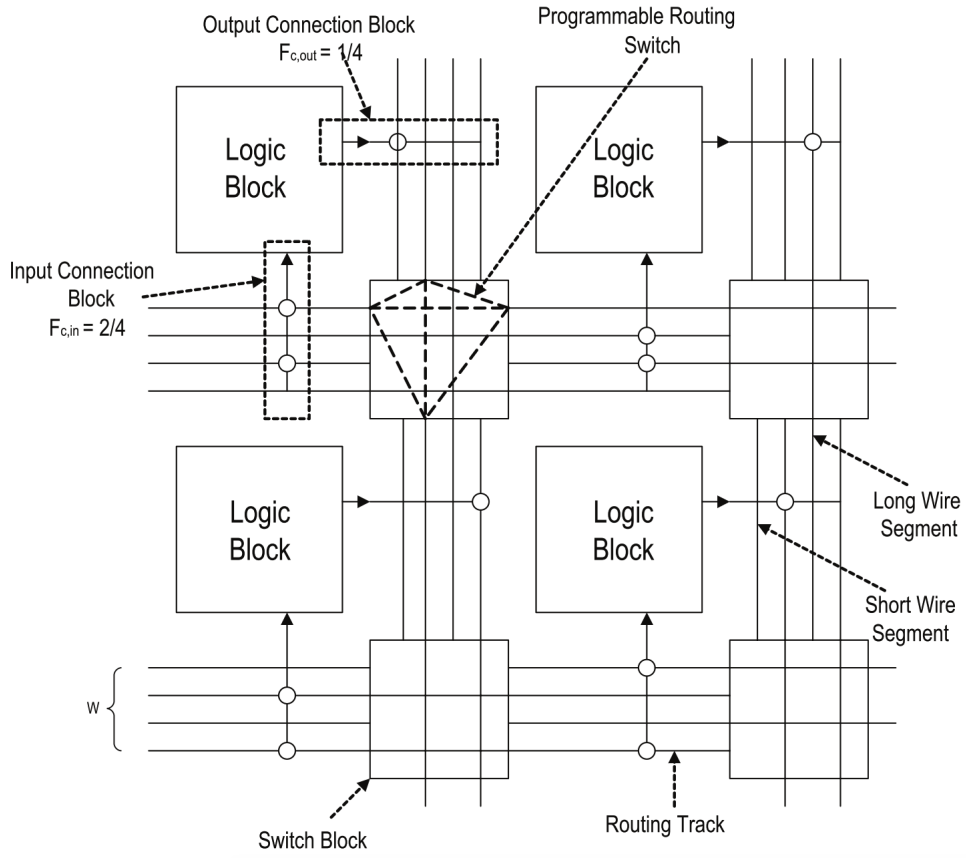


Figure 2.10: Detailed island-style FPGA routing architecture (Betz et al., 1999)

parameters, $F_{c,in}$ and $F_{c,out}$, are further defined to indicate the flexibility of the input and output connection blocks respectively (Kuon et al., 2007).

The switch box (Rose and Brown, 1991), which contains a set of programmable routing switches, is placed between logic blocks in the island-style architecture. The function of the switch box is to form connections for the horizontal and vertical routing channels, and, along with the logic blocks, produce a functional circuit. Switch box flexibility is defined as the parameter F_s, where it is the possible connection number that a wire segment can connect to other wire segments. For example, F_s is 3 in Figure 2.10. For the routing architecture model, the common switch box models are the disjoint (or also

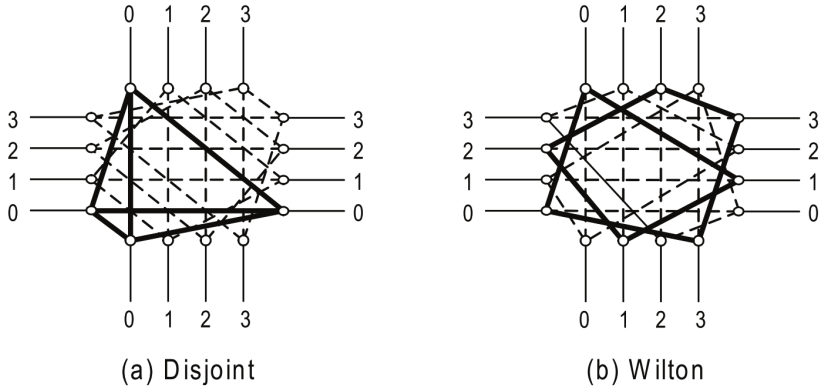


Figure 2.11: Disjoint and Wilton switch box (Kuon et al., 2007)

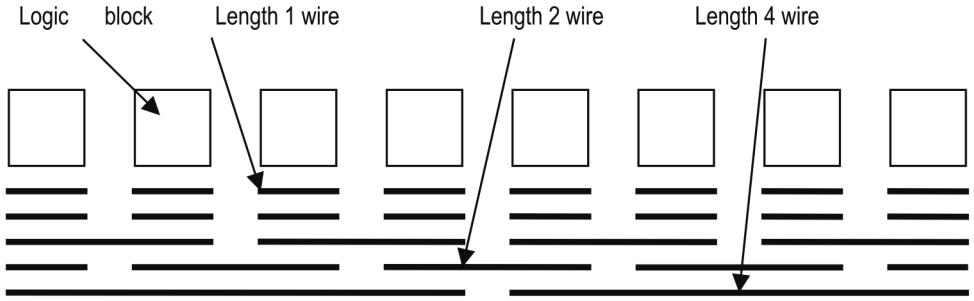


Figure 2.12: The definition of wire segment length, this figure also shows the staggered wire segments. (Marquardt, 1999; Kuon et al., 2007)

known as subset) (Wu and Marek-Sadowska, 1995) and Wilton (Wilton, 1997) switch boxes. Figure 2.11 shows these two models. The disjoint switch box was used in the Xilinx XC4000 series (Xilinx Inc., 1993), and other commercial FPGAs. However, this switch box has its limitations, where, as shown in Figure 2.11 (a), the connection can only be made for the same numerical designation, which limits the routing domains. The Wilton switch box uses the same number of programmable switches, but it is more flexible (Kuon et al., 2007).

In this architecture, the lengths of wire segments can be uniform or different, and the length is defined as a logical length which is suggested by

Table 2.4: Routing architecture model parameters and meanings

Parameter	Meaning
Homogeneous	FPGA architecture feature
$X \times Y$	FPGA scale (area), $X = Y$ square logic block arrays
W	Channel width – the number of tracks
$F_{c,in}$	Logic block input connection block flexibility
$F_{c,out}$	Logic block output connection block flexibility
F_s	Switch box flexibility
Wire segments	length – uniform or different.

how many logic blocks it spans as shown in Figure 2.12 (Betz et al., 1999; Marquardt, 1999). For instance, a wire has length one, it means that this wire segment spans one logic block. The wire segments in channels are arranged in a staggered formation (Kuon et al., 2007), where some wire segments are over the switch box uncut, but some are cut as shown in Figure 2.10 – the long wire segment or the short wire segment. This is another feature of this architecture. Table 2.4 has summarised the key parameters of the cluster-based island-style FPGA routing architecture model.

2.5 Summary

This chapter introduces the reconfigurable devices, and these devices can be classified as the reconfigurable digital device and reconfigurable analogue device. This chapter then focuses on the FPGA, the reconfigurable digital device. The FPGA definition, applications and the advantages and disadvantages have been described. This chapter also reviews the generic FPGA structure, which covers the programmable logic architectures and routing architectures – it highlights the BLE, CLB, routing infrastructure, IO block and hierarchical routing and island-style routing architectures. As the cluster-based island-style architecture is usually used to modern FPGAs, based on the generic FPGA structure, this chapter introduces the cluster-based island-style FPGA model, and clarifies what is the bandwidth-constraint CLB and how

the routing switch boxes and routing channels are arranged. Apart from these concepts and arrangements, this chapter also explains how to evaluate the routing flexibility and estimate the routing wire length. The cluster-based island-style FPGA architecture is important, so it is typically used in CAD algorithm research, especially in the circuit clustering algorithm research. The FPGA CAD flow and classic circuit clustering algorithms will be introduced next chapter.

Chapter 3

CAD for FPGAs and Circuit Clustering Methods

3.1 Why Computer-Aided Design (CAD)?

When the first commercial FPGA was introduced to the market by Xilinx, there were no CAD (Computer-Aided Design) tools available. Designers or customers had to implement their applications (or designs) on FPGAs manually, although Xilinx had provided simple graphic-based tools which allowed the designer to configure specific logic elements and switches in the FPGA model, and produced the configuration bitstream automatically by their design tools (Kuehlmann, 2012). This design implementation method was extremely complex, even for a small application. In addition, small changes to their designs would require a new design cycle to be made to produce a new configuration for the FPGA. Such a design approach also increases the difficulty of debugging.

Implementing an application on a modern FPGA could involve the configuration of hundreds of thousands, or even billions, of FPGA programmable elements and switches, which is not able to be carried out by "hand". To

fill the design-to-implementation gap, Xilinx first developed CAD tools to assist FPGA application implementations, with a particular focus on design automation, subsequently, other vendors have been involved in this development. Due to FPGA becoming increasingly popular, the design of CAD algorithms and tools has aroused the interest of ASIC tool vendors, and has also become a hot topic in research areas (Kuehlmann, 2012). Nowadays, there are a number of vendors who supply FPGA CAD tools, and a number of these tools take advantage of ASIC tool features. These tools significantly simplify the application implementation process, and also provide essential testing and verification opportunities for implementing correct applications on FPGAs.

By applying CAD, it allows designers and customers to implement their applications on FPGAs at a higher level of abstraction, where the normal entries are HDLs (Hardware Design Languages), for example VHDL (IEEE, 1988) and Verilog (IEEE, 1995), or circuit schematics. The CAD flow can assist designers to automatically convert applications as bitstreams to targeting FPGAs. A CAD flow contains a set of subprocesses, and each process is implemented for a particular task during FPGA application mapping processes. One potential advantage of the separation is that these steps can keep the designed application traceable.

3.2 A complete CAD flow for FPGAs

3.2.1 Overview CAD flow for FPGAs

A complete CAD flow for FPGA is complex; Figure 3.1 shows a simplified CAD flow, which contain three major steps. This flow summaries steps necessary for converting a higher level abstraction for an application to a FPGA configuration bitstream.

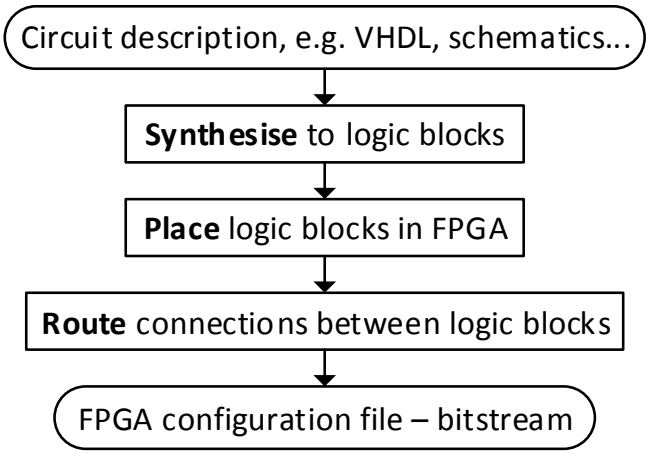


Figure 3.1: A simplified FPGA CAD flow (Betz et al., 1999), it is a sequential process, and contains three steps: Synthesis, placement and routing.

Synthesis and logic block packing

Synthesis means to convert applications, which are implemented in VHDLs, Verilogs and schematics – the higher level abstraction, into a netlist. The netlist contains the technology-independent logic design which can be further realised on FPGAs. There are two goals that a synthesiser has to achieve: Firstly, the number of synthesised logic gates have to be minimised. Secondly, the synthesised circuit has to maintain suitable delays (higher speed). Detailed flow of synthesis and logic block packing is shown in Figure 3.2.

A synthesised circuit might include many redundant logic gates. The technology-independent logic optimisation is the process that optimises the number of logic gates, and produces the most compact logic gates (Brayton et al., 1990; Sangiovanni-Vincentelli et al., 1993; Chen et al., 1992; Cong and Ding, 1994a,b). Subsequently, the optimised logic design is mapped to LUTs, and also FFs (Francis et al., 1991a,b), based on the targeted FPGA architecture, for example, the size of LUT. As discussed in Section 2.3.2, modern FPGAs have grouped a few basic programmable logic blocks into a larger logic cluster, and supplied cluster internal routing resources. Therefore,

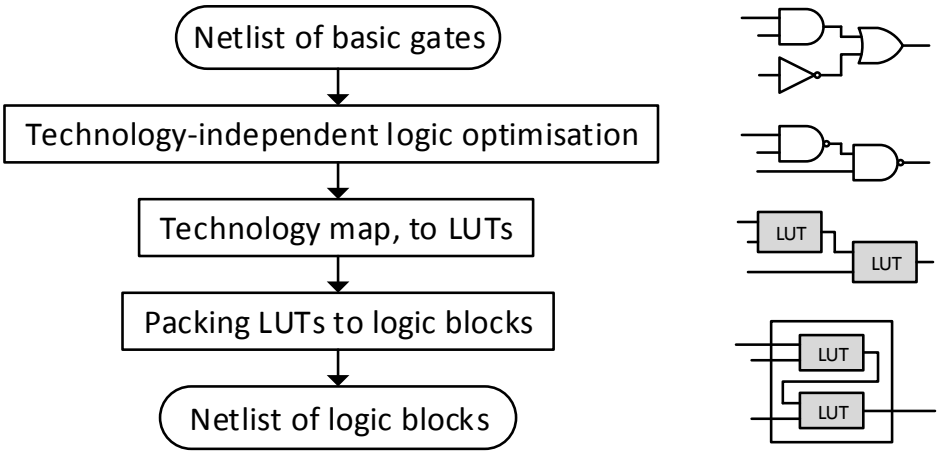


Figure 3.2: Details of synthesis and logic block packing (Betz et al., 1999)

these LUTs (and FFs) have to be further packed into the larger logic clusters, and produce a new netlist. To produce a better routing result on the FPGA, the separation of a circuit has to follow certain rules, and this will be discussed in section 3.3.

Placement

After the synthesis and logic block packing process, a placement process will arrange logic blocks onto the FPGA, so further CAD processes can connect them as a circuit.

In the placement stage, different orientations of the placement can result in different circuit features. If the placement aims to place all logic clusters as close as possible, and optimise connection wire lengths, this will be a wire-length-driven placement. Alternatively, if the placement algorithm balances wire density on the FPGA, this can be a routability-driven placement. Additionally, in most cases, the placement is required to optimise circuit speed, which is a timing-driven placement.

Mainstream research-based placement algorithms can be divided into three types, which are min-cut (Dunlop and Kernighan, 1985; Huang and Kahng, 1997), analytic (Kleinhans et al., 1991; Sigl et al., 1991; Srinivasan, 1991; Riess and Ettelt, 1995; Alpert et al., 1998a) and local search – simulated annealing algorithm (Kirkpatrick et al., 1983; Sechen and Sangiovanni-Vincentelli, 1985; Sechen and Lee, 1987; Sun and Sechen, 1995; Swartz and Sechen, 1995). Of these methods, the simulated annealing algorithm is the most versatile method, where it is easy to add different optimisation goals and constraints. To adapt to different FPGA architectures, the simulated annealing is more suitable. The explanation and comparison, also shown in Table 3.1, of these three types of methods are described as follows:

1) Min-cut method:

   - Basic method: Recursively perform bipartitioning (Alpert et al., 1998b) for a graph – a circuit, and minimise total connection lengths (also known as netlength).
   - Advantage: It is a standard approach – implementation is easy. Execution time is short.
   - Disadvantage: Sequentially cutting a circuit might result in poor following cut solutions if the first cut is minimised (optimal). It is difficult to find a real optimum. Placement is closely linked to a FPGA architecture. Placement objectives are difficult to be incorporated, and usually dependant on weighting approaches. It has a low stability – not a deterministic result.

2) Analytic method:

   - Basic method: Minimise netlength without considering the placement overlaps, and then processing the overlaps – a place might be filled with more functions; It mainly uses the quadratic netlength minimisation (QC).

Table 3.1: The comparison of three placement methods (Vygen, 2002)

Method	Quality	Flexibility	Run Time	Stability
Min-cut	+	+	+	−−
Local search (SA)	+	+ +	−−	−−
Analytic (QC)	+ +	+	+	+

+: Positive, ++: More positive
−: Negative, −−: More negative

- Advantage: QC execution time is short, and QC has a deterministic solution – it is stable. QC is efficient at dealing with small component (logic block) placements.
- Disadvantage: If a circuit has buffers inserted, the results can be changed significantly. Power and delay of a placed circuit can be increased.

3) Local search method:

- Basic method: Minimise netlength via a search algorithm – simulated annealing algorithm (SA).
- Advantage: SA is flexible (no strong link to a FPGA architecture), and implementation is easy. Objectives and constraints can be incorporated by a simple weighting approach.
- Disadvantage: The algorithm convergency is poor. It has a long execution time. The stability is low as using the random search. Objectives are weighted.

Routing

A routing process starts once logic block locations have been determined by placement. The function of routing is to select suitable wire segments and

switch boxes, by programming switch states in the boxes, to connect discrete logic blocks as a circuit.

In order to facilitate the routing of FPGA, a common method is to represent a FPGA routing architecture as a routing resource graph, where it is either a directed graph (Ebeling et al., 1995; Nag et al., 1998) or undirected graph (Alexander et al., 1994) according to which type of routing switches the FPGA uses. To prevent a router (routing method) using too many wires which are beyond the FPGA routing resources, the router is usually set to connect a circuit using the shortest path. Similar to the placement, if the router gives priority to connections on or near the circuit critical path, and routing them using short wires, this will be a timing-driven routing. Otherwise, if the router only considers wire density of routed circuits, it will be known as a routability-driven routing (Betz et al., 1999).

FPGAs routers can be classified in two groups: combined global-detailed (Alexander et al., 1994; Wu and Sadowska, 1994; Ebeling et al., 1995; Alexander and Robins, 1996; Lee and Wu, 1997), and two-step routers. As their name suggests, the global-detailed router routes a FPGA in one step. In contrast, the two-step router will first perform a global routing (Rose, 1990; Chang et al., 1994) to determine which channel segments a logic block connection can be used, and then carry out a detailed routing (Greene et al., 1991; Brown et al., 1992b; Lemieux and Brown, 1993; Lemieux et al., 1997). A FPGA usually has the limited routing flexibility and also the routing constraint. As the global-detailed router routes FPGAs in one step which considers the routing in a global perspective, this router can supply a highly optimised routing solution. Therefore, the global-detailed router is often utilised to facilitate the FPGA routing, and its algorithms, normally path-finding algorithms, can be summarised as follows:

1) Maze router (Lee, 1961):

   - Basic method: Maze router finds a path by propagating a "wave" from a source and waiting it hits a sink. By tracing the sink to

the source, a shorter path can be found.

- Advantage: The algorithm is simple, but cannot guarantee to find the optimal (shortest) path. It allows complex cost functions which involve more routing objectives.
- Disadvantage: It is inefficient when dealing with multiple-terminal nets. The algorithm depends on a 2-D grid and 2-D data structure – complex and more memory used.

2) Dijkstra's algorithm based and A* routers (Dijkstra, 1959; Tessier, 1998):

   - Basic method: Dijkstra's algorithm builds a tree to find the shortest path. In the FPGA routing, Dijkstra's algorithm can combine to maze router algorithm which uses maze router to facilitate a small area routing. Maze router is a special case of A* (or A star), and A* is also similar to Dijkstra's algorithm, where the difference is that A* utilises heuristics to optimise the search.
   - Advantage: Both routers are better for two-terminal nets. The optimality of A* can be adjusted by cost functions.
   - Disadvantage: They have long execution time as they are iteration based, but A* is faster; Dijkstra's algorithm might not find all possible paths.

3) Pathfinder router (Ebeling et al., 1995):

   - Basic method: The Pathfinder router extends maze algorithm, and increases the algorithm speed by considering the routing on an "obstacle-free" environment and reusing routing resources. The algorithm routes nets via iterations of ripping up and re-routing nets (Prado, 2006).
   - Advantage: It produces excellent results, and uses cost functions to guide the search.
   - Disadvantage: The algorithm is complex, and execution time is long.

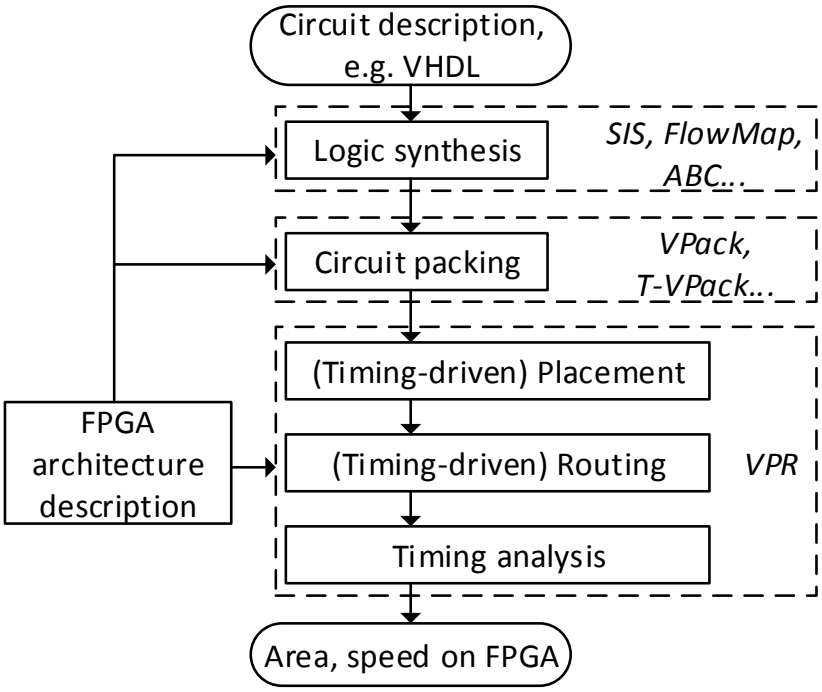


Figure 3.3: A research based CAD flow for FPGAs (Luu et al., 2011)

3.2.2 CAD flows in academic research

In fact, details of commercial FPGA internal implementations and FPGA CAD algorithms are usually unavailable. To research them, it is common to use research based tools. Unlike commercial CAD tools, these are supplied as an integrated developing environment (IDE) which is an all-in-one solution for FPGA application mappings. Research based CAD flow uses a set of tools to form a tool chain (flow), and these tools are focused on one or several subprocesses in the flow. Figure 3.3 shows a research based CAD flow which has been widely used for FPGA architecture and CAD algorithm related researches.

Logic synthesis

The flow begins with circuit descriptions, for example in VHDL scripts, and these text-based VHDL scripts are imported to SIS (Sentovich et al., 1992), FlowMap (Cong and Ding, 1994a) or ABC (Mishchenko et al., 2007) for circuit synthesising and technology mapping. A FPGA architecture description is pre-defined and used in the entire CAD flow. In terms of these synthesis tools, SIS stands for a system for sequential circuit synthesis, and it is defined as an interactive tool for sequential circuit synthesises and optimisations. It was a common academic-based synthesis tool in the early time. FlowMap is an optimised technology mapping algorithm. This algorithm has been proved that it can reduce the LUT network depth by up to 7%, and also decrease the number of LUTs by up to 50% compared with previous methods. Apart from the function of synthesis, ABC, an academic industrial-strength verification tool, also involves verification approaches for testing binary logics in the design of synchronous circuits. These tools, algorithms, are active tools for academics.

Circuit packing

After the synthesis, a circuit netlist is obtained and the netlist will be processed by a circuit packing algorithm, for instance VPack (Betz and Rose, 1997a) and T-VPack (Marquardt et al., 1999), which separates the synthesised circuit to FPGA logic blocks. Based on a new netlist which is produced by the circuit packing, post-synthesis processing can be achieved by VPR (Betz and Rose, 1997b; Luu et al., 2011). Note that the netlists in this flow and in this thesis are all referred to a BLIF format – a logic interchange file (Berkeley, 1992). In this thesis, the circuit packing is the core topic, and previous methods will be further detailed in Section 3.4.

VPR

VPR (Betz and Rose, 1997b; Luu et al., 2011) is short for Versatile Placement and Routing, and is an open-source FPGA CAD tool, which is developed by the University of Toronto for assisting FPGA architecture and FPGA CAD algorithm researches, and can also be used to mapping designs to commercial FPGAs.

This tool allows a researcher to specify a FPGA architecture and fabrication technology as a plain-text file, as well as a circuit netlist which is in BLIF format. Subsequently, using the architecture and netlist files, the VPR will place the netlist described logic functions on the FPGA, and route them as a functional circuit. After placement and routing processes, the VPR will analyse the mapped circuit, and indicate its area and timing information on the FPGA.

Placement in VPR was a simulated annealing algorithm as VPR is designed to deal with various FPGA architectures. The routing uses the global-detailed router, and it is based on "Pathfinder negotiated congestion algorithm" (Brown et al., 1992b; Ebeling et al., 1995; Betz and Rose, 1997b).

VPR supports delay estimations of a mapped circuit. To obtain an accurate delay, the best approach is to use SPICE (Nagel and Pederson, 1973). However, it is impossible to utilise SPICE to measure delays for an entire circuit on FPGAs as the routed circuit is huge which causes an extremely long SPICE execution time – a few minutes for a large logic block. As logic blocks and routing switch boxes on FPGAs are identical components, the VPR only measures different configuration delays of one logic block and one routing switch box via SPICE and stores them as a look-up table (defined in the architecture description file) (Luu et al., 2011). Once their configurations are determined, delays of logic blocks and switch boxes can be quickly obtained. Currently, the delay estimation is focusing on routing wires, where these are other sources of delays. To provide delays, VPR models these wires as RC-trees (Rubinstein et al., 1983; Khellah et al., 1993) and incorporates the

Elmore delay model (Elmore, 1948). In this case, if the length of a wire is identified, its delay can be estimated. By adding logic block, switch box and wire delays, the entire circuit delay is provided.

3.3 Circuit clustering

3.3.1 Definition

In an FPGA CAD flow, circuit clustering, also known as circuit packing in a more general way – refers both circuit clustering and circuit partitioning, is a process to separate a synthesised circuit as sub circuits, and allow each separate circuit to be filled by FPGA logic clusters, the CLBs, without conflicting to CLB hardware constraints – for example CLB size, input number, where each sub circuit can be enclosed and connected (routed) by CLB and CLB internal routing resources (Betz and Rose, 1997a). On the other hand, the circuit clustering also indicates how better to group the basic programmable logic blocks (their functions), the BLEs, which each BLE is paired from LUTs and FFs based on their connections and this process is known as pattern match (Betz and Rose, 1997b,a), into the CLBs – BLE combinations. Figure 3.4 shows the process of circuit clustering.

The reason of using the circuit clustering to describe the process of circuit separation, is that most methods are based on greedy algorithms to directly group CLBs without using graph-based partitioning techniques. Betz and Betz's colleague (1999) have commented that if a circuit is first separated as a few large blocks, and each large block is recursively divided as small pieces – graph-partitioning-based techniques, this will be referred to as circuit partitioning. In contrast, the circuit clustering means that an entire circuit is separated into many small pieces directly – into clusters. As a result, according to the proposed methods in this thesis, using circuit clustering can more precisely reflect the process of separating circuits to FPGA CLBs.

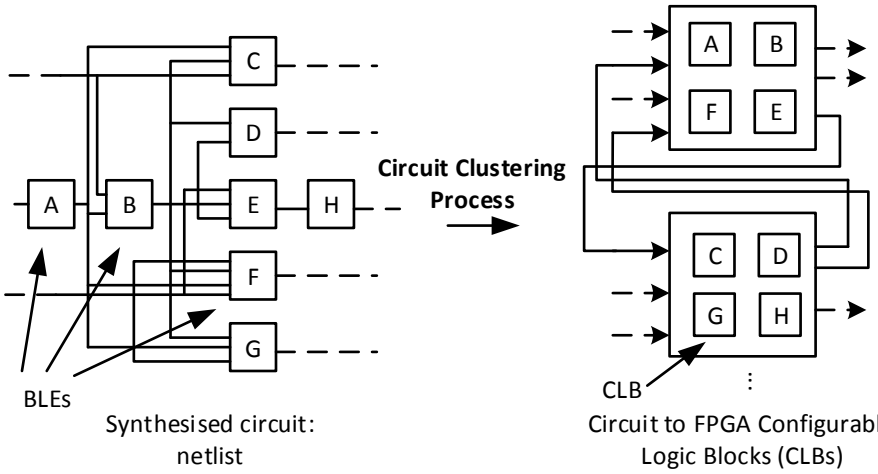


Figure 3.4: An example to show the process of circuit clustering (Marquardt, 1999)

Note that, for simplicity, circuit separation (packing) has been generalised as circuit clustering to review different methods.

3.3.2 Significances and limitations

Circuit clustering is a fundamental process in the CAD flow, and the quality of the clustered circuit can significantly impact placement and routing processes which have a knock-on effect on the performance of a mapped circuit. Similar to other post-synthesis CAD processes, if the circuit clustering optimises FPGA routing wire density, for example the number of tracks used, which allows a larger circuit to be routed on a resource limited FPGA, it will be a routability-driven circuit clustering. If it reduces the mapped circuit delay and improves its speed, this will be a timing-driven circuit clustering. As well as routability- and timing-driven, there is also power-driven circuit clustering. Note that the "x"-driven means the clustering priority is on "x". Since the circuit clustering is the first stage in post-synthesis processes, if a circuit has not been well clustered, its performance can be difficult to improve via

subsequent CAD processes.

When clustering BLEs into CLBs, the circuit clustering method can produce various solutions, and even if the clustered circuit has the same number of clustered CLBs among solutions, their BLE combinations within the CLBs and the CLB interconnects can be different. In addition, a circuit clustering process is also limited by clustering constraints, for example, the CLB input number and the BLE number in each CLB. On the other hand, the circuit clustering method has to optimise for instance, routability, timing and power. This indicates that it is difficult to find a unique solution for circuit clustering problem, or even there are no best solution existing. From the computational complexity perspective, the circuit clustering is a grouping problem, and similar to multi-bin packing problem – packing a set of items into bins – also a well-known NP-hard problem (Bovet and Crescenzi, 2006; Fukunaga and Korf, 2007), but with more constraints and requirements. This means that this type of problems would not be effectively solved by an algorithm in a polynomial time. Therefore, an exact algorithm would not be existing, and it is "hard". Note that the circuit clustering in this thesis mainly refers to the routability- and timing-driven circuit clustering.

3.3.3 Requirements of circuit clustering

In general, basic requirements of circuit clustering, which refer to routability-driven circuit clustering, are as follows: Firstly, a clustering method is required to cluster all BLEs, the synthesised circuit, into FPGA CLBs while maximising CLB utilisation – using fewer CLBs. Secondly, the clustering method has to reduce (optimise) the CLB interconnects, known as nets or CLB exposed nets, where the more circuit connections that are included in the CLBs, the fewer interconnects appear between CLBs after the clustering. Figure 3.5 shows an example and helps to explain why fewer CLB interconnects are better, where routing a clustered circuit that has fewer CLB interconnects can use fewer routing tracks (Marquardt, 1999; E.Bozorgzadeh et al., 2001).

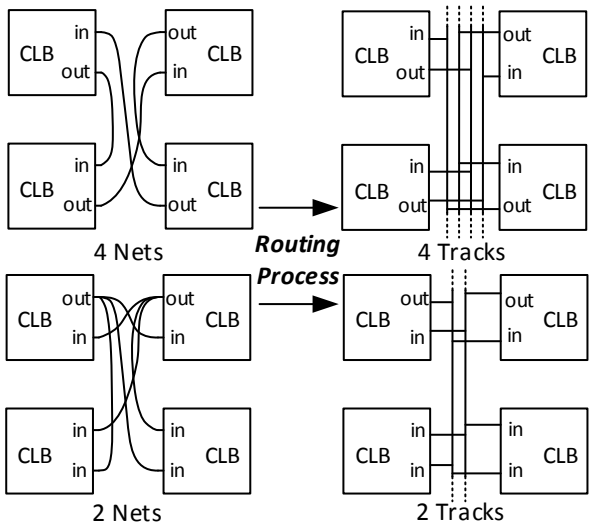


Figure 3.5: Mapping results under different CLB interconnect numbers, when a clustered circuit has fewer CLB interconnects (nets), the routed circuit can have fewer tracks (Marquardt, 1999). Therefore, a narrow channel width is used on the FPGA.

Problem formulation

Based on RPack (E.Bozorgzadeh et al., 2001), a routability-driven circuit clustering method, and the cluster-based island-style FPGA model, where its CLB has the following parameters: I – CLB input number, N – BLE number within the CLB and one clock, the circuit clustering problem can be formulated – giving a synthesised circuit that has a set of BLEs: $B = \{b_1, b_2, ..., b_n\}$, and a set of empty CLBs which are from a FPGA: $C = \{c_1, c_2, ..., c_m\}$. When clustering BLEs into CLBs, the following conditions have to be respected: Equations 3.1-3.4:

$$\text{INPUT}(c_i) \leq I \quad i = 1, 2, ..., m \tag{3.1}$$

$$\text{BLE}(c_i) \leq N \quad i = 1, 2, ..., m \tag{3.2}$$

$$c_i \cap c_j = \emptyset \quad i, j = 1, 2, ..., m, \ i \neq j \tag{3.3}$$

$$\sum_{i=1}^{m} \mathrm{BLE}(c_i) = B \quad i = 1, 2, ..., m \tag{3.4}$$

The more CLBs the clustered circuit has, the more FPGA area can be used. Additionally, the more CLB interconnects exist, the higher the usage of routing tracks.

$$\mathrm{BLE}(c_i) \rightarrow N \ (minimise \ |C|) \quad i = 1, 2, ..., m \tag{3.5}$$

$$\cup_{i=1}^{m} Net(c_i) \rightarrow 0 \quad i = 1, 2, ..., m \tag{3.6}$$

Hence, a routability-driven circuit clustering method usually optimises two aspects of a clustered circuit, which are shown in Equations 3.5-3.6: Equation 3.5 represents circuit absolute areas on a FPGA. Note that actual FPGA area usages can be increased by excessive inputs and outputs (pads) of the circuit, but, in general, the fewer CLBs there are, the less area used on the FPGA. As shown in Figure 3.5, the CLB interconnect (net) number also has to be as small as possible; this condition is represented by Equation 3.6. Since the above two parameters are important, and affect FPGA whether or not successfully implementing a target circuit, as a result, these two parameters can be considered a "golden rule" to evaluate the quality of the clustered circuit.

Extending the problem

Although a well clustered circuit has fewer CLB and CLB interconnects, the routed circuit can still have a worse routability or timing (Chen and Cong,

2004). The main reason is that the clustered circuit involves many non-suitable CLB interconnects, where the interconnects are the set of common connections of the clustered CLBs which are determined by the CLB internal BLEs. For this reason, the final routed circuit might have spiral wires (Chen and Cong, 2004) which extend total wire lengths (increase delays) or use more routing tracks in FPGA channels. On the other hand, though there are fewer CLB interconnects, this cannot guarantee that all the circuit on or near critical path connections are well arranged (Marquardt, 1999), so larger circuit delays might be caused. This is also the reason that the timing-driven circuit clustering methods (Marquardt et al., 1999; Bozorgzadeh et al., 2004; Feng, 2012) usually have a circuit static timing analysis (Hitchcock et al., 1983) before clustering, and clusters the circuit on or near critical path connections preferentially in CLBs, where the CLB internal routing resources have short wires, so the delay is small.

At the beginning of this section, it was mentioned that there are routability-driven, timing-driven and power-driven circuit clustering methods, so circuit clustering is a comprehensive problem. For example, if a clustered circuit has fewer interconnects and uses short wires (wire lengths), when it is mapped onto a FPGA, it first indicates that the circuit is routable, and then suggests that this clustering method might also optimise the timing or power. Therefore this is the reason that circuit clustering methods usually start with the routability. However, based on the above discussions, it can also be concluded that, to evaluate a circuit clustering method, only using the "golden rule" is not enough. As a result, to fully evaluate a circuit clustering method, its clustering result has to be implemented on a FPGA, or using VPR as a FPGA simulator. On the FPGA, the routed circuit area, channel width reflects the routability, the implemented circuit critical path delay indicates the circuit speed, as well as the total routing wire lengths implies the power consumptions.

3.3.4 MCNC-20 benchmark

The MCNC-20 (Microelectronics Center of North Carolina) benchmark suite is a popular in benchmarking the performance of circuit clustering methods (Yang, 1991). "20" means that this suite contains 20 benchmark circuits, and these circuits are from the ISCAS'85 (sps, 1985), ISCAS'89 (Brglez et al., 1989) and other industry or academic resources. In circuit clustering method research (VPR based), these benchmarks are usually synthesised as the BLIF netlist files. The benchmark circuit details have been listed in Appendices in Tables A.1 - A.2. In this suite, after synthesis, where the LUT size K is set to 4, when it performs pattern match, the largest benchmark has more than 8,000 BLEs. In contrast, the smallest benchmark has around 1,000 BLEs. Note that these benchmarks are used exclusively in this work to assess the efficiency of the new circuit clustering algorithms developed.

3.4 Previous methods

This section reviews a few well-known circuit clustering methods (algorithms). For most methods, they are targeting the CLB-input-bandwidth-constraint cluster-based island-style FPGAs, and having the FPGA model k, N, I, M_{clk} set to $4, 8, 18, 1$ respectively, these parameters refer to Chapter 2. In contrast, there are also methods for the input-bandwidth-free CLB FPGA, where the I is set to 32 ($k \times N$) representing no constraint.

According to these methods, their clustering perspectives can be mainly classified as bottom-up and top-down methods. Bottom-up means clustering a circuit by moving BLEs into CLBs sequentially, and CLBs are constructed one by one. Top-down refers to using circuit partitioning methods, which view a circuit from a global perspective, and separates the circuit by recursively partitioning it until each part of the circuit is able to fit to FPGA CLBs.

3.4.1 Bottom-up methods

Bottom-up FPGA circuit clustering methods were initially introduced in VPack (Betz and Rose, 1997a) – a heuristic algorithm, and was the circuit clustering method used in early VPR based CAD flows. VPack was also an initial FPGA circuit clustering algorithm that considered the CLB utilisation and the circuit routability. Subsequently, a number of circuit clustering algorithms were developed based on the VPack algorithm, involving more clustering metrics, for example the timing.

In bottom-up methods, a seed BLE has to be selected via a specified method. Then, the seed is directly moved into an empty CLB. To cluster more suitable BLEs in the CLB, these algorithms usually use an attraction function, also known as cost function, to determine which is the best candidate BLE that can be moved next, and the attraction function is weighted by a number of clustering objectives – which are used to accomplish different clustering metrics routability, timing, etc. The value of the function is known as the "gain". Bottom-up circuit clustering processes can be summarised in the following steps:

1) Select a seed BLE, and copying (moving) the seed BLE to an empty CLB.

2) Treat all BLEs within the CLB as an entity to calculate gains of unclustered BLEs to this CLB by an attraction function.

3) Rank gains and determine the highest gain BLE (higher gain BLE is better).

4) Check the CLB constraints, if it can fill the highest gain BLE, then clustering (moving) it into the CLB, and returning to STEP 2. Otherwise, storing the CLB and staring to cluster next CLB.

The clustering process is iterative, and refers to the greedy algorithm (Cormen et al., 2009) (a type of heuristic algorithms) – CLB are built sequen-

tially and continuously until all BLEs are clustered into CLBs. In addition, algorithms like VPack (Betz and Rose, 1997a) and T-VPack (Marquardt et al., 1999) also implement a hill-climbing algorithm to deal with zero-gain BLEs. The differences in these bottom-up methods usually indicate the different seed BLE selection methods and attraction function designs.

VPack

The input of VPack (Betz and Rose, 1997a) is a BLIF format file, and VPack will perform the pattern match and cluster BLIF file represented circuit into CLBs, then generate a new netlist (also in BLIF format). Detailed VPack algorithm is introduced in Section 5.2.

$$Attraction(B) = |Nets(B) \cap Nets(C)| \tag{3.7}$$

To construct a CLB, VPack will select a seed BLE according to input and output number of unclustered BLEs. A BLE which has the most inputs and outputs will be the seed. The reason for using this BLE is that it potentially increases the probability of including more connections within a CLB. The attraction function in VPack, shown in Equation 3.7, indicates the number – the gain – of common connections between a constructing CLB and other unclustered BLEs; a higher gain BLE suggests that a BLE has more common connections with the CLB. The main aims of VPack are to optimise CLB usages and increase CLB included connections. Detailed clustering flow of VPack is shown in Appendices in Algorithm A.1.

T-VPack

T-VPack (Marquardt et al., 1999) is an extension of VPack, and it is short for Timing-driven VPack which optimises clustered circuit delays. In T-VPack, its seed selection is the same as VPack. The difference from VPack is that, before

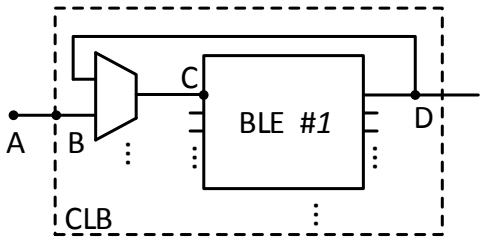


Figure 3.6: The CLB, BLE delay model used in T-VPack

the clustering process, T-VPack will analyse circuit timings and incorporate the timing information in its attraction function.

Without considering FPGA routing delays, T-VPack models, as shown in the Figure 3.6, CLB and BLE delays in three stages:

1) From A to B: The delay between the routing channel and the CLB.

2) From B to C: The delay of CLB internal routing resources.

3) From C to D: The delay of the BLE itself.

Based on this delay model, using Hitchcock and Frankle proposed timing analysis methods (Frankle, 1992; Hitchcock et al., 1983); T-VPack will traverse a synthesised circuit in two directions, which are from its inputs (FF outputs considered as input) to outputs and its outputs (FF input considered as output) to inputs, which determine two (signal) time points of each circuit connection: $T_{arrival}$ and $T_{required}$ respectively, and further deriving the time difference – see Equation 3.8 – a time "slack" of each connection is obtained.

$$Slack(net) = T_{required}(net) - T_{arrival}(net) \tag{3.8}$$

During the slack calculation, T-VPack recodes the maximum slack as $Maxslack$. Therefore, the criticality of each circuit connection can be repre-

sented using Equation 3.9, where the higher criticality of a connection means that the connection is near or on the circuit critical path.

$$Connection\_Criticality(net) = 1 - \frac{Slack(net)}{Maxslack} \tag{3.9}$$

When clustering BLEs into CLBs, these higher criticality connections have to be reasonably arranged, for example, clustering some of them in CLBs for reducing circuit-on-FPGA delays. In T-VPack, it has weighted the timing information in the attraction function, as shown in Equations 3.10-3.11.

$$\begin{aligned} Attraction(BLE) = \alpha \times Criticality(BLE) + \\ (1-\alpha) \times \frac{Nets(BLE) \cap Nets(CLB)}{G} \end{aligned} \tag{3.10}$$

$$G = \#BLEinputs + \#BLEoutputs + \#BLEclocks \tag{3.11}$$

Where $Attraction(BLE)$ is the gain of a BLE to the constructing CLB, and $Criticality(BLE)$ is based on the highest criticality connection on the BLE. α is a proportionality coefficient, which adjusts the ratio between the timing and the common connections when producing the gain. In Marquardt's experiment (Marquardt et al., 1999), they indicate that $\alpha = 0.75$ can produce the best clustered circuit.

RPack

RPack (E.Bozorgzadeh et al., 2001) is short for Routability-driven FPGA circuit clustering algorithm and it is an improved VPack algorithm. Reducing CLB interconnects is a major feature of this method. The initial RPack (and also T-RPack (Bozorgzadeh et al., 2004) – timing-driven RPack) algorithm is similar to VPack and T-VPack. The only difference is that the attraction

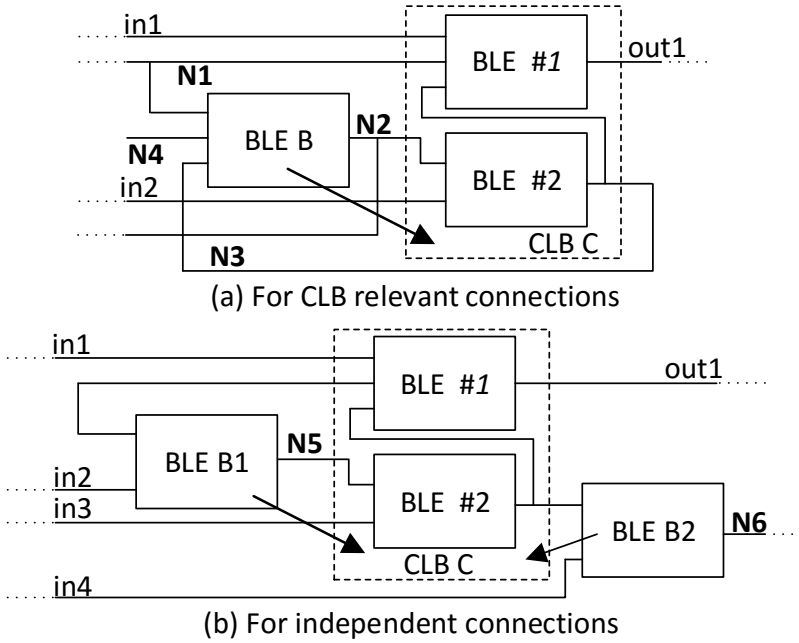


Figure 3.7: An example circuit for showing connection gains in the RPack, CLB relevant connections, (a), mean these connections share current CLB connections. Independent connections, (b), mean these connections either can be absorbed in the current CLB or not sharing connections to the current CLB.

function of RPack is improved.

$$Gain(B,C) = f(Nets(B), Nets(C)) = \sum_{i \epsilon Nets(B)} g(i, Nets(C), B) \qquad (3.12)$$

$$g(i, Nets(C), B) = \begin{cases} 1 + f_{in}(P(i,B), P(i,C)) \\ \quad + f_{out}(P(i,B), P(i,C)) & i \epsilon Nets(C) \\ \\ -1 \times T(i,B) & otherwise \end{cases} \qquad (3.13)$$

Table 3.2: Gains of each connection in Figure 3.7 circuit, gain calculations are based on Equations 3.12 - 3.13

Connection	In-pin Gain	Gain for Output Congestion	Edge Gain	Total Gain
N1	0	0	1	1
N2	1	0	1	2
N3	0	1	1	2
N4	-1	0	0	-1
N5	1	1	1	3
N6	0	0	0	0

In RPack, its attraction is defined as in Equations 3.12, where the RPack is designed to analyse the relationships between each connection of *BLE B* and connections of constructing *CLB C*, and accumulating gains of each connection of the BLE: $g(i, Nets(C), B)$. Detailed gain calculations are shown in Equation 3.13.

In Equation 3.13, the BLE connection gain has been divided into three types, which are: Firstly, $f_{in}(P(i,B), P(i,C))$ is called an in-pin (input-pin) gain, and it is produced on input connections of *CLB C*. Secondly, $f_{out}(P(i,B), P(i,C))$ is the gain that reflects output congestions of *CLB C* when it clusters the *BLE B* into *CLB C*. Note that when a BLE connection is already in CLB, a fixed gain "1" is added, and it is known as "edge gain". Thirdly, $-1 \times T(i,B)$ is the other in-pin gain that indicates a new input added to *CLB C*; If an input connection, which does not exist in *CLB C*, can be fully absorbed by *CLB C*, $-1 \times T(i,B)$ will return as "0"; otherwise, it will be "-1". To demonstrate how gains are calculated for connections, Figure 3.7 shows an example circuit, and the calculated gains of connections are listed in Table 3.2.

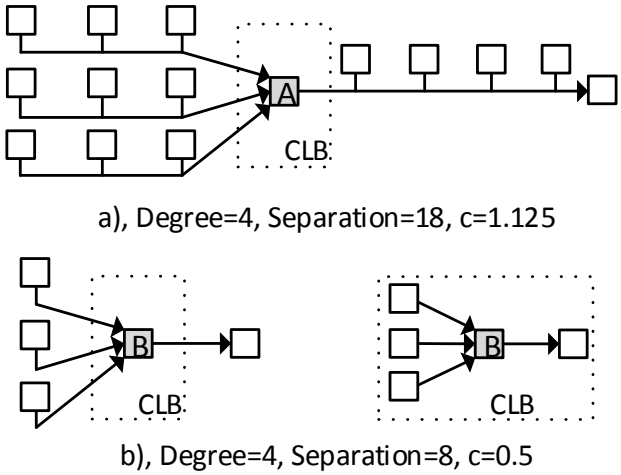


Figure 3.8: *BLE A* has a larger "*c*", in contrast, *BLE B* has a smaller "*c*". For example, a CLB can accommodate 5 BLEs, using smaller "*c*" BLE as a seed can absorb 4 connections in a CLB; squares represent BLEs. When uses *BLE A* as a seed, no matter how to cluster 5 BLEs, the CLB cannot include 4 connections (Singh and Marek-Sadowska, 2002).

iRAC

iRAC (Singh and Marek-Sadowska, 2002) was an outstanding circuit clustering method to include connections in input-bandwidth-constraint CLBs, whilst at the same time using fewer tracks in the routing. Similar to VPack, it deals with BLIF files. Using iRAC to cluster circuits can significantly reduce CLB interconnects, so the post routing has fewer channel tracks and shorter wires, which improve circuit routability and reduce power. However, the disadvantage of iRAC is that the clustered circuit has more clustered CLBs, which puts it at risk of using more FPGA areas.

The iRAC models the connection connectivity of circuits based on graph theory (Gross and Yellen, 2005), and produces a new seed BLE selection method and a new attraction function. To find the best seed, the connectivity factor (c) of a BLE is defined, Equation 3.14.

$$c = \frac{separation}{degree^2} \tag{3.14}$$

Where the *separation* signifies terminal number of a connection, and *degree* is the different connection number of a BLE. Figure 3.8 illustrates how the connectivity factor is used to determine a best seed of a CLB; when c is smaller in a BLE, as shown in Figure 3.8 b), and assuming the CLB can accommodate 5 BLEs, this means that using this BLE (*BLE B*) as seed can absorb more connections in a CLB.

To continuously select a suitable *BLE X* that clusters into *CLB C*, iRAC accumulates all connection gains of each unclustered BLE, and defines attraction function as shown in Equation 3.15, where $\omega(x)$ is equal to $2/r$, r is terminal number of a connection x, x is the CLB already included connection, and a_x represents the number of terminals of the connection x. The constant n is the CLB size (referred to as N in the FPGA model). k is an adjustable coefficient, and its value is set by incorporating Rent's rule (Landman and Russo, 1971; Donath, 1979). The main function of k is to maintain that if a candidate BLE is clustered in the CLB, it will not add more interconnects between CLBs. Note that Rent's rule will be further introduced in Section 7.3.3.

$$G(X, C, x) = k[2n\omega(x) \times (1 - a_x)] \tag{3.15}$$

MO-Pack

The MO-Pack (Rajavel and Akoglu, 2011) is designed to improve the clustered circuit for several performance metrics after routing, for example timing, power and routability. MO is short for Many Objectives, and the MO in this method means that its attraction function involves a few clustering objectives. The entire algorithm flow is similar to other bottom-up methods, and seed selection method is based on iRAC. Equation 3.16 shows the attraction function of

MO-Pack. Note that this MO is different from the Multi Objective (MO) which is used in this thesis. The Many Objectives are implemented by a weighted cost function, and this method is coarse. The new MO (Multiple Objective) methods that are produced in this thesis, Chapters 6-8, allow more sophisticated objectives are involved into the clustering process, and the multiple objective evaluation is based on the principle of Pareto optimality (Pareto, 1906).

$$\begin{aligned} Cost(B,C) = (\alpha \times criticality) + \\ (1-\alpha-\gamma)\frac{|Nets(B) \cap Nets(C)|}{G} + \\ (\gamma \times k \times \sum_{i \epsilon Nets(B)} \frac{1+p_{ic}}{p_i}) \end{aligned} \tag{3.16}$$

Where: B is an unclustered BLE, C is a CLB under construction, α is a trade-off coefficient for delay optimisations, γ is a coefficient for optimising the number of connections included in the constructing CLB, G normalises the shared connection number in the CLB, P_{ic} is the number of connections that are already in the CLB, P_i is the total terminal number of connection i in the CLB and k is equal to either "1" or "1.15" for the multi-terminal or two-terminal connection respectively.

3.4.2 Top-down methods

The popularity of the top-down FPGA circuit clustering method has rapidly increased in recent years as it views the clustering process from a global perspective. To facilitate a top-down circuit separation, most circuit clustering methods in this category are based on graph partitioning approaches. The reason is that a circuit can be treated as a hypergraph: $G(V, E)$, where the set V, vertices of a graph, represents circuit components and the set E, hyperedges of the graph, signifies circuit connections.

A simple partitioning process will refer to the graph bipartitioning method – giving a circuit and treating it as a hypergraph, then the method cuts the hypergraph into almost half, where, in the partitioning process, the method optimises vertex combinations in two parts and aims to find the minimised edges (or known as min-cut) between these two parts (Alpert et al., 1998b). By repeating the bipartitioning process for already cut parts, a large circuit is clustered into pieces, and fitted in a FPGA CLB.

Currently, the top-down circuit clustering methods usually deploy hMETIS (Karypis and Kumar, 1999) hypergraph partitioning algorithm, and these methods can be viewed as different extensions of the hMETIS. hMETIS is a standalone tool for performing a k-way graph partitioning based on a multilevel paradigm. Compared with traditional graph partitioning methods, the k-way multilevel partitioning can not only produce well optimised solutions, but also achieve solutions quickly even if a graph has a large number of vertices. Note that, similar to other FPGA circuit clustering methods, the top-down circuit clustering methods also deal with BLIF format files.

LIP6-ASIM Laboratory's circuit clustering method

The method presented in Marrakchi's paper (Marrakchi et al., 2005) does not have a specific name. This clustering method is designed for the CLB input-bandwidth-constraint CLB FPGAs, and its clustering process can be divided into two major steps: At the first step, a circuit is partitioned into sub circuits by the hMETIS, and each sub circuit can be fitted or nearly fitted to a FPGA CLB. "Nearly" means that each sub circuit might be sightly larger than the CLB size, or clustering constraints cannot be met. In the second step, an extra BLE moving process is implemented, which moves or swaps BLEs in the partitioned sub circuits and ensures each sub circuit can fit to CLBs. To maintain a high solution quality, the BLE moving or swapping in the sub circuits is evaluated by an attraction function, and the function is similar to that of the RPack. Since there is an extra step in this method,

it will degrade the quality of hMETIS produced results. However, its final solution is still better than most of the bottom-up methods in reducing CLB interconnect number, routing tracks and wire lengths, but the circuit delay is not considered.

PPack, T-PPack

PPack (and T-PPack, which is short for timing-driven PPack) (Feng, 2012; Feng et al., 2014a) attempts to solve the circuit clustering problem for the bandwidth-constraint-free CLB FPGAs, which means that all LUT inputs can be accessed on the CLB. In this case, the CLB input number is $N \times k$, for example, the CLB input number is 32, where $N = 8$ and $k = 4$. Since the constraint has been removed, the clustering problem can be solved using a circuit partitioning method, and PPack is such a method. PPack utilises hMETIS to directly cluster a targeted circuit. If a partitioned circuit cannot be fitted into a CLB due to being oversized, PPack will emulate hMETIS method to move unsuitable BLEs. Therefore, PPack is one of the best methods that reduces CLB interconnects of a clustered circuit, and also has fewer routing tracks and wire lengths in the post routing. As tracks and wire lengths have been largely reduced, PPack also has better performances for reducing circuit delays. T-PPack is an improvement of PPack which is implemented for speeding up circuits on FPGAs. Similar to other timing-driven circuit clustering methods, a circuit timing analysis is preformed before the partitioning and timing information is weighted on edges of a hypergraph (the targeted circuit). When partitioning the hypergraph, hMETIS can preferentially arrange on or near circuit critical path connections into CLBs.

3.4.3 Other methods

Methods in this category refer to two different types of methods, which are hybrid methods and post-routing-assisted methods. “Hybrid” means that

this type of method combines the bottom-up (greedy algorithms) and top-down (circuit partitioning methods) methods, and "post-routing-assistant" indicates that the methods or the combined methods produce better solutions by incorporating the placement and routing processes in the CAD flow. The reason is that using the CLB number and CLB interconnect as clustering objectives might not fully evaluate the quality of a clustered circuit as discussed in 3.3.3. Utilising the placement and routing processes in the clustering method and executing them as a loop, it can continuously adjust or optimise a solution based on real mappings.

HDPack (Chen et al., 2007) is a typical example of the hybrid method category. At the start, HDPack uses the circuit partitioning method to preferentially cluster BLEs into small sub circuits, where they are better for including small (fewer terminals) connections. Then these small sub circuits (each sub circuit might have only 1 BLE) are further clustered using seed-based (bottom-up) methods. In addition, the HDPack also incorporates placement process in the CAD flow, which can roughly determine which regions are more congested based on a FPGA model, and extra adjustments can be produced for the clustered circuit. Similar to HDPack, which involves the placement process, Un/DoPack (Tom et al., 2006) and T-NDPack (Liu and Akoglu, 2009) involve the entire mapping process – post synthesis processes, and uses either pure bottom-up and top-down methods or hybrid methods. Un/DoPack and T-NDPack also introduce the concept of depopulation (Tom et al., 2006) in their methods – depopulation means that, in a routing process, whitespace will be inserted to the congested regions of the targeted FPGA, and all CLBs in the congested region will be unpacked and re-packed, where some BLEs are packed to the whitespace. Therefore, CLBs in that region will not be "too-full", and it essentially relieves routing congestions.

3.4.4 Advantages and disadvantages

It is notable that bottom-up methods are popular in solving the FPGA circuit clustering problem. This is due to the fact that the bottom-up circuit clustering methods are easy to implement, and useful solutions can be obtained quickly. In addition, the clustering objective can be weighted in their attraction functions. As the seed BLE selection and attraction function exist in these methods, where it is uncertain whether or not the above two functions can supply the best BLE in each clustering step, the produced solution is usually local-optimal only. The same problem is also presented in their deterministic results. Although BLE selection methods, both for the seed selection and attraction function, are improved in some methods, for example iRAC, the best BLE might not be unique, so the deterministic result can imply that the greedy algorithm produces worse solutions. Weighting objectives in a single attraction function is able to meet the multiobjective optimisation needs, but the simple weighting can also destroy the proportionality between objectives. As well as this, since VPack, the hill-climbing algorithm is widely utilised in bottom-up methods – however, this extra algorithm might have limited improvements on results.

In contrast, the top-down FPGA circuit clustering methods have also been proven to be able to solve this problem. As the top-down method considers the targeted circuit from a global perspective and also focuses on circuit connections, the clustered circuit can be optimised on CLB interconnects after clustering. This is the reason that these methods can produce better solutions than the bottom-up method. However, using graphs to cluster a circuit can be difficult to involve clustering constraints. Although there are methods that combine both the top-down and bottom-up methods and use the top-down method as the first step, the quality of the results is usually decreased from the second step – the bottom-up methods. Apart from the two major types of methods, there are also methods that incorporate the placement and routing, but the optimisation is still based on a complex weighted cost functions, where optimising one objective might affect another

objective.

3.5 Summary

This chapter first reviews why the CAD is important, and then introduces a typical CAD flow for FPGAs. In addition, a research based CAD flow is highlighted. Subsequently, the chapter focuses on the circuit clustering process in the FPGA CAD flow. Requirements and significances are introduced for that process. A few classic FPGA circuit clustering methods have been described, and the advantages and disadvantages of these methods are discussed. Based on the discussion, the main issues are extracted as follows: First is the clustering order. If the circuit clustering method clusters a circuit from a global perspective, the clustered solution can be evaluated from the global perspective. This means an optimal solution can be identified. Second is the seed and BLE selections in the bottom-up methods. Using the above bottom-up circuit clustering methods, the performance of clustered circuits is limited by the seed and BLE sections. Third is the Multiple Objective approach. Although current circuit clustering methods start to consider multiple performance metrics of a clustered circuit, the Multiple Objective approach is usually coarse, which uses a weighted cost function. In the next Chapter 4, a new optimisation method, MultiObjective Genetic Algorithm (MOGA), is introduced. By using this method, a set of new clustering methods are proposed. These methods potentially solve the issues that are identified in the classic circuit clustering methods.

Chapter 4

Evolutionary Computing

4.1 Evolutionary Computing (EC)

The concept of Evolutionary Computing (EC), or also known as evolutionary computation, can be traced back to the later 1940s. For example Turing proposed "genetical or evolutionary search" in 1948 and Bremermann's experiment of "optimization through evolution and recombination" (Bäck et al., 1997; Fogel, 1998; Eiben and Smith, 2007), and it is a subfield of Artificial Intelligence (AI) in computer science. EC is actually defined as a set of Evolutionary Algorithms (EAs), and these EAs are based on Darwinian evolutionary principles. This is the reason that these algorithms are called EAs. EAs can be classified in the family of trial and error problem solvers (Bei et al., 2013), and considered as automatic problem solvers and optimisation methods via a stochastic character or metaheuristic. Since the 1960s, EC is further described as four dialects, which are: Evolutionary Programming (Fogel et al., 1966), Genetic Algorithms (GAs) (Holland, 1973), Evolution Strategy (ES) (Rechenberg, 1973) and Genetic Programming (GP) (Koza, 1990). Due to the nature of evolution, it means that EAs are suitable solvers in many cases for complex problems, which include scientific and industrial areas (Greiner et al., 2013), such as design optimisation, result searching,

system control and event scheduling. In addition, EA is still an active topic for research (Sun et al., 2014). The remarkable advantage of EAs is that they can produce better solutions for a targeted problem, but without the need for fully understanding the problem.

4.2 The inspiration of nature

In 1895, Charles Darwin indicated that the root of a large number of different species existing on our planet was based on the principle of mutation and natural selection (Darwin, 1859) – the theory of Darwin's natural selection. This theory was considered a scientific explanation for the evolution of different forms of life (Coyne, 2009). It was said that there were around 2 million of classified species, which did not include unknown ones (May et al., 1988). Evolution not only produces new species, but also forces the low level species to evolve as high level species which are much more suitable for their environment. During evolution, when individuals reproduce, the offspring with characteristics best suited to their natural environment are the most likely to survive and reproduce, which can eventually lead to new species being born. This process can be extracted as a useful bio-inspired model for solving engineering problems.

4.2.1 The theory of Darwin's natural selection

From a macroscopic perspective, the theory of Darwin's natural selection can be briefly explained as follows: given a set of members of a species, as well as a natural environment, the number of members increases via reproduction at a rate influenced by the environment. In this case, the growth of such individuals in a population might be limited by necessary resources which are provided by the environment, for example food. Consequently, individuals which are not able to adapt the environment tend to die off before reproduction. The

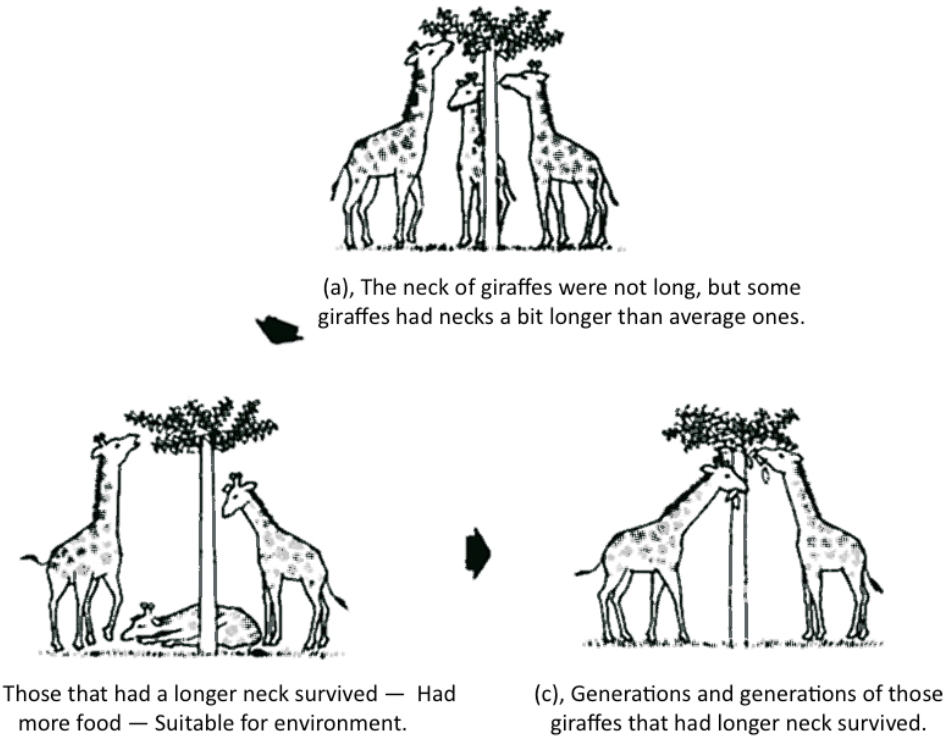


Figure 4.1: Why giraffes have long necks, explanation based on the theory of Darwin's natural selection (Rikizo and Suzuki, 1974).

individuals that best adapt to the environment survive, and continuously replace the less-adapted individuals from their gene pool. This process reflects the mechanism of natural selection, or the "survival of the fittest" in Darwin's words, and how the individuals fit to the environment is termed as fitness.

The fitness can be determined by the individual's physical and behavioural features, and also its physiological features, where these features are defined as phenotype – the observable characteristics or traits of an organism. During the evolution, "stronger" individuals have higher fitnesses. Hence, they are most likely to produce offspring, and pass their genetic information, which is known as genotype – the collection of individual's genetic information, to the next generation. In this process, the higher fitness individuals' genetic information is inherited by their offspring, and so the offspring have similar traits and

characteristics as their parents. Apart from the inhabitation which is from the progenitors to the progeny, a small amount of inhabited genetic information is also changed during reproduction by variations, which are caused by random mutation. These small variations cause differences in the offspring, and further result in different fitnesses of offspring – where some individuals might be better than their parents, but some might not. Ultimately, evolution is converged to a balance point (speed) by the reproduction and selection, and it is continuously pushing the evolution. Figure 4.1 shows an example that helps to explain why giraffes have long necks under evolution based on the theory of Darwin's natural selection.

4.2.2 Basic concepts of evolution

Chromosome and gene

To introduce the chromosome and gene, we have to start from DNA. DNA is short for DesoxyriboNucleic Acid, and it is the basic storage of entire genetic information. In general, DNA comprises of a long chain of base pairs in which each pair is a carrier of some genetic information. The genetic information is stored as DNA, and exists in the nucleus of an organism cell which is known as the genome. The genome is further separated into a few chromosomes. Note that, for some organisms, they might have a single chromosome in the nucleus, other than the genome. Compared with a chromosome, a gene or allele refers to a series of base pairs which is determined by the phenotypical traits, and its locations on the chromosome are called locus. In evolution, different genotypes can be created by changing an allele on a locus of a genotype, and the phenotype is the genotype's physical realisation, where it reflects specific behaviours of the genotype. Figure 4.2 illustrates chromosome and gene of an organism.

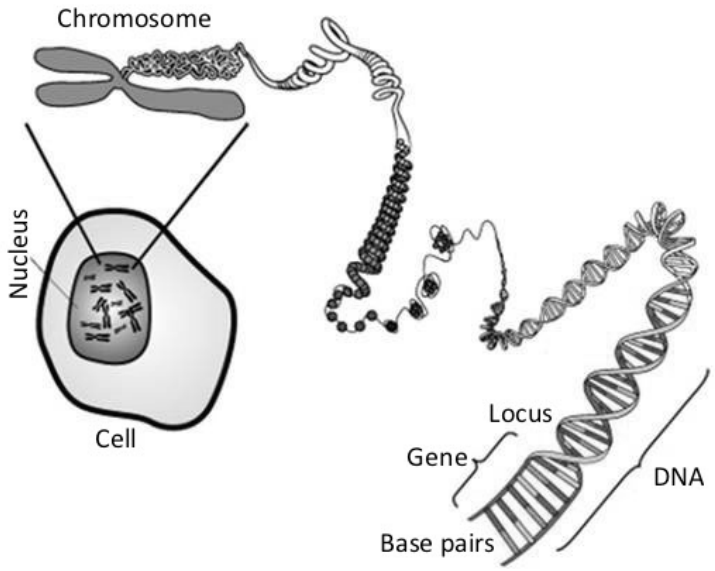


Figure 4.2: Chromosome and gene of an organism, a cell contains a genome (a few chromosomes) or a single chromosome depending on the type of organism. Genetic information is stored on DNA, and DNA comprises of a long chain of base pairs. A set of base pairs is called a gene, which controls organism's physical traits. The position of gene is known as locus (GENCODYS, 2010).

Cell reproduction

The types of cell reproduction are amitosis, mitosis and meiosis. Amitosis is for a eukaryotic cell division but without nuclear envelope breakdown, chromosomes condensed and visible spindle are formed. Opposite to amitosis, mitosis divides eukaryotic cells into two involving nuclear envelope breakdown, chromosomes condensed and visible spindle are formed. Among these cell reproduction types, meiosis is a major type for an organism with sexual reproduction, and it is important as it allows parent genetic information to be mixed for offspring (Starr, 2007). Meiosis is facilitated in two major steps (Toole and Toole, 1999; Starr, 2007) as shown in Figure 4.3: In the first step, paternal and maternal chromosomes of a diploid cell, are aligned and formed as homologous chromosomes, where a diploid cell refers a cell that contains two sets of chromosomes and each set is inherited from each parent. Note

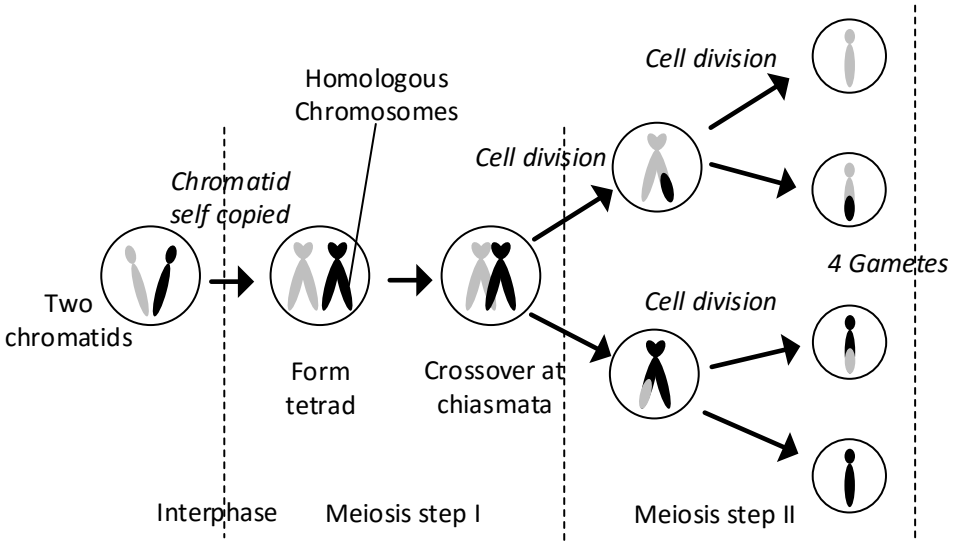


Figure 4.3: Detailed process of meiosis: A diploid cell contains paternal and maternal chromosomes, or called chromatids (single pair). In the interphase, chromatids are self copied. In the first step of meiosis, chromosomes are aligned and formed as homologous chromosomes – tetrad. The paris of chromatids are joint at random crossing points – chiasmata. In the second step of meiosis, chromatids are first divided into two cells, and further divided into four sets of chromosomes stored in gametes.

that a diploid cell has both the paternal and maternal genomes, and these genomes are self copied formed as two chromatids. As each chromatid has two pairs of homologous chromosomes, in this case, it is called tetrad. Then the pairs of chromatids are joint at a few random crossing points, known as chiasmata, and the exchanged parts between the chiasmata is called crossing over. In this way, it allows the paternal and maternal genetic information to be mixed. Subsequently, these mixed chromatids are obtained at the end of the first step. In the second step, these chromatids are first divided into two cells, and further divided into four sets of chromosomes stored in four separated cells – gametes. Since each cell only has one set of chromosomes (the mixed genome), it is called haploid.

Haploidy and diploidy cells

Although there are not only haploid and diploid (cells) creatures, these two are related to higher-level creature evolutions. For example, human beings are diploid creatures, but the gamete is haploid since meiosis exists. However, when the mating process occurs, two gametes are merged and produce a new diploid cell, where paternal and maternal genomes are inherited simultaneously. Subsequently, a new child grows from the new diploid cell, and its development process is known as ontogenesis. Compared with the diploidy creatures, in the haploid creatures, the genome is directly copied to a new cell, and the new cell is further developed in the ontogenesis process. In this way, the responses of the genome in ontogenesis are all passed to the offspring, most of EAs are based on this process (Langeheine, 2005). On the other hand, in the diploid creatures, their phenotypes are dependant on the mixed maternal and paternal genomes, and so are not fully correlated to a former genome.

Mutations

Mutation is an important mechanism to drive evolution, and involves variations in the evolutionary process. Mutation can occur at the gene level, chromosome level or the genome level. This section focuses on the first two levels as these concepts are close to EAs. The smallest mutation happens at the gene level, where partial base pairs are replaced by other base pairs in the DNA. However, this might not change the phenotypical traits. On the chromosome level, the mutation can occur on a single chromosome, or two or more chromosomes. On a single chromosome, one or more genes might be deleted, and these genes will no longer be in the genome. In addition, the order of some genes on a chromosome can be reversed (180°) and reinserted in the chromosome itself. For two chromosomes, where these two chromosomes need to be initially homologous, but later non-homologous, the mutation is usually classified as duplication and translocation. In a duplication mutation, a segment of one chromosome is inserted into the other chromosome

– this segment is omitted from the first chromosome and it is duplicated in the second chromosome. Similar to the crossing over, the genetic material changes since the duplication causes the genetic material position to shift on chromosomes, and this phenomenon is known as translocation (Toole and Toole, 1999).

Ontogenesis

Ontogenesis is a relatively complex process which involves the conversion of the DNA stored genetic information into phenotypical traits under a particular environment. In terms of a genome, it can carry a large amount of genetic information, but not all the information can be used to encode the phenotype. Put simply, a genotype to phenotype mapping can be summarised as the following two major steps: First, to guide a protein synthesis, the DNA attached genetic information will convert to a special substance called messenger RNA (RiboNucleic Acid), and this process is known as transcription. Second, the messenger RNA will instruct the amino acids to form different proteins, and these proteins are the sources to produce and control traits of an organism or creature.

4.3 Evolutionary Algorithm (EA) and its components

EAs are a set of algorithms that imitate the process of natural evolution, which uses the evolutionary process as a model to solve or optimise actual problems. A population concept is used in these algorithms, and the population contains a number of individuals – potential solutions of a targeted problem. The environment is represented by a cost function which allows an individual to be evaluated and numerically scored – the fitness. Under the environment, individuals produce their offspring, referred to as evolutionary loop, and

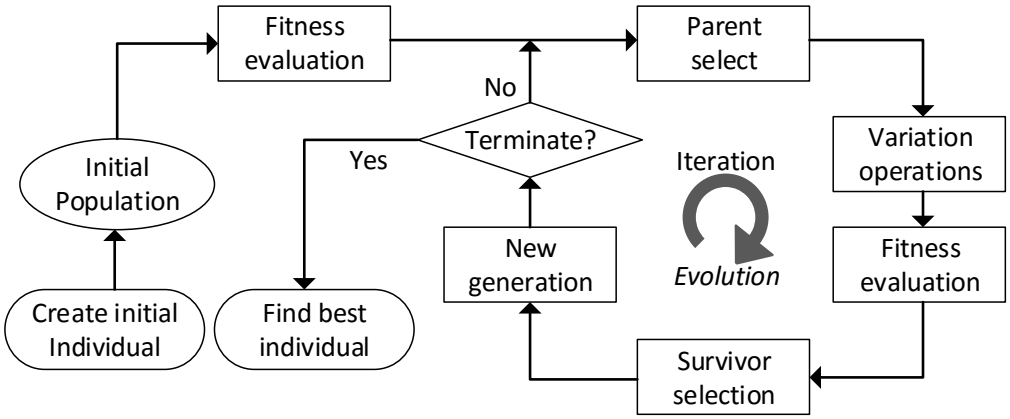


Figure 4.4: A generic flow of EAs (Langeheine, 2005)

the variations are involved by variation operators, for example a mutation operation, in each generation. In the process of evolution, the higher fitness individuals survive. In contrast, poor fitness individuals die out. Figure 4.4 shows a generic flow of EAs.

EAs start from a population of individuals, and each individual has for example at least a chromosome which is used to encode the solution of a target problem. In the initial population step, an individual has an initial random solution to the problem, and then individuals are assigned fitness by the cost function. In the evolutionary iterations, best individuals are selected to produce offspring, and variations are applied to new offspring. Subsequently, they are evaluated by the cost function again. Using a selection mechanism, highly fit individuals are selected to form a new generation and are involved in the next step of the evolutionary loop. When the termination condition is met, evolution stops and the fittest individuals are filtered out; these solutions can be viewed as the ultimate solutions of the problem.

Table 4.1: Binary code of x, and the result of y for finding the minimum value of Equation 4.1.

Binary string	x value	y value
01111110	126	4
10000000	128	0
10000010	130	4
10000011	131	9

4.3.1 Representation

In order to solve an actual problem, an EA has to represent its solution in certain ways – this is called the representation. The representation refers to a special data structure, which is used to enclose all parameters of a problem, the genotype; where these parameters are the solution of the problem. Normally, the solution is carried and defined as at least one chromosome in an individual in the EA, and genes reflect all parameters of the problem. The actual solution of the problem, for instance using an EA to find suitable component values of a circuit and finally producing the circuit, is the phenotype. The representation is usually selected based on the problem, and also considers how the variations are applied. The following five representations are commonly used by EAs, which are: binary strings, integers, real numbers, graphs or hybrids.

Taking the binary string and integer representations as examples: the binary string, also known as binary codes, is simple, but versatile. It is a popular representation used in EAs. The typical application of binary codes is to represent values, or show certain relationships. For example, finding the minimum value y for Equation 4.1, and defining x range is $0 - 255$. x can be encoded as 8-bit binary string, and Table 4.1 lists some values of x and y, when"x" is "10000000", then y is minimum. In addition, the binary string can also present for example the status of switches in a circuit, which can define the circuit function or topology as shown in Figure 4.5.

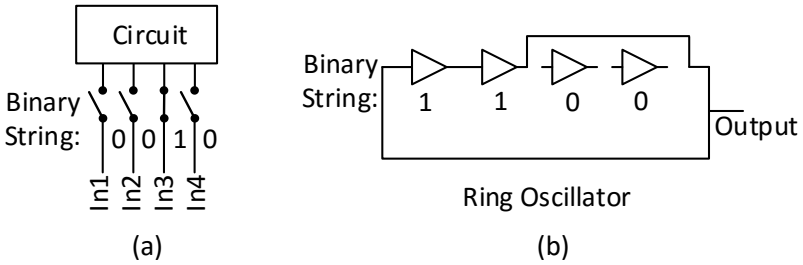


Figure 4.5: Binary string can be used to represent the status of switches in a circuit: (a), binary code can control "on" and "off" of a switch, so input of a circuit is controllable. (b), binary code can also control a set of switches, for example controlling an inverter is connected in a ring oscillator loop or not – referred to a circuit topology, then the frequency of the oscillator is adjustable.

$$y = (x - 128)^2 \tag{4.1}$$

In some cases, the binary string might not be able to effectively present a problem, for example the grouping problem. In this type of problem, the integer representation is usually used. Figure 4.6 shows the integer representation for a multiple-bin packing problem – packing 8 items into 4 bins, and the bin index is from 0-3. In this problem, each item can be specified as an integer, and each integer value can indicate to which bin the item is allocated.

4.3.2 Variation

Variation is a primary operation that drives the evolution in EAs. It applies small random changes to existing individuals, which changes a bit of the encoded parameters, to create new offspring. In general, mutation and recombination are two common variation operators, or commonly referred to as genetic operators in most EAs.

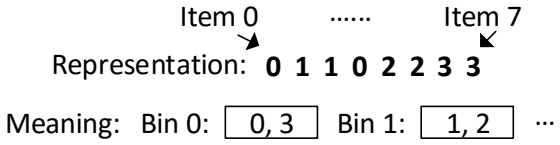


Figure 4.6: An example of the integer representation – multiple-bin packing: Each integer can be used to represent an item. The value of integer helps to explain which bin the integer matched item is packed. As shown in the figure, items "0" and "3" have a value of "0", these items is packed in bin "0".

Mutation operator

The mutation operator applies small random changes to a genotype, and it happens when only one parent producing one offspring. The main task is to produce a new offspring by directly copying the parent, and randomly manipulating some encoded parameters of the solution on the parent. As an optimisation problem is referred to the traverse of solution domain, the mutation can sightly change the searching area which allows the EA to effectively find the optimal "point" for a solution.

The mutation operation has to match the representation, for instance, the most fundamental mutation operation for binary string is called flipping a bit, which means that every bit in the string would be flipped with an equal probability. This probability is known as mutation rate. Mutation has to be different in the integer representation case. Figure 4.7 shows two integer representation mutation operations, and these are inspired by biological mutation, which are the gene locus swapping and the inversion operations. Note that the integer crossover is not limited to these two methods, and mutation can be further extended. For example, in the Grouping Genetic Algorithm (Falkenauer, 1994), mutation is performed on a higher level of abstraction, and this mutation will be introduced in the following Chapter 6.

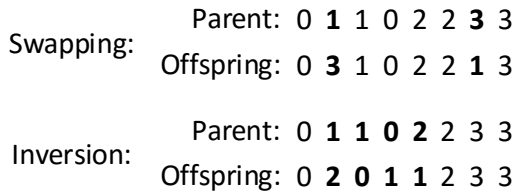


Figure 4.7: Two mutation operations for integer representation: gene swapping and gene inversion. Gene swapping: Any two genes can be swapped on a chromosome. Gene inversion: The order of a segment of genes is inverted and inserted into the original chromosome.

Recombination operator

Recombination operator recombines parent's genetic information in order to produce offspring. This operation usually produces two children in each operation. As this process is similar to the crossing over in meiosis of an organism, this operation is called crossover. The number of crossover operations that happen in a generation, like the mutation rate, is controlled by a crossover probability, or known as crossover rate. Figure 4.8 lists a few typical crossover operations, which are single-point (or 1-point), two-point, uniform and arithmetic crossover operations. To begin with, the parent's genetic information, the genomes, are represented as two vectors, $\vec{a}$ and $\vec{b}$. Note that the arithmetic crossover operation cannot be applied to binary codes.

In the single-point crossover operation, these two vectors are split at a random point and exchanged for producing offspring. Two-point crossover operation, which can be further extended to the n-point crossover operation, indicates that n random segments of vectors are swapped. Uniform crossover is different from the previous two operations. It treats each vector as an independent element and assigns a random decimal number ranging from 0 to 1 at each crossover operation. If the assigned number is greater than 0.5, the corresponding vector will be swapped, otherwise, it will be directly copied. The arithmetic crossover operation can be described by Equations 4.2-4.3,

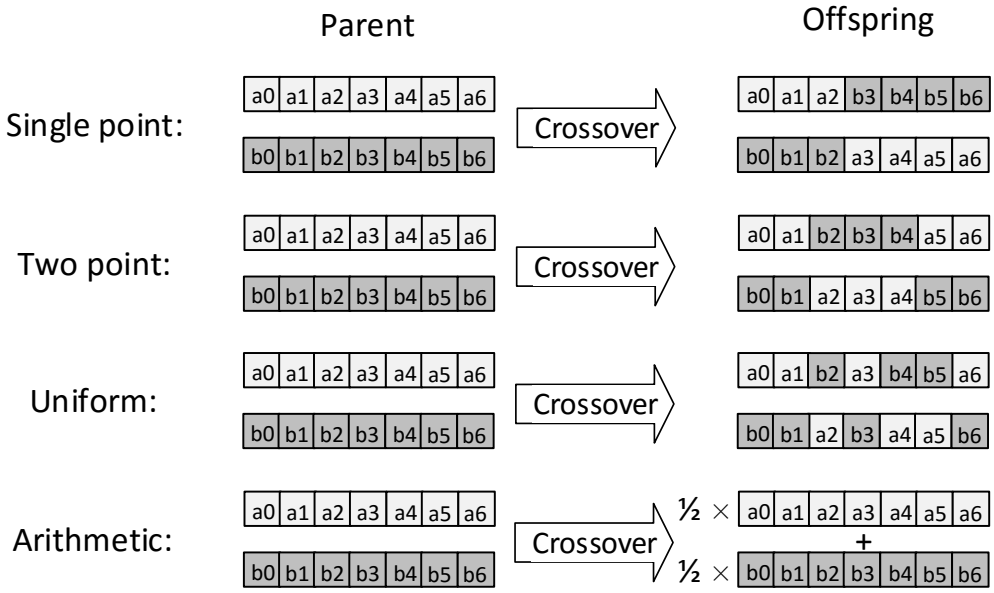


Figure 4.8: Different crossover operations (Langeheine, 2005): There are four common crossover operations, which are single point crossover – two vectors crossover at a random point, two point crossover – two vectors crossover at two random points, uniform crossover – any element of vectors can be crossovered under a probability and arithmetic crossover – two vectors are executed an arithmetic calculation and generated two new vectors.

where the vectors $\vec{c}$, $\vec{d}$ are two genomes of generated individuals.

$$\vec{c} = k\vec{a} + (1-k)\vec{b} \tag{4.2}$$

$$\vec{d} = (1-k)\vec{a} + k\vec{b} \tag{4.3}$$

Where k is a proportional coefficient. When k is equal to $\frac{1}{2}$ as shown in the Figure 4.8, the new genomes of individuals are created by inheriting 50% of each individual of the parent.

4.3.3 Evaluation

In EAs, an individual, or a solution, is generally evaluated by a cost function, but, in multiobjective EAs, they might use a few cost functions where each function evaluates one major property of the solution, and this will be introduced in the following sections. Using the function, it allows the solution to be given a numeric score, and the score indicates how the close the solution is to the optimal solution of a targeted problem. Therefore, all genetic information encoded parameters of the individual have to contribute to the score. Since the cost function or the evaluation reflects the environment in biological evolution, the function is usually called fitness function in EA related literatures. In most cases, as discussed previously, an EA can be viewed as searching all solution spaces of a problem and finding the maximum, so a higher fitness value often suggests a better solution. As shown in Equation 4.4, where f is a fitness function, and s, s' are two possible solutions, s is considered a better solution as its fitness is higher than s'.

$$f(s) > f(s') \tag{4.4}$$

Unfortunately, in real world problems, when maximising a property of a problem, other unexpected properties can also be maximised. These are usually referred to as constraints of the problem, and have to be avoided in order to accomplish a valid solution. A classic method to avoid invalid solutions in the evolution is to implement a penalty along with the original fitness function as presented in Equation 4.5.

$$fitness(s) = f(s) - penalty \tag{4.5}$$

Where s is a solution, the $fitness(s)$ is the new fitness function that incorporates a penalty, for example the penalty can be either a large fixed value or a function of s: $penalty = f_{penalty}(s)$ (Greenwood and Tyrrell, 2007).

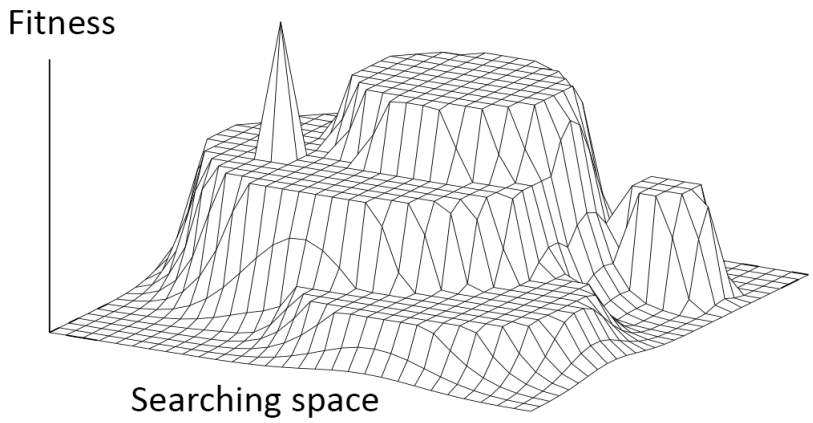


Figure 4.9: 3-D fitness landscape (Verel, 2015)

In a special condition, if s is a valid solution, the *penalty* can be considered as "0", so fitness value is only contributed by $f(s)$. From the evolutionary perspective, this penalty can essentially decrease the fitness of an invalid solution, and result in the solution not being selected for reproduction. Note that, to better present the circuit properties, experiments in this thesis are set up to minimise the fitness, where a smaller fitness value means that a better solution can be found.

It has been introduced that the EA can be treated as a tool that traverses solution spaces for a problem, and this can be further considered as an optimisation task on a 3-D fitness landscape as shown in Figure 4.9, where the hyperplane is formed by all combinations of the genotype, the search space. The landscape itself features peaks and valleys, and its altitude represents fitness values. In the evolution, variations are the mechanisms to move a population on the searching space, and the ultimate goal is to find the maximum fitness or the minimum fitness on the landscape depending in the problem. In fact, the smoothness (or ruggedness) of the landscape and searching range is strictly determined by the combination of representation and fitness function. If the landscape is smooth enough and has only one summit, a global optimal can be found easily. Otherwise, the search might

find a local optimal. Moreover, the most difficult scenario is also shown in the Figure 4.9, where the landscape contains a few flat surfaces, and a global maximum is surrounded by a flat surface (not the highest surface). This is called the needle in haystack problem. A search might not be lucky enough to get the global maximum, and focus on a surface instead – local optimal. Therefore, a robust EA usually has the suitable representation and fitness function, and the fitness function needs to be precise which provides a smooth fitness landscape and supplies enough gradients to avoid flat searching space. Even if the EA gets stuck into a worse fitness, the fitness function can guide the search to finally find the optimal solutions.

4.3.4 Selection

So far, some components of EAs have been reviewed, but, in addition to these, another important feature of the EA is the selection mechanism or the selection and replacement which is linked to the natural selection in the biological system. Eiben, Smith and Michalewicz (Eiben and Smith, 2007; Michalewicz, 2013) have indicated that an EA can be divided into two phases, which are the exploration phase and exploitation phase. Taking the fitness landscape again, in the exploration phase – at the beginning of an EA, a population can be initialised and dispersed on the searching space, so the recombination operator moves the population on a large range on the searching space, and the mutation operator assists the population to search the population adjacent space. Ultimately, the population converges on a small region of the searching space due to the selection mechanism. As the population is converged in the small region, the combination operator no longer provides large movements on the search space for the population, and the search is mainly conducted by the mutation operator, which is referred to as exploitation. Therefore, the selection can significantly alter the effect of the above two phases – the convergence speed – and further determine whether or not the EA can successfully solve a problem, where if the EA quickly converges in a suboptimal area, termed premature convergence, the

EA might not find an optimal solution.

To discuss the selection methods, the population model in EAs has to be introduced. Figure 4.4 shows the basic evolutionary process in EAs – the population produces offspring, and higher fitness offspring increases the chance to produce their offspring in the future. In artificial evolution, the population can be modelled as the generational model or the steady-state model. In the generational model, the population is renewed entirely at each generation. In contrast, the steady-state model refers to only the partial population, the worst individuals, being replaced in a generation, and how many percent of individuals in the population are replaced is called the generation gap. During selection, the probability of an individual being selected is termed selection pressure. The typical (major) selection methods are:

1) Uniform selection: an individual is selected from a population with an equal probability. The fitness of the individual is not considered, and this method is usually used in Evolution Strategy (ES).

2) Fitness proportionate selection: an individual is selected based on a fitness probability – the higher fitness individual has a higher probability of being selected.

3) Fitness ranking selection: all individuals are ranked according to their fitnesses, and assigned an index, for example, the best one is ranked as "1". Similar to the fitness proportional selection, but individuals are selected based on ranking indexes (Bäck et al., 2000). The multiobjective selection can be also classified in this category, and it will be detailed in section 4.4.2.

4) Truncation selection: μ individuals (population) produce λ offspring, and these individuals are combined $(\mu + \lambda)$ and ranked using their fitnesses. The best μ individuals are formed as a new population.

5) Tournament selection: Randomly takes $q > 1$ individuals from the population using the uniform selection, and selects a best fitness individual

from q individuals. This above selection process is repeated until all the necessary individuals have been found.

Apart from the truncation selection, which keeps a small number of the best individuals generation by generation – a mechanism known as elitism; all other selections are pure stochastic. This means that a best individual might be lost in the future evolution unless a number of elite solutions are kept.

4.3.5 Termination conditions

The termination condition indicates when the EA is stopped, and can be commonly divided into four criteria:

1) Use the convergence: If an EA executes for k generations but there is no improvements on results, the EA is terminated.

2) Use a fixed number of generations: The EA evolves the solution for a fixed number of generations, then it is stopped.

3) Results is enough: The EA might not find the best solution, but the solution is good enough.

4) EA finds known best optimal solution, the target solution.

4.4 The Genetic Algorithm

The beginning of this chapter clarifies that EAs are a suggested set of algorithms, which are described in four dialects. Genetic Algorithm (GA) is one of them, and it is a robust optimisation and searching method for solving complex problems (Holland, 1975; Goldberg and Holland, 1988). It is popular in music generation, VLSI technology, strategy planning, machine learning,

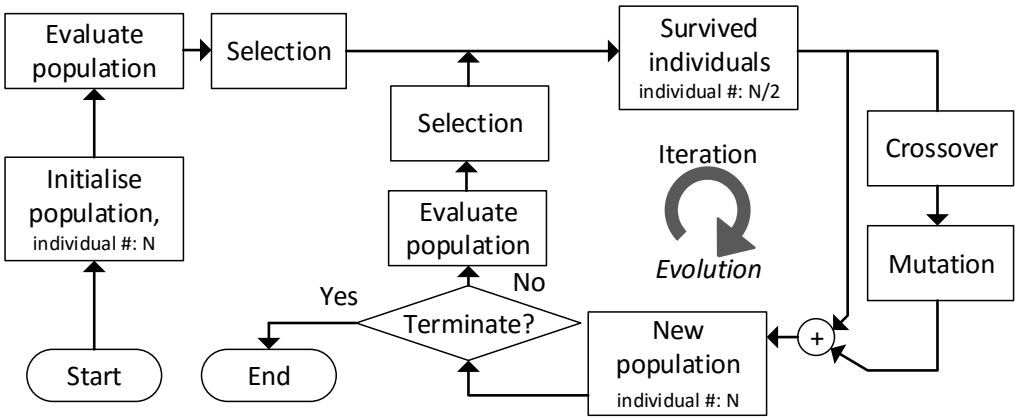


Figure 4.10: A flow of simple genetic algorithm

etc. (Srinivas and Patnaik, 1994). GA was proposed by Holland (Holland, 1975), and it was referred to as a Simple Genetic Algorithm (SGA). Subsequently, the SGA has been enhanced by adapting different representations, genetic operations (variation operators) and selection mechanisms. This is also the outstanding advantage of the GA, which means that almost any of the components and features of EAs that are previously discussed can be utilised in the GA. In this thesis, a modified GA has been used for solving the circuit clustering problem, which is the main methodology in this research.

4.4.1 Simple Genetic Algorithm (SGA)

In the SGA, a solution is encoded by a binary string, and the string is described as the chromosome of an individual. The genetic operators are the single-point crossover and "flipping a bit" mutation. The selection uses the roulette wheel selection (Zhong et al., 2005), which belongs to the fitness proportionate selection scheme. Figure 4.10 shows the SGA flow. At the beginning, a population is initialised, and the population is generated randomly – the chromosome of an individual is generated as random binary codes. Then the population is evaluated against a fitness function, and all targeting properties

of a problem can be weighted in the function. Some outstanding individuals are filtered by the selection – half of the initial population size – and these individuals produce offspring by the genetic operations. After the genetic operation, a new half of the population is generated and combined as a new population – the steady-state population model. The GA executes for many generations and stops according to a termination condition. A genetic algorithm usually has the population size from 30 to 200. The crossover rate is around 0.5 to 1.0, and mutation rate is from 0.001 to 0.05. These parameters are the control parameter of a GA, and these parameters are usually used as empirical data for adjusting a GA performance (Srinivas and Patnaik, 1994).

4.4.2 Multi-Objective Genetic Algorithm (MOGA, or MOEA)

In the real world, optimising a problem can involve multiple optimisation targets, say objectives, such as area, routability, delay and power consumption when mapping a circuit on FPGAs, and these objectives can be partially conflicted (Greiner et al., 2003). Especially, in the circuit clustering problems, these conflictions are, for example, the increase of the number of BLEs in a CLB might use more inputs and outputs of the CLB, and the reduction of interconnects between CLBs might increase as more CLBs are used. To enable the GA to work for multiple objective problem optimisation, a standard GA framework, as shown in Figure 4.10, can be extended as MultiObjective GA (MOGA). This is typically achieved by using two methods:

First, the targeting objectives can be weighted in a single fitness function as introduced before, in which the fitness can represent all features of the objectives. However, this might change the optimisation ability of the GA for a particular objective, so the weighting factor of each objective has to be carefully selected. Although the weighting proportions can be adjusted, if in a worst case there are a few conflicted objectives in a problem, the weighted fitness can cause the optimisation to get stuck in a local optimal or an invalid

solution to be produced.

The other approach to deal with multiobjective optimisation in GA is to use the concept of Pareto optimality (Pareto, 1906); this is implemented in the selection mechanism – it can be categorised as a fitness ranking selection. Pareto optimality, also called Pareto efficiency, stats an resource allocation method for a set of individuals in a society, where it is not possible to make any one individual better off without making at least one indivdiual worse off. This concept is proposed by an Italian engineer and economist, Vilfredo Pareto, based on his research in economic efficiency and income distribution. This concept is well known, and widely applied to for example economical and engineering areas. It is an efficient approach to deal with multiobjective optimisation problems. Compared with the standard GA, or the SGA, which uses only one fitness function in its selection, this method deploys a few fitness functions matching to each objective that have to be optimised in the evaluation and selection. Instead of searching one global or near global solution according to one fitness function, especially when there are a set of conflicted objectives, this method allows that the GA finds a set of tradeoffs of solutions throughout n-dimensional fitness spaces – via n fitness functions. The set of tradeoffs is referred to as Pareto set, or non-dominated front. If solution A dominates B, then the domination relationship can be presented as the following Equation 4.6.

$$\text{A dominates B} \Leftrightarrow \forall i \in \{1, 2, ..., n\}\ a_i \geqslant b_i, \text{and } \exists i \in \{1, 2, ..., n\}\ a_i > b_i \tag{4.6}$$

Where a_i and b_i are the fitness of objective i in the solutions A and B. Then the non-dominated fronts, or Pareto fronts, can be defined based on the solution domination relationships. This is normally processed by a non-dominated sort method in many MOGAs. After sorting, the first Pareto front solutions are the best tradeoff solutions. In many cases, finding Pareto fronts, particularly the first Pareto front, are usually useful for engineering.

Using this method to evaluate solutions can avoid the weighting proportions used in the single fitness function which limits the GA optimisation ability, and enable the GA to handle more conflicting objectives.

Using GA to solve multiobjective problems can be traced back to the 1990s, where Fonseca and Fleming's MOGA (Multi-Objective GA) (Fonseca and Fleming, 1993), Srinivas and Deb's NSGA (Non-dominated Sorting GA) and Horn's NPGA (Niched-Pareto GA) (Horn et al., 1994; Srinivas and Deb, 1995) have drawn much attentions in this area. These methods are built on the SGA, or more generally referred to as EA, and extend to the multiobjective EA (MOEA). This multiobjective (MO) extension can be summarised in two steps: First, the evaluation involves multiple objectives, and the individual fitness assignment is based on the non-dominated sorting. Second, the diversity of individuals on the same Pareto front needs to be maintained. Although these methods are able to solve multiobjective problems, Zitzler (Zitzler et al., 2000) have indicated that the previous methods lack the convergence property. They proposed a method that used elitism to improve the convergence property of MOEAs. In the MOEA, the non-dominated sorting is a computing density process, so it consumes more time on a computing system. NSGA-2 (Deb et al., 2002) then was introduced. This method uses a fast non-dominated sort to replace the conventional sort method, which reduce the computing complex from $O(MN^3)$ down to $O(MN^2)$ where M is the number of objectives and N is the population size, and utilises crowding distance method to select individuals. The detailed method is reviewed in Chapter 6. Compared with other MOGAs or MOEAs, this method overcomes the major drawbacks, and also reduces the computing density. It fast became a state-of-art multiobjective GA, and its multiobjective sorting and selection methods can be also used for different GAs.

4.4.3 Constraint handling in MOGAs

The real world optimisation problem not only comes with a few objectives, but also has constraints. For example, when designing a water pipe for maximising flow, the larger the diameter used, the more flow can get through. However, the diameter cannot be infinite. The cost and feasibility need to be balanced, and these are the constraints, so constraint handling is important. The constraint handling methods in MOGAs can be classified in four types:

1) Penalty function approach (Srinivas and Deb, 1995; Deb et al., 2000b): Similar to the penalty added in a single fitness function, to control constraints for MOGAs, the adding-penalty method can be used. To prevent the MO selection treating the penalty as an objective in an optimisation problem, which misleads the GA evolution, the penalty needs to add to all fitness functions. Note that, when adding the penalty, all functions have to be normalised.

2) Filtering infeasible solutions via the selection: Jiménez and Verdegay proposed a method (Jiménez and Verdegay, 1998) to control constraints in the selection which implemented a binary tournament selection for comparing solution feasibilities. In this process, two solutions are compared, where if one is infeasible, the solution is abandoned. If both solutions are feasible, Horn's NPGA (Horn et al., 1994) is used to find the best one. On the other hand, if both solutions are infeasible, it will keep the one that is closer to the constraint boundary.

3) Elaborating constraint handling: This method was proposed by Ray (Ray et al., 2001). It suggests firstly performing a non-dominated sort by using the fitness only, and performs another non-dominated sort by purely using the constraint violations. Ultimately, using non-dominated sort sorts both the fitness function and the constraint violations. By combining the above sorts, the infeasible solutions are filtered.

4) Constraint-domination principle method (Deb et al., 2000a): It differ-

entiates infeasible solutions from feasible solutions when performing the following non-dominated sort procedures:

It defines the principle as: "a solution i constrained dominates a solution j" if any of the following statement is true:

1. i is feasible, j is not.
2. Both i and j are feasible. However, i has smaller constraint violations.
3. Both i and j are feasible. Moreover, i dominates j.

4.5 Advantages and disadvantages

The use of EC, otherwise known as EAs, to solve problems has a number of advantages. Firstly, the EA is not like most of the heuristic algorithms which need to have specific knowledge on the targeted problem. This indicates that EA is a model-free heuristic algorithm, and can be used as a general problem solver. Compared with other model-free heuristic algorithms, such as random search, local search, tabu search and simulated annealing, where most of them do not respect the fitness landscape, or even the simulated annealing uses one individual which can be viewed as a simplified EA (Eiben and Smith, 2007), EAs are able to produce better solutions. In addition, there is a "no free lunch" theorem, and it proves that EA performs no worse in terms of conventional heuristic algorithms (Wolpert and Macready, 1997). Secondly, the EA can be extended for supporting multiobjective problems, and also the EA use the population model. It is not only useful for real world problems, but also supplying more possible solutions. Thirdly, due to the population that exists, it enables that the EA to be easily parallelised for optimising its computational intensities (Koza, 1999). Apart from the population, the EA components, such as representation, fitness, variation and selection are independent. These components can be reused, and easily tested.

However, the EAs are not problem free – the disadvantages can be summarised as the following: First, the EA can easily work on a problem, but it is difficult to guarantee that the EA can find the optimal solution for the problem. This is even more difficult for NP-hard problems. Second, since the EA uses a population, and other complex computations, such as fitness assignment, non-dominated sort, selection and the evolutionary iteration (generation based), results in the EA taking a longer time to solve a problem compared to conventional approaches. Third, an outstanding EA relies on the well designed representation, fitness function, variation operation, selection, or calibrated parameters. These designs are usually difficult, and have to involve a number experiments or extra self-adaptation mechanisms, or a strong understanding of an experienced designer.

4.6 Summary

This chapter introduces the concept of Evolutionary Computing (EC), and highlights that Evolutionary Computing is a set of Evolutionary Algorithms (EAs). The basic concepts and terms are described from the biological perspective. According to these biological terms, the components of Evolutionary Algorithms are introduced. The chapter focuses on the Genetic Algorithm (GA), which is one dialect of the Evolutionary Algorithms. The advantage and disadvantages of Evolutionary Algorithms are discussed. Genetic Algorithm is an automatic problem solver, and it is suitable for solving or optimising complex engineering problems. The proposed FPGA circuit clustering methods in this thesis are based on Genetic Algorithms, and use Genetic Algorithms as a main methodology. In the first method, stochastic variations are adapted to classic circuit clustering approaches. Subsequently, three Genetic Algorithm based clustering methods are proposed. To effectively evaluate solutions, the proposed Genetic Algorithm based methods are incorporated Pareto optimality for multiobjective schemes. These proposed methods will be introduced in the following chapters.

Chapter 5

RVPack: Bottom-Up Circuit Clustering Approach Using A Stochastic Perspective Greedy Algorithm

5.1 Introduction

This chapter introduces Random VPack (RVPack) – an extension of the VPack algorithm. VPack (Betz and Rose, 1997a) is a notable FPGA circuit clustering algorithm, which is implemented using the greedy algorithm (Cormen et al., 2009). The main advantages of this algorithm are that it is simple, effective and flexible. Unfortunately, there are also some disadvantages. These include inefficient cost-function design, non-optimal BLE selection, and the non-global building strategy. To improve the VPack algorithm, randomness can be injected reducing the effects of these disadvantages. Instead of obtaining a deterministic output, the random variations enable VPack to produce a number of stochastic solutions, and better solutions might be identifiable.

This chapter is organised as follows: Section 5.2 introduces the VPack algorithm in details and clarifies disadvantages, and problems of VPack in Section 5.3. An extended method, RVPack is proposed, and its implementation is described in Section 5.4. Experimental configurations and results are presented in Section 5.5 and Section 5.6 respectively, with discussion in Section 5.7. Then the summary of this chapter is in Section 5.8

5.2 The VPack algorithm in detail

The VPack algorithm clusters a large synthesised circuit into sub circuits, and each circuit is fitted on an island-style FPGA CLB. The key to VPack is the greedy algorithm (Cormen et al., 2009). The synthesised circuit contains hundreds or thousands of BLEs that are paired from LUTs and FFs via the pattern match (Betz and Rose, 1997b). Within a CLB, BLEs cluster in a sequential manner; the clustering process is stopped when the CLB constraints are met, such as, when there is no more space for BLE allocation, no more input available, etc. This is what the term "greedy" refers to. Otherwise the process continuous.

To construct a CLB, a seed BLE is needed. The seed has to have the largest number of inputs, this allows the seed to have an increased probability of forming more connections inside the CLB. Clustering BLEs in this way, a single CLB tends to have fewer connections, resulting in less interconnects between CLBs after clustering, which can also be understood as CLB exposed nets. Using the seed BLE as a reference, the next BLE is determined by a cost function. Absorbing the BLE, the clustering algorithm treats both the seed and the new BLE as an entirety to process more BLEs. The priority of BLE selection is quantifiable, and described as "gain", which is the value of the cost function. In VPack, the gain reflects the common connections between the constructed CLB and unclustered BLEs. Since the circuit is large, and the clustering using a bottom-up perspective, this may well be no common connection (zero gain) between a constructed CLB and unclustered BLEs.

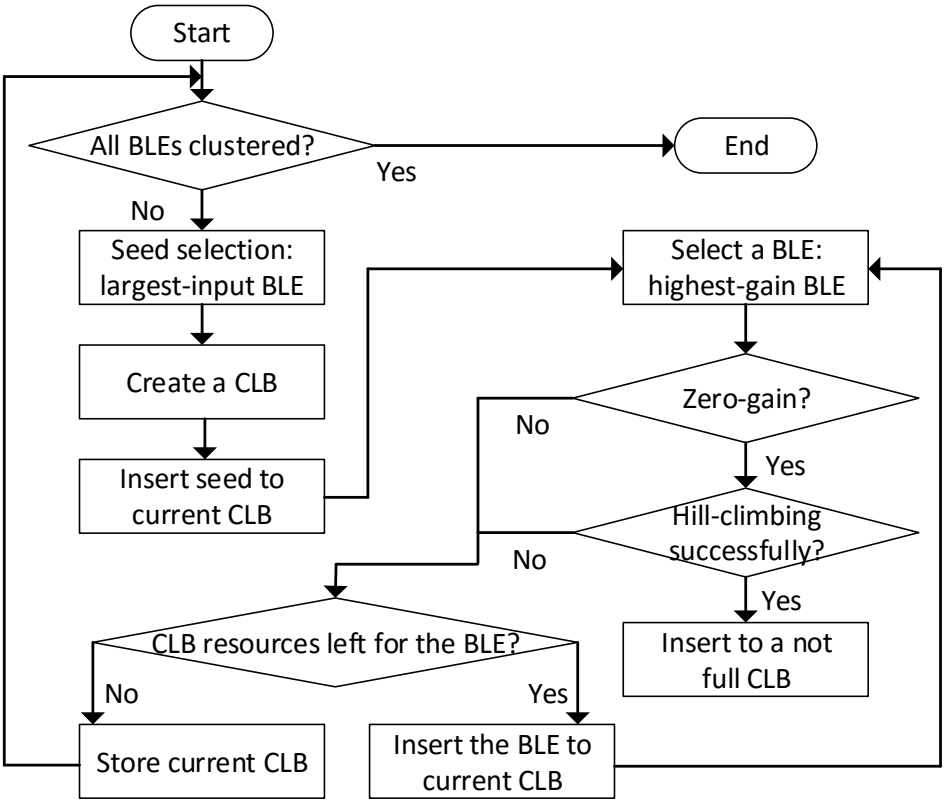


Figure 5.1: VPack circuit clustering flow

To deal with the zero gain BLEs, VPack adopts a hill-climbing algorithm to attempt to insert suitable BLEs into space-remaining CLBs. "Space-remaining CLB" means that the CLB has remaining hardware resources for allocating certain BLEs. The clustering flow of VPack is shown in Figure 5.1 (Marquardt, 1999; Betz and Rose, 1997a; Betz, 1998; Betz et al., 1999), and the algorithm pseudocode is provided in Appendices in Algorithm A.1. As introduced before, the VPack clustering process can be classified into three blocks – these are: the seed selection, BLE absorption and the hill-climbing.

To analyse the VPack algorithm in detail, figures are used to introduce the entire clustering process. Figure 5.2 shows a synthesised combinational-logic circuit schematic, represented as BLEs. This example circuit contains six 4-input BLEs, marked from *BLE-1* to *BLE-6*. The entire circuit has eight

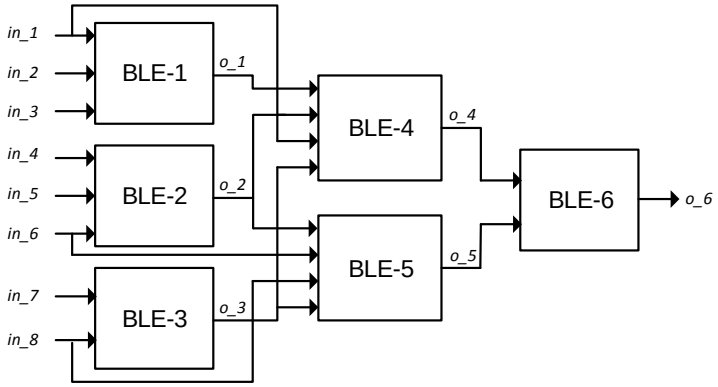


Figure 5.2: A synthesised combinational-logic circuit containing six BLEs

inputs and one output. To simplify the schematic, unused pins of the BLEs have been removed.

Seed selection

Assuming that the clustering conditions are as follows: each CLB can accommodate two BLEs, and CLB input constraints are not considered. According to VPack, the seed BLE has to be determined preferentially. In the above figure, where the number of inputs of each BLE has been listed in Table 5.1, *BLE-4* is the first BLE to have the maximum number of inputs, so *BLE-4* is selected as the seed.

Table 5.1: The number of inputs for each BLE

BLE index	# of inputs
BLE-4	**4**
BLE-5	**4**
BLE-1	3
BLE-2	3
BLE-3	2
BLE-6	2

Table 5.2: "Gain" values obtained via the cost function

BLEs index	Gain
BLE-1	**2**
BLE-2	1
BLE-3	1
BLE-5	**2**
BLE-6	1

BLE absorption

Subsequently, using *BLE-4* as a reference, the gains of the rest of the BLEs are calculated by a cost function. Equation 5.1 shows the cost function of VPack, where B represents unclustered BLEs, and C is the CLB that is under construction. This function indicates the number of inputs and outputs that B and C have in common, which is the gain.

$$Attraction(B) = |Nets(B) \cap Nets(C)| \tag{5.1}$$

After applying the cost function, corresponding gain values of BLEs are calculated, and these values are listed in Table 5.2. As the VPack algorithm is designed to pick the highest gain BLE in a CLB, in this case *BLE-1* is selected as it is the first BLE to obtain the highest gain. When the *BLE-1* is absorbed, the constructed CLB is shown as Figure 5.3. Since there is no space for more BLEs, the clustering process of this CLB is finished, and the CLB is stored. However, the entire clustering process is not stopped until all BLEs are clustered into CLBs. Under this clustering scheme, connections are formed inside CLBs, and fewer interconnects are produced between CLBs.

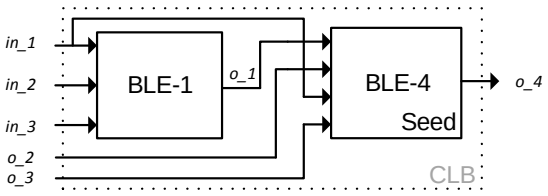


Figure 5.3: A CLB contains two BLEs with one connection inside the CLB, where *BLE-4* is the first BLE to have the maximum number of inputs (the seed), and *BLE-1* is the first BLE to obtain the highest gain. *o_1* is the included connection in the CLB.

Hill-climbing

In the gain calculation, zero gain BLEs can be obtained. Taking Figure 5.1 as an example again, there are BLEs that have no connection between them, such as *BLE-1* to *BLE-6*, and *BLE-2* to *BLE-1*. These BLEs are the main source of the zero gain. On the other hand, zero gain is also a common phenomenon when the clustering process is close to the end. This is because most of the common-connection BLEs have already been included in CLBs. In addition, since clustering constraints are also present in practice, the CLB usage is low where clustered CLBs still have resources to allocate to certain BLEs. To improve the CLB usage, a hill-climbing algorithm has been implemented in VPack. Once the VPack detects the top ranked BLE has zero gain, the hill-climbing is triggered.

Figure 5.4 interprets how the hill-climbing works. Compared with the previous clustering conditions, a CLB input constraint is applied, which limits the input number from eight to six. This constraint results in certain BLEs being unable to fit in a CLB, and leaves a few CLBs that are not full (low CLB usage). This phenomenon usually occurs when the CLB has more BLEs ($>$ 2 BLEs per CLB). Note that the figure is only used as an example to illustrate the hill-climbing algorithm.

In Figure 5.4, there are *N-1* clustered CLBs which are on the left. *CLB-N*

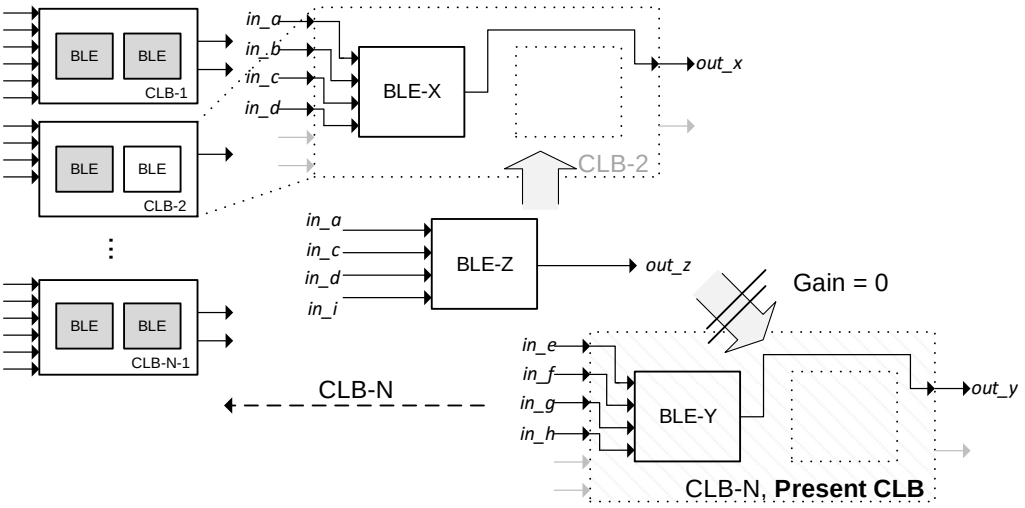


Figure 5.4: An example illustrates how the hill-climbing works. There are *N-1* clustered CLBs are on the left, the N^{th} CLB, *CLB-N* – present CLB, is on the right. Since the CLB input constraint, zero-gain *BLE-Z* cannot be fitted in *CLB-N*, but can be clustered in *CLB-2* as they have common connections.

is the CLB that is under construction. After assigning gains, assuming that the top ranked BLE is *BLE-Z* and it has zero gain to *CLB-N*. As there are many clustered CLBs, the hill-climbing is triggered and manages to traverse all clustered CLBs whether or not they have spaces for allocating the zero gain *BLE-Z*. In this traversal, the not-full *CLB-2* is searched, and then the hill-climbing will attempt to add *BLE-Z* to *CLB-2*. There are two possible outcomes: either the *BLE-Z* can be added, or the *BLE-Z* cannot be added. If the *BLE-Z* cannot be added, the algorithm will keep trying the next available CLB until all CLBs have been checked. If the BLE is still not able to be added, it will be added to the present CLB or a new CLB based on CLB hardware constraints. Although the success rate is low in practice, the hill-climbing algorithm can still improve the usage of a CLB, and reduce the number of CLB interconnects.

5.3 Disadvantages of the VPack algorithm

In this section, the problems of each block, which are the seed selection, BLE absorption and the hill-climbing, are identified.

Imprecise seed selection

Seed BLE is the infrastructure of a CLB. In VPack, seed BLE is defined as the BLE with the largest number of inputs. However, there are a large number of BLEs that have this feature, which increases the difficulty of determining the suitable seed. To examine the seed selection, the synthesised and pattern-matched MCNC-20 benchmark suite is used as a reference. In this examination, these BLEs are synthesised as 4-input LUTs, and clustered to 8-BLE-18-input input-bandwidth-constraint CLBs. For sequential circuits, a clock input is used on the CLB for connecting the reconfigurable FF of BLE. The FPGA and CLB structures refer to Chapter 2.

$$SuitableSeed\ \#\ perCLB = \frac{\sum_{k=1}^{N} SuitableSeed\ \#\ of\ CLB(k)}{N} \qquad k = 1, 2, 3, ...N \quad (5.2)$$

Equation 5.2 then is defined to investigate the average suitable seed number found in each CLB clustering; where k is the index of clustered CLBs, $SuitableSeed\ \#\ of\ CLB$ is the suitable (possible) seed number for a CLB, and $SuitableSeed\ \#\ perCLB$ is the average seed number per CLB. Based on the equation, the proportional relation of average seed number per CLB to total BLE number is presented in Figure 5.5. The red-coloured part in each bar in the figure indicates that there are a number of BLEs can be selected as a seed for a CLB. For example, in the benchmark "clma", average 2,535 BLEs can be a seed, but in practice, only one seed, or saying the best seed

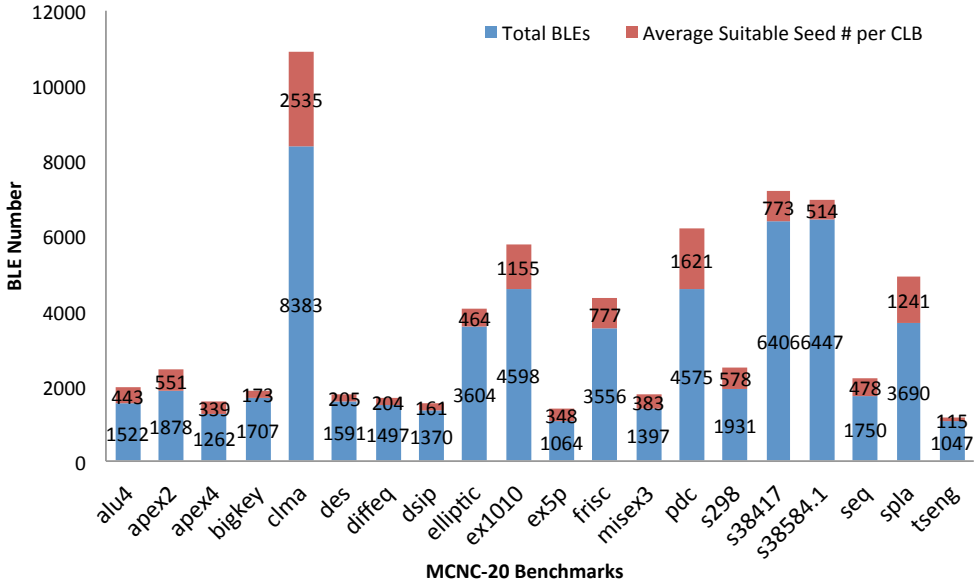


Figure 5.5: Average number of possible seed per CLB in the MCNC-20 benchmarks, each bar shows the number of BLEs in each benchmark, and red-coloured part in each bar indicates the average possible seed number for each CLB clustering.

BLE, is expected for each CLB clustering. This means that the seed selection is imprecise in VPack.

Coarse BLE absorption

$$HighestGainCLB\ \#\ perPacking = \frac{\sum_{k=1}^{N} HighestGainCLB\ \#\ of\ PackingStep(k)}{N}$$
$$k = 1, 2, 3, ...N \quad (5.3)$$

The BLE absorption, adding a BLE to a CLB, also faces the same problem as the seed selection. Table 5.2 is an example to help explain this. The cost

Table 5.3: Statistics for average highest gain BLE number of appearances per absorption in the VPack

Benchmark	BLEs	Benchmark	BLEs
alu4	10	ex5p	6
apex2	7	frisc	19
apex4	8	misex3	8
bigkey	27	pdc	12
clma	39	s298	27
des	14	s38417	20
diffeq	13	s38584.1	29
dsip	23	seq	9
elliptic	30	spla	9
ex1010	11	tseng	21

function, which is shown in Equation 5.1, is called at each BLE absorption and the gains of the unclustered BLEs are refreshed. The table shows that more than one BLEs can obtain the same gain. This means that the highest gain BLE is not unique. To monitor how many the highest gain BLEs appear in each BLE absorption, Equation 5.3 is defined and implemented in the VPack algorithm to count the average highest gain BLE number for each absorption, the recorded data is collected in Table 5.3. In the equation, k indicates each absorption, or considered as each time the cost function is called. $HighestGainCLB\ \#\ of\ PackingStep(k)$ is total number of BLEs that have the maximum gain at the k^{th} step absorption. $HighestGainCLB\ \#\ perPacking$ is the average number of highest gain BLEs per absorption throughout the clustering process. Although the highest gain BLE number is dependent on the circuit, the large BLE number still indicates that the BLE selections are coarse.

Inefficient hill-climbing

The VPack algorithm is a bottom-up clustering algorithm, where the construction of the present CLB is not linked to already clustered CLBs, hence

zero gain BLEs are obtained frequently. However, this does not mean that these zero gain BLEs are useless. The hill-climbing algorithm is an extension to deal with these BLEs. The hill-climbing picks a zero gain BLE, and adds the BLE to a space-remaining CLB. The nature of a hill-climbing algorithm means it always performs a "first-come-and-first-serve" rule. When the first available CLB is found, the BLE is added; despite the fact that there might be a better CLB to accommodate the BLE, the hill-climbing is stopped. Because of this feature, the clustered circuit often gets stuck in a particular solution pattern – a local optimum, and reduces the solution optimisation level.

5.4 The Random VPack (RVPack)

5.4.1 Motivation

In the previous section, the VPack algorithm and its problems have been reviewed. This section introduces the extended VPack algorithm, known as Random VPack (RVPack). The RVPack algorithm is designed to keep the originality of VPack whilst improving the clustered circuit (solution) quality. Injecting random variations in the VPack clustering process is a fundamental intention in the RVPack. Using the same clustering scheme, these variations can drive VPack to produce different solutions other than a deterministic clustering solution. Compared with VPack, RVPack offers greater possibilities to create better solutions, and the best circuit clustering solution is identifiable.

5.4.2 Implementation

The RVPack algorithm uses a stochastic perspective to rebuild the VPack algorithm. This section clarifies how the random variations are injected into the VPack, and shows how the randomness affects the clustered circuits

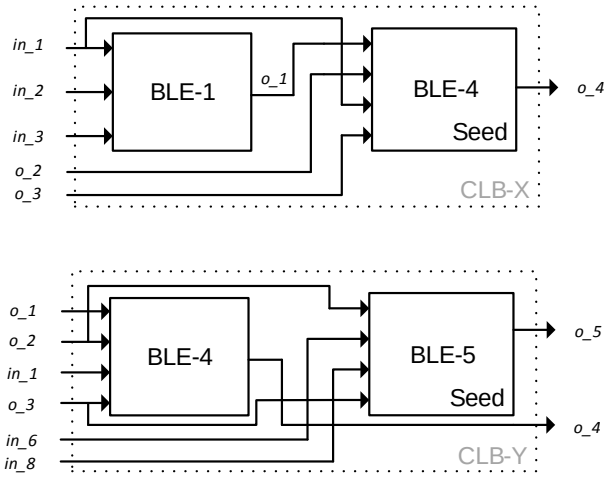


Figure 5.6: *CLB-X* is the first CLB clustered by VPack, and it is a unique solution. *CLB-Y* is the first CLB clustered by RVPack, and it is a possible solution. This means that RVPack can produce different clustering solutions.

(solutions) in different clustering stages.

Randomness in seed selection

Assuming that RVPack still clusters BLEs into 2-BLE input-constraint-free CLBs, and starting from the seed selection. Using Figure 5.2 as an example circuit again, *BLE-4* and *BLE-5* have the largest input among all unclustered BLEs, and both BLEs can be the seed. Rather than using *BLE-4*, RVPack deploys a pre-selection to all suitable seeds, which offers equal opportunities for them to become the seed. To facilitate this selection, all largest-input BLEs are pre-selected by a time-based-pseudo-random-number generator under a uniform probability. When a new seed is needed, RVPack counts all the largest input BLEs and marks these BLEs with new indexes that start from zero. The random number generator then provides a random integer ranged from zero to the total number of largest-input BLEs minus one. The index matched BLE is used as a seed. If there is only one largest-input BLE, RVPack uses the unique BLE as a seed.

Randomness in BLE absorption

Another change in the RVPack algorithm is the absorption of the highest gain BLEs. After picking the seed for a CLB, the gain of each unclustered BLE is calculated. Similar to the seed selection, there are likely to be a large number of BLEs that have the highest gain. To create various solutions, randomness is applied to these highest gain BLEs. In the same way as the seed selection, the highest gain BLEs are counted and marked. An additional random number generator is used to choose which BLE is the next BLE to be added to a CLB under construction. By using both the random seed and random BLE absorption methods, the clustered circuit can be significantly changed. Figure 5.6 shows two CLBs from VPack and RVPack. *CLB-X* is the first CLB that is created by VPack. *CLB-Y* is a feasible solution from RVPack. In the RVPack case, the *BLE-4* and *BLE-5* might be grouped, but this combination cannot happen in the VPack case since the *BLE-1* and *BLE-4* are already in the same CLB.

Randomness in hill-climbing

The RVPack algorithm also considers hill-climbing in the clustering process. In contrast to VPack, it uses the "first-come-and-first-serve" rule to allocate a zero gain BLE. The RVPack algorithm utilises a random insertion. Once the hill-climbing conditions are met, the random insertion will traverse all space-remaining CLBs and record them. These recorded CLBs are then randomly selected to receive the zero gain BLE. Similar to the previous two modifications, this random selection is under a uniform probability.

Figure 5.7 illustrates how this random hill-climbing affects the results. Assuming that *CLB-2* and *CLB-5* are two available CLBs, where the zero gain BLE can be inserted. The random insertion is able to produce different clustering solutions. For example, in solutions A and B, these clustered circuits present different circuit features.

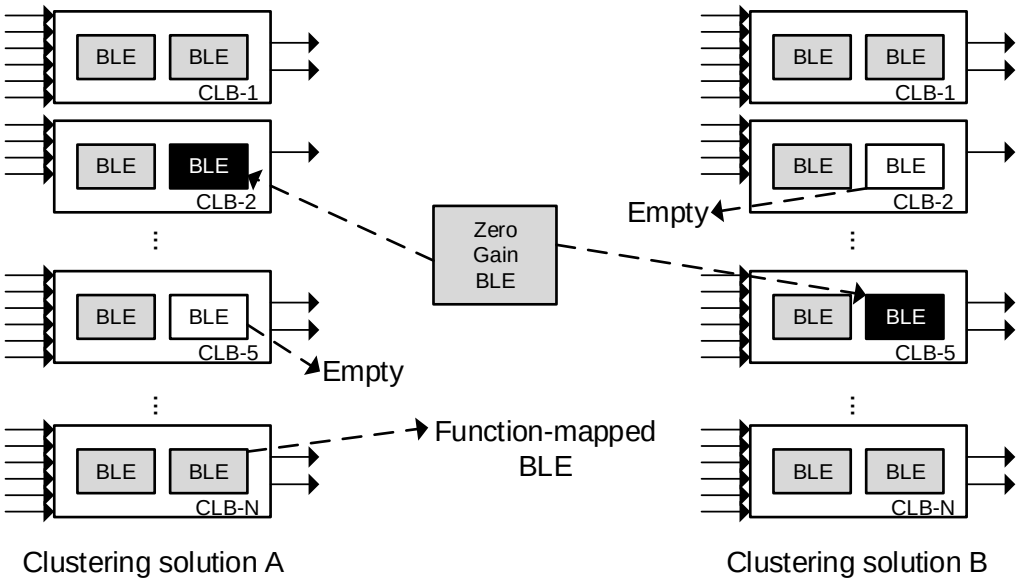


Figure 5.7: Different clustering solutions are obtained when the zero gain BLE is inserted into different CLBs, for example *CLB-2* or *CLB-5*

5.5 Experimental setups

Experiments are set up to investigate the stochastic properties of RVPack. To examine the stochastic outputs of RVPack, one hundred RVPack programs are executed. The outputs are accumulated for analysis. In the experiments, two major aspects are reviewed: First, what do the RVPack results look like compared with the original VPack? Second, are there any improvements that the RVPack can achieve?

Figure 5.8 illustrates the execution and testing flows for RVPack. Each RVPack reads one synthesised MCNC-20 benchmark (Yang, 1991) netlist and performs the circuit clustering. RVPack produces the clustered circuit as a new netlist, and the netlist is used in VPR (Luu et al., 2011; Betz and Rose, 1997b; Betz et al., 1999). VPR simulates the real mapping conditions, and supplies a FPGA mapping report, which includes the FPGA area usage, channel width and critical path delay. Note that same VPR tests are also

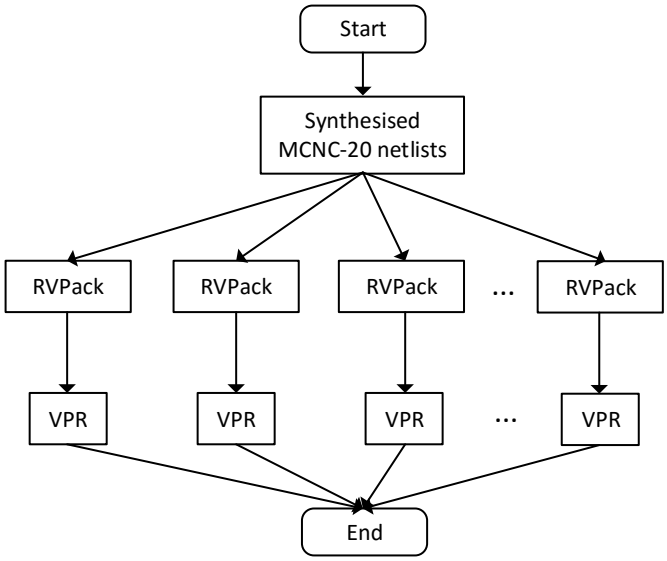


Figure 5.8: RVPack executing and testing flow: Each synthesised MCNC-20 benchmark netlist is processed by RVPack. RVPack produces new netlists to VPR for further testing. The flow is the same as to VPack.

implemented to VPack.

According to Marquardt's thesis (Marquardt, 1999), a FPGA model, which refers to the cluster-based island-style homogeneous FPGA architecture and its fabrication technology details, can be set up in VPR for efficiently evaluating a clustered circuit. The details of the FPGA model are summarised in Table 5.4. In this model, FPGA area is set as resizable, which means the area of FPGA is determined by the number of clustered CLBs and the number of pads (circuit IOs) of a clustered circuit, and VPR always attempts to use a minimum area to implement the circuit. The "*clustersize*" indicates a single FPGA CLB area (physical size – based on the CLB structure and FPGA fabrication technology), and it is normally a square block. This indicates that if the CLB array physical size is determined, the FPGA maximum pad number can be calculated. Though the use of CLB number can be small in a circuit, to arrange a large number of pads of the mapped circuit, FPGA area can be enlarged. In Table 5.4, K, I, N, M_{clk} are the parameters of FPGA

Table 5.4: FPGA model details for evaluating clustered circuits in VPR (Marquardt, 1999)

Parameter	Setting
FPGA Architecture	Homogeneous
FPGA size	Resizable – $X * Y$ CLBs, based on CLB and pad number
Pads, each side of FPGA	$Pads = \left\lvert 2 \times \sqrt{clustersize} \right\rvert$
BLE, CLB – K, I, N, M_{clk}	$K, I, N, M_{clk} = 4, 18, 8, 1$, for most circuit clustering methods
Channel width W	Depending on the max. W used for the mapped circuit
Switch box type	Wilton
Switch box flexibility – F_s	3
In. connect. block flex. – $F_{c,in}$	0.25
Out. connect. block flex. – $F_{c,out}$	1
Wire segments	Segment length 4, uniform
Routing switch type	50% tris. buffers, 50% pass transistors
Technology	TSMC's $0.35\mu M$ CMOS process

BLE and CLB models, and also used to VPack and RVPack circuit clustering. The reason of using these parameters refers to Chapter 2. The channel width W is a pre-defined parameter in fabricated FPGAs, but this is also a variable in VPR, where VPR keeps using the minimum W to map a circuit. This means the value of W, the channel width, can reflect the routability of a clustered circuit.

These tests were implemented on a high performance computing cluster, where the cluster uses SGE (Sun Grid Engine) (Oracle Corp., 2010) to automatically schedule computing tasks, so all single-thread RVPack and VPR programs can run in parallel using multiple processors. The execution time was recorded, and defined as the cluster-processor-occupying time. The computing cluster has 128 identical processors, and each processor can execute one program, so that the program execution time is the cluster-processor-occupied time. Note that the execution time is dependent on the processor

and operating system types. In these tests, the cluster processors were all identical Intel Xeon processors, and the operating system is the 64-bit CentOS V6.7.

5.6 Experimental results

In this section, the RVPack results are reviewed with discussions in Section 5.7. The results are shown in two levels: First, the direct outputs are taken from RVPack, which includes the number of CLBs, the number of CLB interconnects and execution time. Second, the reported information is obtained from VPR. This includes the FPGA area usage, routing channel width, routing wire-length and the critical path delay. Each comparison focuses on a single feature of the solutions, and the comparisons use the VPack as a baseline (More method comparisons are presented in following chapters). Detailed results and variation box plots have been included in Appendices.

5.6.1 RVPack direct outputs

CLB usage

Figure 5.9 shows the CLB number comparisons. For the MCNC-20 benchmarks, the sums of the clustered CLBs in the RVPack worst case, best case and VPack are 7,581, 7,534 and 7,551 respectively. These RVPack results indicate that improvements can be obtained in the RVPack best case, which is 0.23%, but there is also 0.40% deteriorations when compared to the results in the RVPack worst case.

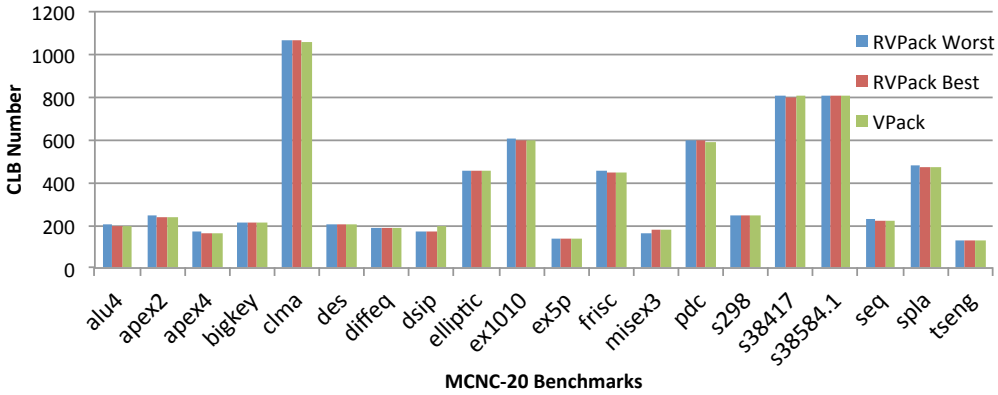


Figure 5.9: RVPack CLB number for MCNC-20 benchmarks compared to VPack, lower is better. Data boxplot and detailed data are provided in Appendices in Figure A.5 and Table A.3.

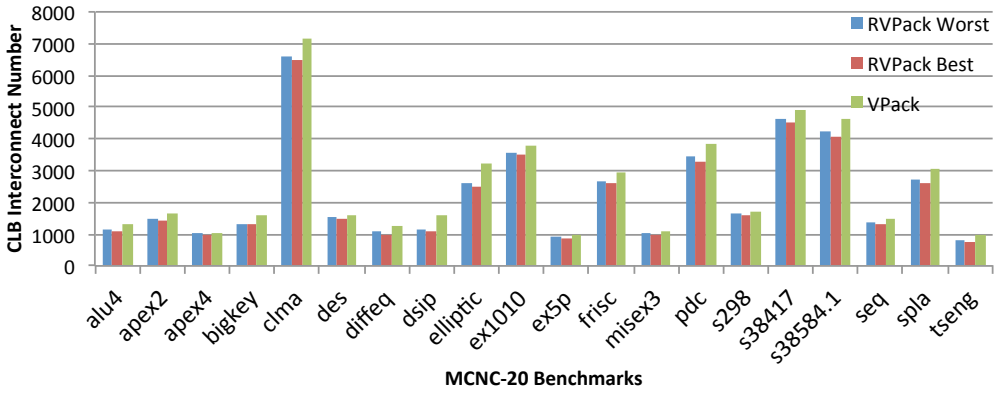


Figure 5.10: RVPack CLB interconnect number for MCNC-20 benchmarks compared to VPack, lower is better. Data boxplot and detailed data are provided in Appendices in Figure A.6 and Table A.4.

CLB interconnect

Figure 5.10 shows the CLB interconnect number comparisons for the MCNC-20 benchmarks. The total CLB interconnects in the RVPack worst case, best case and the VPack are 44,980, 43,349 and 49,840 respectively. Compared with VPack, RVPack worst case improvement is 9.75%, and the RVPack best case improvement is 13.02%.

Table 5.5: RVPack execution time comparisons, single execution. Data boxplot and detailed data are provided in Appendices in Figure A.7 and Table A.5.

Benchmk.	Long.	Short.	VPack	Benchmk.	Long.	Short.	VPack
alu4	4	2	2	ex5p	2	1	2
apex2	6	4	5	frisc	17	13	15
apex4	3	2	2	misex3	3	2	3
bigkey	5	3	4	pdc	34	22	29
clma	109	73	98	s298	8	4	5
des	4	3	4	s38417	59	42	49
diffeq	3	2	3	s38584.1	55	39	50
dsip	3	2	3	seq	5	3	5
elliptic	19	12	16	spla	20	16	19
ex1010	34	24	30	tseng	2	1	1

Unit: Second
Benchmk. = Benchmark
Long. = RVPack longest execution time
Short. = RVPack shortest execution time
VPack = VPack execution time
Shorter time is better.

Execution time

Since circuit processing time of each CLB is variable, which is dependent on different BLE combinations as well as cost function calculation time is based on unclustered BLEs, when randomness is involved, the entire-program-execution time is affected. Table 5.5 shows the differences of RVPack execution time. Note that VPack execution time is fixed and it is a subset of RVPack. This is due to the VPack clustered solution being deterministic.

Table 5.6: Worst case RVPack on FPGA area usage, $X \times Y$ arrays, for MCNC-20 benchmarks compared to VPack, lower is better. Data boxplot and detailed data are provided in Appendices in Figure A.8 and Table A.6.

Benchmark	RVPack*	VPack	Benchmark	RVPack*	VPack
alu4	15*15	15*15	ex5p	12*12	12*12
apex2	16*16	16*16	frisc	22*22	22*22
apex4	13*13	13*13	misex3	14*14	14*14
bigkey	36*36	36*36	pdc	25*25	25*25
clma	33*33	33*33	s298	16*16	16*16
des	42*42	42*42	s38417	29*29	29*29
diffeq	14*14	14*14	s38584.1	29*29	29*29
dsip	36*36	36*36	seq	16*16	15*15
elliptic	22*22	22*22	spla	22*22	22*22
ex1010	25*25	25*25	tseng	15*15	15*15

RVPack* = RVPack worst case

5.6.2 RVPack VPR results

FPGA area usage

After comparing the basic information from the RVPack experiments, VPR is used to map these clustered MCNC-20 benchmark circuits onto the targeted FPGA. Table 5.6 shows the RVPack worst case and the VPack FPGA area usages for the MCNC-20 benchmarks. In general, the RVPack (no matter whether worst or best cases) uses the same areas as the VPack. Only for the "seq" benchmark, does RVPack use more area in the worst case, which is 16×16 CLBs vs. 15×15 CLBs for VPack.

Channel width

Figure 5.11 shows the routed channel width for RVPack and VPack. The results indicate that the injected randomness affects the final FPGA routing. In the RVPack best case, the channel width is significantly reduced. For

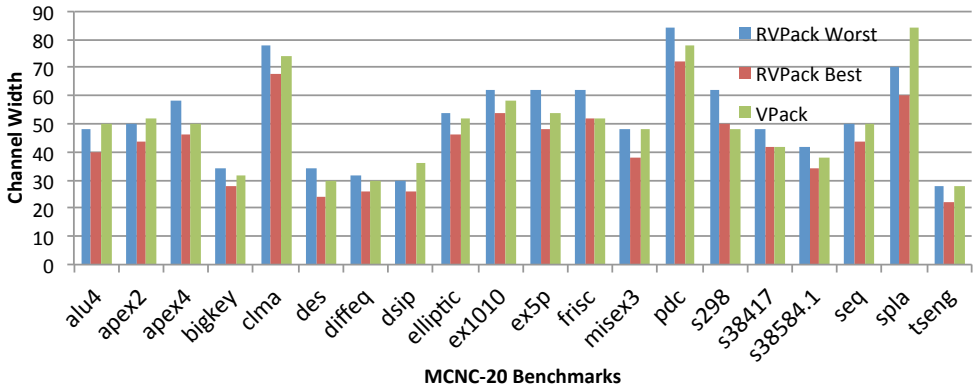


Figure 5.11: RVPack on FPGA channel width for MCNC-20 benchmarks compared to VPack, lower is better. Data boxplot and detailed data are provided in Appendices in Figure A.9 and Table A.7.

the MCNC-20 benchmarks, the sums of channel width in the RVPack worst case, best case and VPack are 1,036, 864 and 986 respectively. These figures illustrate that the well clustered circuits can reduce the channel width up to 12.37%, but, on the other hand, a worse clustered circuit can increase the channels to 4.83% compared with the VPack results.

Wire length

Wire length is another factor to evaluate the clustered circuit quality. Shorter wires mean that fewer wire lengths are used when connecting the clustered circuit on FPGA. If shorter wires are achieved, the FPGA uses less power or the mapped circuit speed is increased. Figure 5.12 shows the RVPack and VPack total wire lengths for each MCNC-20 benchmark. The sums of wire lengths in the RVPack worst case, best case and for VPack are 585,911, 518,179 and 558,315. In the RVPack best case, it shortens the wire lengths 7.19% compared with VPack, or extends the wire lengths to 4.71% in its worst case.

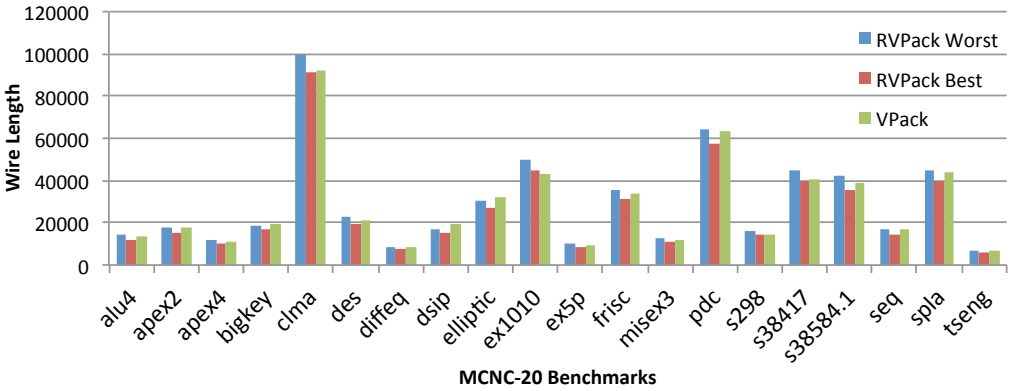


Figure 5.12: RVPack on FPGA wire length compared to VPack for MCNC-20 benchmarks, lower is better. Data boxplot and detailed data are provided in Appendices in Figure A.10 and Table A.8.

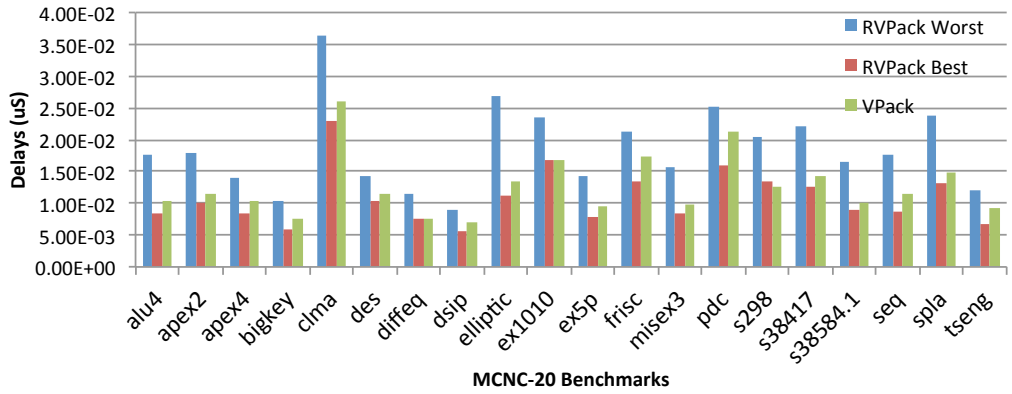


Figure 5.13: RVPack on FPGA critical path delay compared to VPack for MCNC-20 benchmarks, lower is better. Data boxplot and detailed data are provided in Appendices in Figure A.11 and Table A.9.

Delay

When the BLE combinations are changed in CLBs, mapped circuit timings are affected. Figure 5.13 compares the critical-path delay for the RVPack clustered circuits in best case, worst case and for VPack. The sums of delays of the MCNC-20 benchmarks are $3.70306 * 10^{-01}$, $2.16193 * 10^{-01}$ and $2.53159 * 10^{-01}$ μS respectively. In the RVPack best case, it improves circuit

speeds by up to 14.60% compared with VPack. However, the worst case RVPack clustered circuit is 31.64% slower than VPack.

5.7 Discussion

In the previous section, the RVPack experimental results are presented. These results indicate that the RVPack can consistently improve on the VPack results. The fundamental difference between the two methods presented in this chapter is that the RVPack injects randomness into the original VPack algorithm. This means that the RVPack results have random features. Although they can sometimes produce better results, they can also produce worse results than VPack. This is due to the RVPack only using randomness, or saying a random search, and there is no guide (or direction) introduced, so that results are stochastic.

The RVPack algorithm is an extension of VPack, however, the RVPack does not fix all the problems that are identified in VPack. RVPack is only a method to reduce the effects of the problems. Moreover, the use of randomness changes the results, but the changes are not significant, which means that the optimisation level is still low. Compared with injecting randomness to all three key processes of VPack, injecting randomness to one process would also change the result in VPack as this can change the BLE combination of a CLB.

Although the single set execution time of RVPack is similar to VPack, to produce a better result, RVPack has to execute many times; this increases the relative execution time significantly. For example, the previous experiment has a hundred executions for each benchmark. To obtain the same result as the experiment, the total execution time of one benchmark is around one hundred times that for VPack. However, since RVPack uses a random search, this indicates that a useful result might be produced at any execution, which means a better result might be found when the RVPack is executed less than

hundred times.

The RVPack, results show that it can reduce the CLB numbers of the clustered circuits, but the real mapping, VPR emulation, indicates that the area optimisation is small. The main problem is that VPR maps a clustered circuit on a FPGA under X by Y CLBs array, where X is equal to Y. Unless the CLB number is significantly reduced, the FPGA area usage changes are invisible. Another problem discovered is in the CLB interconnects. From the experiments, no matter whether the RVPack worst or best cases, RVPack always produces fewer CLB interconnects than VPack. One potential conclusion is that, since the randomness is injected to all key processes in VPack, the total effects of these processes can produce such results. If only considering the RVPack basic results, one execution result is better than VPack.

The reduction of channels in RVPack is reasonable. The 12.37% channel width reduction (best case) leads to the conclusion that the RVPack clustered circuits can save more routing tracks and increase the FPGA routability. Furthermore, RVPack also presents better results on the wire lengths and circuit delays. However, RVPack clustered circuits might have even worse performances on the wire lengths and circuit delays in the worst case, this means that RVPack produced solutions are unstable.

5.8 Summary

This chapter introduced the VPack algorithm in detail, and the problems of VPack have also been examined. Subsequently a rebuilt algorithm, RVPack, has been proposed to partially reduce the effects of these problems by injecting randomness into the VPack algorithm. The experimental results showed that RVPack was able to produce better solutions than VPack in terms of CLB number, CLB interconnects, channel widths, wire lengths and delays. It also proved that incorporating stochastic variations could improve the clustered

circuit quality for a standard greedy algorithm. GA utilises both the stochastic variation and fitness, so it is important to facilitate circuit clustering by using GAs. The GA-based circuit clustering methods will be introduced in the following chapters.

Chapter 6

GGAPack: Top-Down Circuit Clustering Approach Using MOGAs

6.1 Introduction

In Chapter 5, the VPack algorithm has been reviewed, and an enhanced circuit clustering method, the RVPack algorithm, has been described. That chapter proves through a number of experiments that the involvement of stochastic variations in a standard greedy clustering algorithm can improve solution quality. In this chapter, a new circuit clustering method, GGAPack, is introduced for FPGAs, which is based on the Grouping Genetic Algorithm (GGA) (Falkenauer, 1994) and MOGA (Fonseca and Fleming, 1993; Srinivas and Deb, 1995; Horn et al., 1994; Zitzler et al., 2000). This method not only inherits the stochastic features of RVPack, but also benefits from the GA (Holland, 1992) powerful searching ability. Compared with unguided stochastic searching, the random search, the GA utilises fitness to guide optimisation. It enables the GA to explore more effective solutions for a problem rather than blind searching. In addition, the use of MO features

in GA can also supply the best tradeoff solutions, which is better than the solutions produced by the objective weighting approach, where the weighting method is often used in new circuit clustering methods for involving more clustering metrics.

This chapter is organised as follows: Section 6.2 presents the motivation of the GGAPack algorithm. Section 6.3 introduces how GGAPack is implemented. It includes the GA chromosome representation, genetic operations, multiple objective mechanism and the fitness functions. The initial experiments and results, shown in Section 6.4, indicate that it is inefficient to use the GGAPack algorithm to cluster a circuit from scratch. Based on GGAPack, the GGAPack2 algorithm is proposed in Section 6.5.1. The experimental setups and results presented in Section 6.5. At the end of this chapter, a discussion is enclosed in Section 6.6, and a summary is in Section 7.7.

6.2 Motivation

Classic circuit clustering algorithms, such as VPack (Betz and Rose, 1997a), RPack (E.Bozorgzadeh et al., 2001; Bozorgzadeh et al., 2004), T-VPack (Marquardt et al., 1999) and iRAC (Singh and Marek-Sadowska, 2002), are based on bottom-up methods, which build a solution starting from one particular CLB. Apart from VPack which only considers circuit common connections, referred to VPack cost function in Chapter 5 in Equation 5.1, most of them involve many clustering metrics in their cost functions. Although these methods are efficient, the clustered circuits are usually considered less than optimal. On the other hand, researchers are also enthusiastic to investigate the top-down circuit clustering algorithms, and attempting to solve the clustering problem from a global perspective, for example the PPack algorithm (Feng, 2012). The core of these methods is usually composed of graph partitioning approaches (Kernighan and Lin, 1970; Fiduccia and Mattheyses, 1982; Karypis and Kumar, 1999), which clusters a circuit based on its connections. However, using connections as a major clustering objective,

reduces the clustering universality; even PPack cannot deal with CLB input constraints.

Evolutionary algorithms, and in particular Genetic Algorithm (GA), have strong searching abilities, and implement fitnesses to guide the search. Meanwhile, the GA can be extended as MOGA, hence the GA, or in particular MOGA, can be a potential approach to solve FPGA circuit clustering problems. The GGA (Falkenauer, 1994) was implemented for solving the bin-packing problem (Garey et al., 1973), but the GGA targeted problem has many similar aspects to the circuit clustering problem. For example, the bin-packing aims to pack items to bins while using fewer bins. This can be viewed as clustering the BLEs into CLBs. As well as this, the packed items have certain requirements, for instance, in a bin, the sum of the items' weight is not allowed to exceed the bin weight limitation, and the total item worth of each bin has to be maximum. These can be treated as the circuit properties in the circuit clustering problem. To deal with multiple circuit clustering metrics, based on the GGA and MOGA, the GGAPack circuit clustering algorithm is proposed.

6.3 GGAPack implementation

The GGAPack is a top-down circuit clustering algorithm. In the same way as RVPack, GGAPack reads the synthesised netlist as input and generates a new netlist for the VPR testing. The GGAPack utilises the GGA representation and genetic operations. To incorporate multiple circuit properties and clustering metrics, the GGAPack has also been expanded to support multiobjective optimisations. The multiobjective optimisation is based on the NSGA-2 (Deb et al., 2002), which evaluates and selects GA individuals via the Pareto optimality and crowding distance. The details of the GGAPack are explained in the following section.

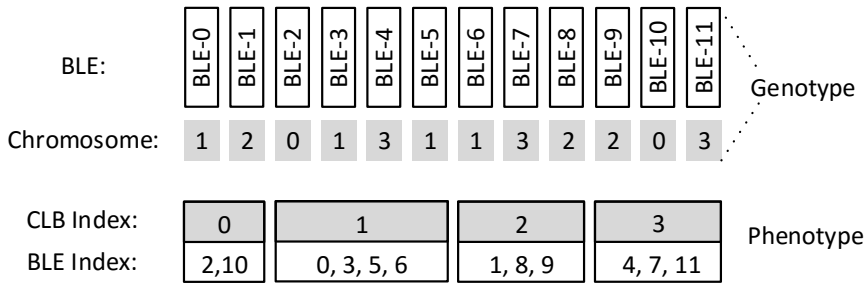


Figure 6.1: An example shows the GGAPack chromosome encoding scheme. The length of chromosome is dependent on the number of BLEs, and each gene position is used to describe the BLE index. The value of gene indicates which CLB the gene represented BLE is allocated.

6.3.1 Representation

In the GGAPack, an integer string has been selected to encode the chromosome. These integer values in the chromosome present the CLB index, and the gene's position is used to encode the BLE index. Figure 6.1 is an example to illustrate this representation.

This example chromosome shows a 12-BLE clustering problem. Below the chromosome, these boxes are considered as CLBs and these CLBs are translated from the chromosome. In terms of the chromosome, it represents four CLBs, and CLB indexes are from "0" to "3". Each integer position, or saying gene position, in the chromosome is used to encode each independent BLE, and these gene values indicate which CLBs the BLEs are allocated. Inside the chromosome, its gene number is equal to BLE number, hence the length of chromosome is variable and dependent on the BLE number. Taking CLB "1" as an example, it shows that *BLE-0*, *BLE-3*, *BLE-5* and *BLE-6* are inside the CLB. These genes, which the BLEs are in the CLB "1", have the integer value of "1". By using this encoding scheme, clustering of a circuit is able to be represented, and the number of different integer values can reflect how many CLBs the chromosome has.

6.3.2 Reproduction

Both crossover and mutation operations are implemented in the GGAPack algorithm to create new individuals. These genetic operations have two functions:

1) The crossover operation is intended to exchange BLE combinations (CLBs) between different individuals – solutions.

2) The mutation operation is designed to generate new BLE combinations for CLBs.

Crossover in GGAPack

The GGAPack crossover operation can be summarised in six steps, as follows:

1) Select two individuals randomly from the population, and copy them as new individuals.

2) Randomly determine which CLBs between the copied individuals are performing BLE combination exchanges – crossover.

3) Inject the chosen CLBs into copied individuals of each other.

4) Eliminate CLBs that contain these injected BLEs.

5) Reinsert released (freed) BLEs back to CLBs under clustering constraints.

6) Store the two copied and crossovered individuals as offspring.

To demonstrate the GGAPack crossover operation, Figure 6.2 (a) presents an example of two randomly selected and copied individuals from the GGAPack population. These copied individuals are the individual X and individual Y, and represent two clustering solutions. In these two individuals,

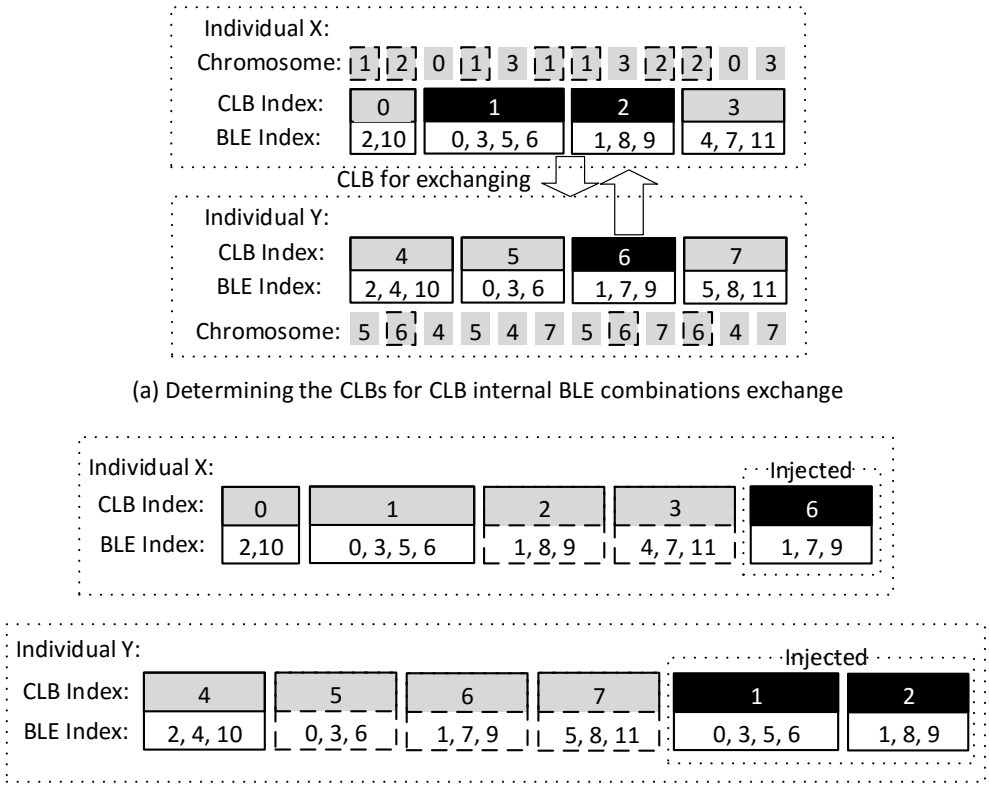


Figure 6.2: Determining the CLBs for performing the crossover operation. (a), the CLBs are determined randomly, and dashed-box genes indicate that gene represented BLEs are appeared in the selected CLBs. (b), these selected CLBs are directly injected in two individuals of each other.

the crossover CLBs, the exchange-purposed CLBs, have to be selected. For example, in the figure, the crossover CLBs are the three CLBs shown in dark black, and these CLBs will perform the CLB internal BLE combination exchange. In fact, in the GGAPack crossover, the number of crossover CLBs in each individual is configurable, and controlled by a pre-defined range. To mark the selected CLBs, CLB indexes are used. However, the GGAPack individuals contain a number of same index CLBs. For instance, there are two individuals, and these individuals have the same index CLBs but describe different BLE combinations. Therefore, the CLB "index" is only meaningful

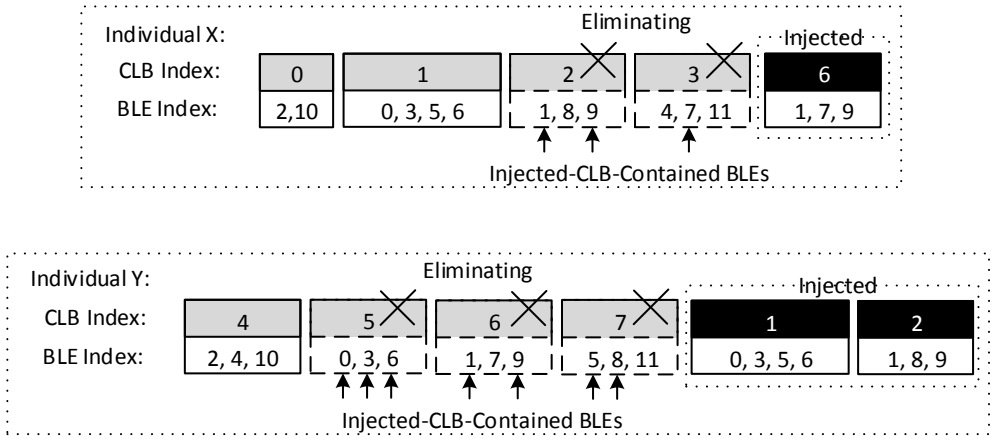


Figure 6.3: Injecting CLBs into each other and eliminating CLBs that contain injected BLEs, this figure continues Figure 6.2

in an individual itself. Note that the use of the CLB index to identify BLE combinations has no impact on the crossover operation as the crossover operation only uses the index to find a CLB in the crossover individual, and is not concerned with the actual meaning of the index – value.

After determining the crossover CLBs, these CLBs are directly injected into two "crossover individuals" of each other, as shown in Figure 6.2 (b). Subsequently, the crossover operation checks injected-CLB contained BLEs, and eliminates the individual CLBs that contain the injected BLEs. The BLEs that do not appear in the injected CLBs need to be freed for reinsertion. Figure 6.3 shows an example of the elimination process. Once the injection process is finished, the injected CLBs are reserved in the injected individual, as shown in Figure 6.4. Next, the crossover operation has to reinsert freed BLEs and these two individuals are stored as offspring. In the GGAPack, a random reinsertion is implemented. Under the clustering constraints, these BLEs are randomly inserted to individual current CLBs. To check the clustering constraints, BLEs need to be converted to actual sub circuits. This conversion will be introduced in Section 6.3.4. Note that once a BLE combination is determined (a subcircuit), the number of BLEs and input numbers are calculable. If these BLEs cannot be fully inserted into the current CLBs, new

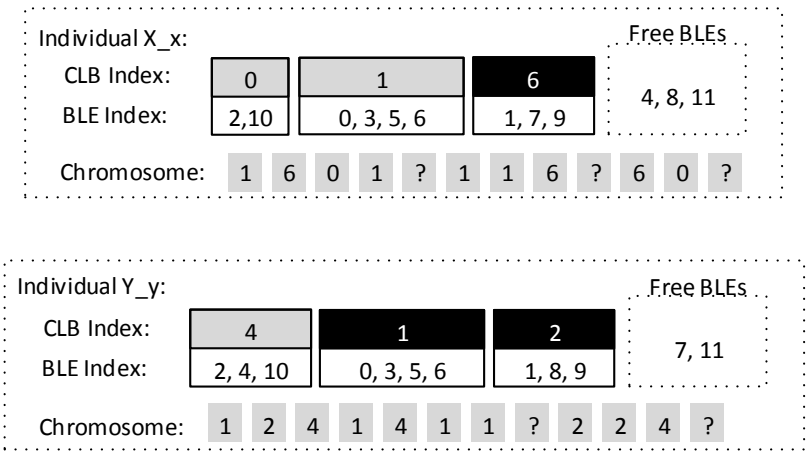


Figure 6.4: After the CLB exchange, a few BLEs are freed from CLBs. Meanwhile, the injected CLBs, referred to CLB internal BLE combinations, have been exchanged. The question mark gene means that the gene does not have a value – matched to freed BLEs. This figure continues Figures 6.2-6.3

CLBs are created with new index numbers assigned.

Mutation in GGAPack

The mutation operation is executed after the crossover operation to generate enough offspring individuals to constitute a population. In GGAPack, the mutation operation is designed to randomly eliminate two CLBs in one crossover generated individual – offspring. However, it could be a problem whether or not the mutation operation destroys the solutions which are generated by the crossover operation. In practice, this has little chance of happening as each individual contains a number of CLBs. Figure 6.5 illustrates the GGAPack mutation operation. Individual Z is an individual before the mutation. Individual Z_z is a possible individual after the mutation. Similar to the crossover operation, these freed BLEs need to be reinserted, where the BLEs can insert to current CLBs, or new CLBs with new CLB index numbers.

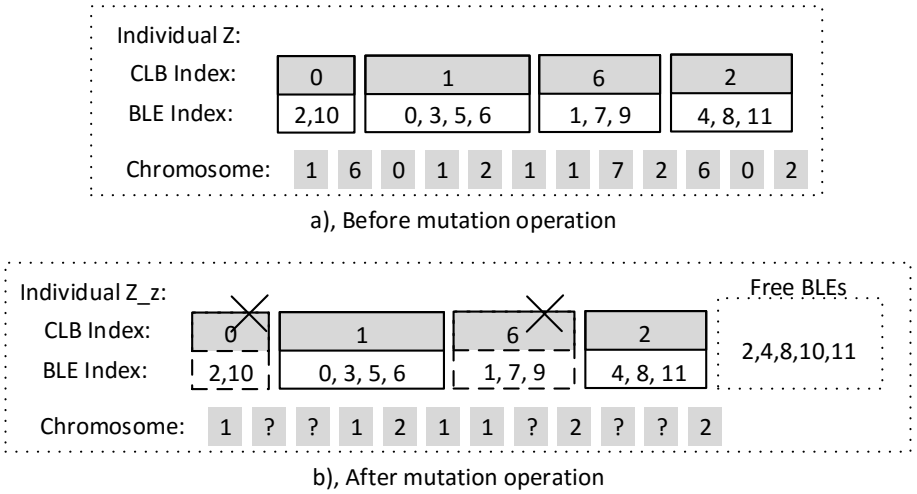


Figure 6.5: The GGAPack mutation operation randomly eliminates two CLBs. In the figure, the CLBs are the *CLB-0* and *CLB-6*. After the mutation, the BLEs with these two CLBs are freed. These freed CLBs are needed to insert back to existing CLBs, or new CLBs.

6.3.3 Multiobjective selection

It is difficult to evaluate a solution that relies on a single or a weighted fitness function, especially for real world problems. The reasons are referred to Chapter 4. Circuit clustering is such a real world problem, and comes with a few conflicting objectives that need to be balanced or judged. Therefore, single objective GA cannot meet, or hardly meet, the complex clustering requirements. Unfortunately, the GGA is a single objective GA, and it limits the application areas that it can solve. To better solve the circuit clustering problem, a multiobjective evaluating mechanism is incorporated into GGAPack.

The multiobjective evaluating mechanism indicates how the GA sorts its individuals under multiple fitnesses – objectives, and how the GA selects the individuals for further evolutions. In GGAPack, the NSGA-2 method (Deb et al., 2002) has been utilised to sort and select individuals for next evolving iteration. The "sort" is based on the Pareto optimality (Pareto, 1906), and

the "select" is facilitated by the crowding distance (Deb et al., 2002). The multiobjective system, in GGAPack, is implemented to replace the ranking-based method in the single objective GGA. In NSGA-2, the Pareto optimal ranking is achieved by a fast-non-dominated sort, and the detailed algorithm is shown in Appendices in Algorithm A.2. In the fast-non-dominated sort, each fitness value of an individual can be compared and set up domination relations to other individuals' fitness values. After the sort, individuals P are sorted on different Pareto fronts F_i. Individuals on the first Pareto front, F_1, are the best non-inferior individuals – solutions.

The number of sorted individuals is greater than the population size as the input of the sort is a sum of the population and the offspring. In the next evolutionary iteration, only a few better individuals, half of the sum, have to be used as a new population. However, the sorted individuals are on different Pareto fronts, and therefore cannot directly be picked. Although we need the individuals that are on Pareto fronts, normally the first Pareto front, individual number is usually mismatched to the GA population size and falling into three situations:

1) The first Pareto front has the same number of individuals as the evolutionary process needs

2) More first Pareto front individuals are found and greater than the GA population size, note that the number of sorted individuals is greater than the GA population size – GA intermediate population – referred to Figure 6.7.

3) Less individuals are on the first Pareto front, different Pareto front individuals are required.

Only in the first situation can the first Pareto front individuals be directly used as a new GA population. Otherwise, additional selection work is required.

To select individuals and preserve a better individual diversity, the NSGA-2 defines the concept of crowding distance, also called density-estimation

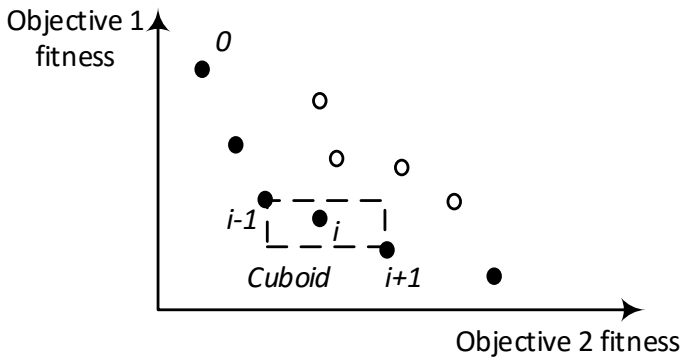


Figure 6.6: Crowding distance calculation of an individual. Solid black dots represent the same Pareto front individuals under the objective-1-and-2-fitness space. The crowding distance of individual i can be calculated by nearest neighbours – individual $i-1$ and $i+1$ – the perimeter of a cuboid which is formed by individual $i-1$ and $i+1$.

metric, to estimate solution density surrounding a particular solution, and use a crowed-comparison operation to select suitable solutions (individuals) for continued evolutions.

In the first step of crowding distance based selection approach, it calculates the average distance of two points on either side of a targeted point along with each of the objective fitness. The crowding distance of the targeted point refers to the quantity that is estimated by the perimeter of a cuboid which is formed by using the nearest neighbours as vertices. This can be illustrated by Figure 6.6, where these solid black dots indicate same Pareto front individuals, and the crowding distance of i is represented as the average side length of the cuboid, which is shown as a dashed box. To assign the crowding distance which is associated with all objective fitnesses, it requires to sort all individuals according to each objective fitness in an ascending manner. For each objective, an infinite distance is always assigned to boundary individuals. All other individuals are assigned a distance value based on a normalised difference in objective fitnesses of two adjacent solutions as shown in Figure 6.6. The overall crowding distance of an individual is then calculated as the sum of each distance value corresponding to every objective fitness. Algorithm A.3

in Appendices clarifies the crowding distance assignment.

In the second step of crowding distance based selection approach – the crowed-comparison operation, this operation will select required individuals from Pareto fronts in order to match GA population size. If many individuals are on the same Pareto front, for example 1^{st} Pareto front, crowed-comparison operation will select the individuals that only have larger crowding distances. This also indicates that the boundary individuals are always selected. On the other hand, if, for example, fewer individuals are on the 1^{st} Pareto front, the crowed-comparison operation will select the individuals on the following Pareto fronts based on the indivdiual's crowding distance.

6.3.4 Evaluating the evolved designs

In the GGAPack, multiple fitness functions are defined to evaluate the individuals. To calculate fitnesses, individual chromosomes are converted in two steps: First, the chromosome is translated to CLBs by the chromosome representation method that is introduced in Section 6.3.1. Second, the CLB features are processed using their BLE pin properties, and these CLBs circuit features are accumulated as entire clustered circuit properties used in fitness calculations. To produce the fitness, after the GGAPack reads the synthesised circuit, after the pattern matched, BLE connections are checked. During the check, the following information is obtained for each BLE pin:

1) Is the pin used as the whole circuit IO (otherwise known as system IO)?

2) If the pin is an input, which (BLEs) pins share the same input? Which (BLE) pin drives this BLE?

3) If the pin is an output, what are the fanout pins for this pin?

Once a set of BLE is determined and grouped, the BLE pins are unique. Using the above information, circuit properties of the BLEs are carried out as follows:

1) How many BLEs in the set? and what are the BLEs?

2) What connections are inside the set of BLEs?

3) Which signals need to be used to connect other sub circuits? The connections of a sub circuit refer to the connections of a CLB or further understood as CLB interconnects.

As mentioned in Section 6.3.2, the above conversion is also used by the genetic operations for reinserting the freed BLEs back to the individual CLBs.

To achieve basic circuit clustering targets which optimise CLB number and CLB interconnect number, the GGAPack has implemented three fitness functions, as shown in Equations 6.1-6.2.

$$f_{\text{obj1}}(x) = \#\ of\ CLBs \tag{6.1}$$

$$f_{\text{obj2}}(x) = \#\ of\ global\ nets \tag{6.2}$$

$$f_{\text{obj3}}(x) = (\#\ of\ CLB\ absorbed\ nets)^{-1} \tag{6.3}$$

These fitness functions are designed to return smaller values when a better individual is found. Equation 6.1 describes the number of CLBs. For a clustered circuit, fewer CLBs are expected. Equation 6.2 indicates the number of interconnects between CLBs. Equation 6.3 reflects an indirect factor, which is the connections the CLBs include. Both Equation 6.2 and

Equation 6.3 push the evolution to find individuals, or saying solutions, that have fewer interconnects between CLBs.

6.3.5 Summary of GGAPack

At the beginning of the GGAPack, initial individuals are randomly generated, where each BLE is randomly allocated to a CLB. This means that every initial individual has BLE number CLBs. The initial individual number is set to two times the population size, and these individuals are exported to the GA loop. In the GA loop, fitnesses of these individuals are assigned by multiple fitness functions as described in 6.3.4, and half of the best individuals are selected as a new population by the multiobjective selector. Next, the new population produces their offspring via the GGAPack genetic operations. GGAPack then combines the population and the offspring together for the next evolution. In GGAPack, the GA is executed for a fixed number of generations. On the last generation, a final individual will be filtered from the first Pareto front. The individual that has the fewest CLBs and fewest interconnects between CLBs is used as the ultimate solution. Subsequently, the individual represented solution is translated and saved as a new netlist. The GGAPack flow chart is shown in Figure 6.7, and the detailed algorithm is summarised in Algorithm A.4 in Appendices.

6.4 Initial experimental results

6.4.1 GGAPack experimental setups

Initial experiments have been set up for investigating GGAPack results. Since the GA outputs are stochastic, the GGAPack was executed for one hundred times, and the results are collected for statistical analysis. The targeted FPGA architecture was the same as used in Section 5.5 for VPack and

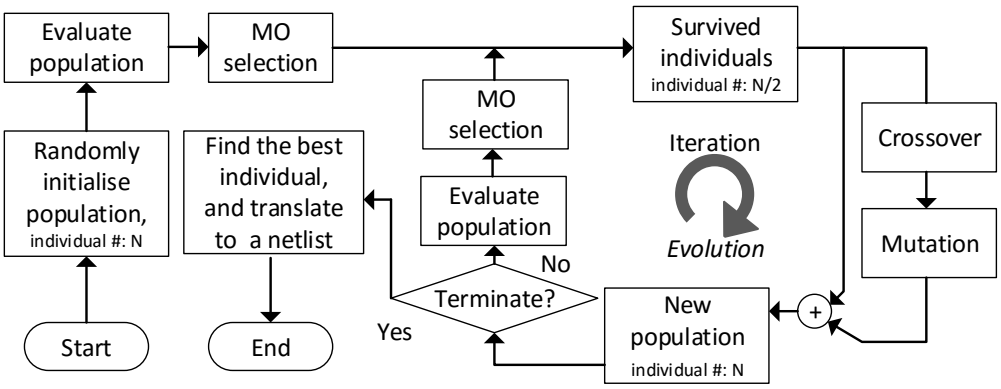


Figure 6.7: The flow of GGAPack: The population is initialised randomly – each BLE is in a CLB with a random CLB index. Individuals are assigned multiple fitnesses by multiple fitness functions. MO selection uses the non-dominated sort and crowding distance (NSGA-2 method) to form new population. GGAPack, the GA, iterates for a fixed number of generations then stops. The best individual is filtered and translated as a netlist.

RVPack, where the CLB has $I = 18$, $N = 8$, one clock and the BLE contains 4-input LUT and a reconfigurable FF, and the experiments are based on the MCNC-20 benchmarks (Yang, 1991). For the initial experiments, GGAPack outputs, solution-quality-related results, are directly compared to other circuit clustering algorithms, which are the CLB numbers and CLB interconnect numbers. The detailed testing flow is shown in Figure 6.8. As the pattern match is a duplicated process, each GGAPack program starts from the pattern matched netlist – the BLEs. These tests are still carried out on the same high performance computing cluster, referred to Section 5.5, where execution time of each program is the cluster-processor-occupying time.

Before loading GGAPack to the computing cluster, a number of GA parameters have to be chosen in order to produce useful results: First, according to the optimal CLB number of the MCNC-20 benchmark, the generation number is estimated. This is accomplished using the largest benchmark "clma". The "clma" has 8,383 BLEs, when clustering this benchmark into a 8-BLE CLB, the optimal CLB number is 1,048 (8,383 over 8). In GGAPack,

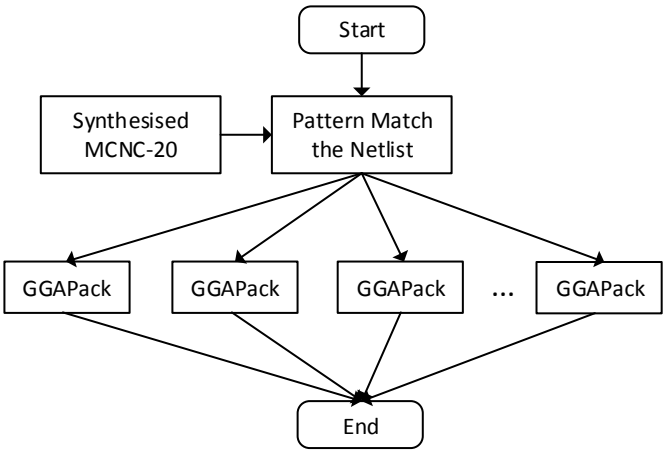


Figure 6.8: GGAPack executing and testing flow. Before forwarding the synthesised MCNC-20 netlist (LUTs + FFs) to GGAPack, a duplicated process, the pattern match, can be first performed, so that the GGAPack deals with the BLEs directly.

mutation is the primary operation to create the CLBs, and in each generation two CLBs are eliminated. If this operation is carried out on all CLBs (the initial individual has one BLE per CLB), this will mean that the smallest mutation number is at least 4,192 (8,383 over 2). However, that is an ideal situation as the mutation does not happen to every effective CLB. In practice, the random-CLB-eliminating mutation can also occur to well clustered CLBs. In order to move all BLEs in an optimal CLB number, or a near optimal CLB number, the generation number is set to ten times the minimum requirement – the smallest mutation number. To simplify the problem, the GA generation number of each benchmark is rounded to 40,000, and this can also ensure smaller benchmarks have enough generations to evolve. Figure A.2 in Appendices shows GGAPack GA convergence under different generations. Test indicates that 40,000 generations are enough.

Second, the crossover rate of GA is adjusted. To get an efficient crossover rate, the crossover rate has been tested under a range, which is from 0.2 to 0.8 based on the GA empirical settings that are introduced in Chapter 4. The testing shows that if the rate is set to 0.6, as shown in Figure 6.9

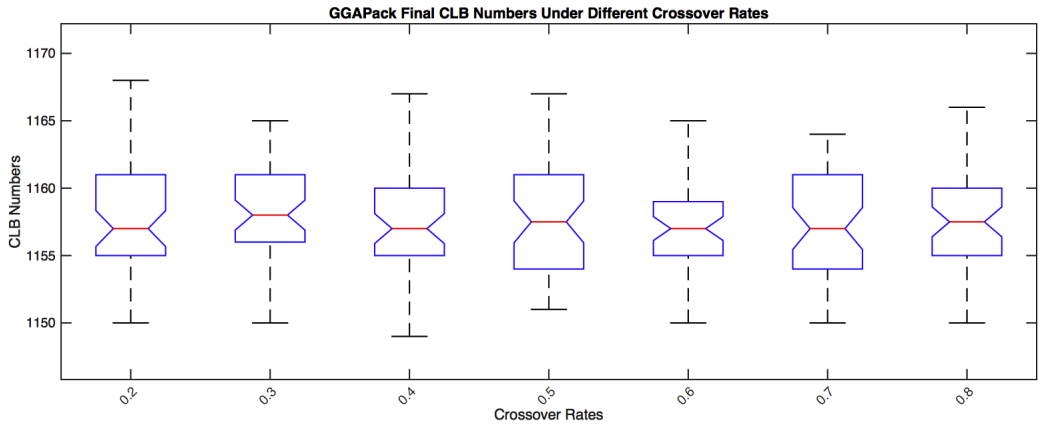


Figure 6.9: Box plot of CLB numbers vs. different crossover rates of GGAPack executions. The test is based on "clma" – the largest benchmark in MCNC-20. For each crossover rate, GGAPack executes for 100 times, final CLB number means that each GGAPack executes for 40,000 generations. When the crossover rate is 0.6, the GA result variation is small, and CLB number is small as well.

– testing GGAPack performance of different crossover rates based on the largest benchmark "clma", the GA can achieve small number of CLBs in a short time, and its results also have small variations. The figure only shows the testing results of "clma", however, the crossover rate is not linked to a particular benchmark; even when the benchmark is changed, the GA is still efficient. For the population size, to speed up the GA, it is set to 10, as, in GGAPack, experiments show that the population size does not significantly affect the quality of GA results, where this can be proofed by the graph of GA convergence vs. different population size as shown in Figure A.3 in Appendices.

The GA parameters for GGAPack are summarised as follows:

1) Population size: 10.

2) Crossover rate: 0.6

3) Mutation rate: eliminating 2 CLBs per generation.

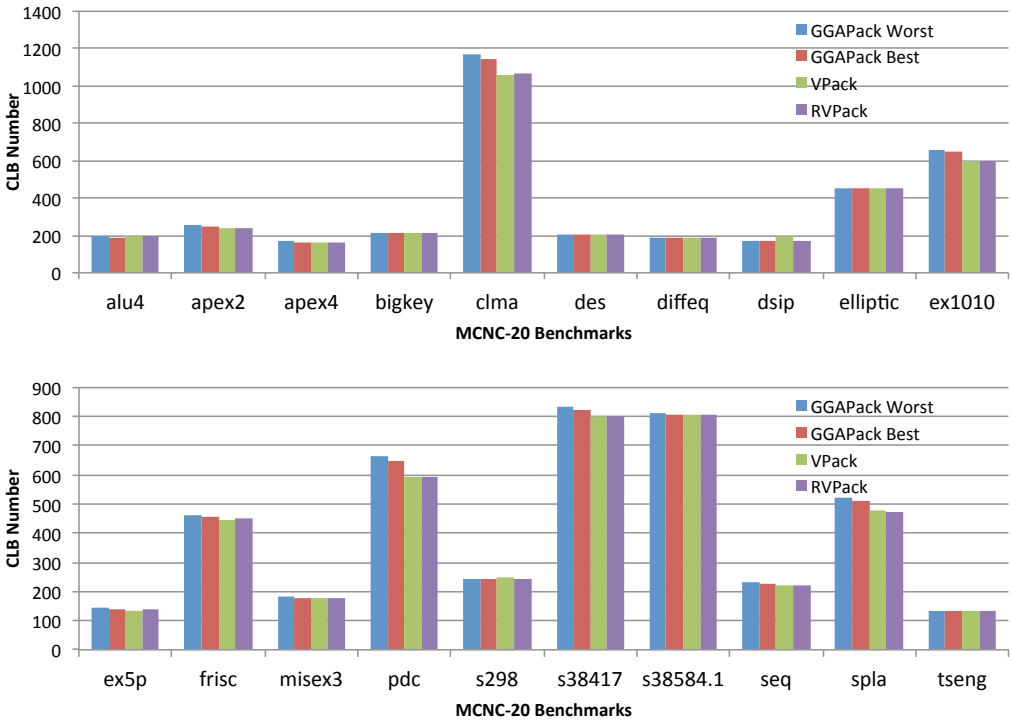


Figure 6.10: GGAPack clustered CLB number for MCNC-20 benchmarks compared to VPack and RVPack, lower is better. Data boxplot and detailed data are provided in Appendices in Figure A.12 and Table A.10.

4) Generation number: 40,000

6.4.2 GGAPack direct outputs

This section presents initial experimental results from GGAPack. These results are directly compared. The experimental result analysis is similar to RVPack, which is described in Chapter 5. In this analysis, VPack and RVPack, (referred to its best case results) are used as baselines, because the direct outputs of GGAPack are in competition with them. Detailed results and variation box plots have been included in Appendices.

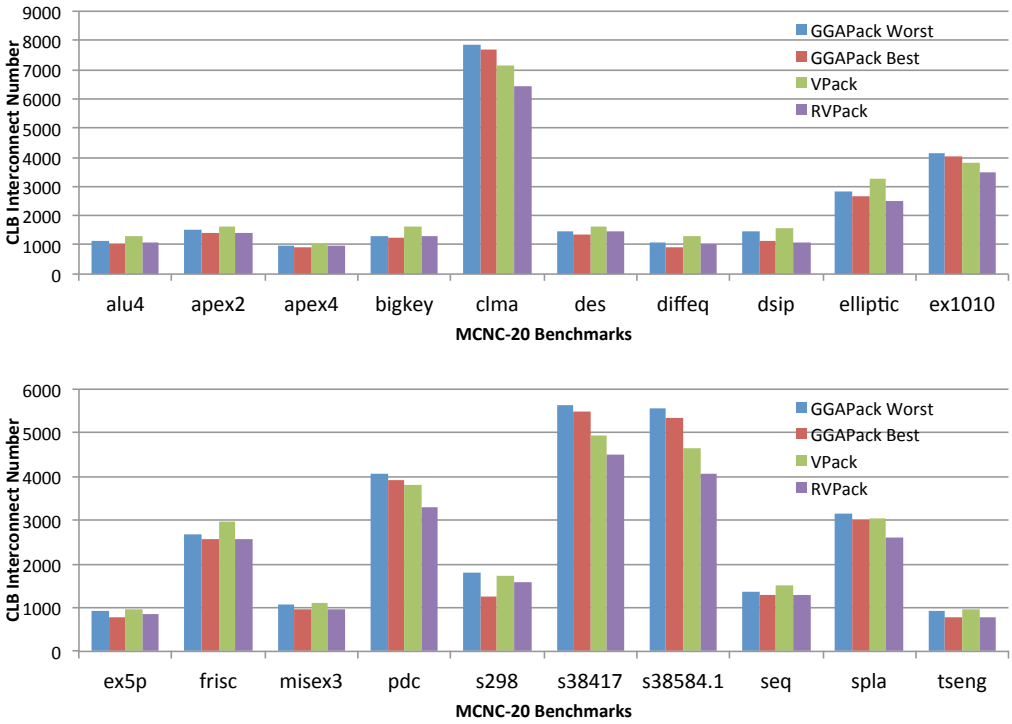


Figure 6.11: GGAPack clustered CLB interconnect for MCNC-20 benchmarks compared to VPack and RVPack, lower is better. Data boxplot and detailed data are provided in Appendices in Figure A.13 and Table A.11.

CLB usage

Figure 6.10 shows clustered CLB number comparisons for the MCNC-20 benchmarks, the comparisons covers GGAPack, VPack, RVPack. The sums of GGAPack CLB number in the worst and best cases are 7,892, 7,780. These numbers are greater than the VPack's 7,551 and RVPack's 7,534. As can be also seen from the figure, the GGAPack has more clustered CLBs when dealing with large benchmarks.

CLB interconnect

Figure 6.11 compares CLB interconnects between GGAPack, VPack and RVPack. The sums of the MCNC-20 benchmark CLB interconnects for GGAPack CLB interconnect number in the worst and best cases, VPack and RVPack are 50895, 47825, 49840 and 43349 respectively. In the best case, GGAPack has 4% improvements compared to VPack, but its results are worse than RVPack. According to Figure 6.11, it clearly shows that the worse CLB interconnect number is mainly caused by a few large benchmarks. This means that GGAPack has a limited performance for large scale circuits.

6.5 Further experimental results

6.5.1 Seeding GGAPack with semi-optimal solutions – GGAPack2

The GGAPack algorithm has been initially tested using the MCNC-20 benchmarks, referred to the previous Section 6.4, and the results show that GGAPack has better clustered results for small scale circuits. These results are better than VPack and RVPack (best case). However, the GGAPack performance is poor for large scale circuits. The main concern of this phenomenon is that the GGAPack has an insufficient evolutionary time or evolution lasts too short. GGAPack builds solutions from scratch as well as using a global perspective. This implies that the GGAPack has to deal with a huge searching space. Although the GA generation numbers can be increased, the actual number is not able to be infinite. Moreover, optimisation time is also important.

To improve GGAPack solution qualities, the GA initial conditions can be provided so that the GA searching space is narrowed. To achieve this, a set of semi-optimal solutions can be used as initial population of GA.

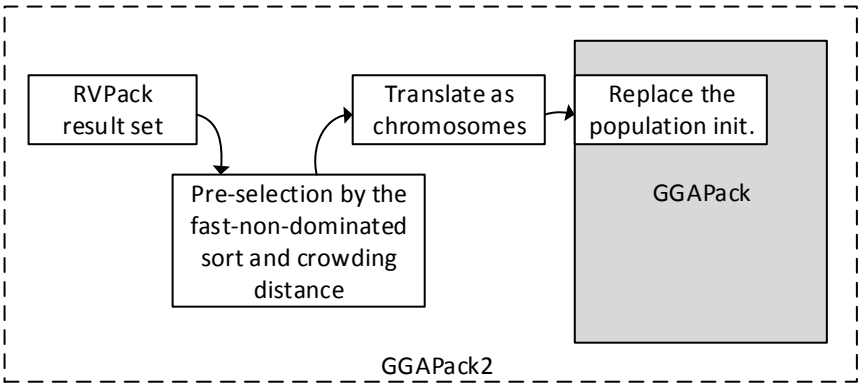


Figure 6.12: GGAPack2 working flow: A new mechanism has be added in GGAPack2 which allows it to read clustered solutions that are produced by other circuit clustering algorithms. This mechanism reads the solutions and translates them as chromosomes for GGAPack individuals.

In order to distinguish from GGAPack, GGAPack that incorporates with initial conditions is called GGAPack2. The only difference is that GGAPack2 involves a new mechanism which allows it to use RVPack's results. Figure 6.12 shows GGAPack2 work flow. Other than generating the initial individuals randomly, GGAPack2 takes the RVPack's stochastic results as the GA initial conditions. Since RVPack supplies a number of solutions, and it is greater than the GGAPack population number, a pre-selection process is implemented. The principle of the selection is similar to the multiobjective selection, which utilises the same fast-non-dominated sort, crowding distance and fitness functions. By using the multiobjective selector, only the best and the number matched results are kept and translated as individuals in the GGAPack evolution.

6.5.2 GGAPack2 experimental setups

To investigate GGAPack result qualities, new experiments have been set up. Similar to GGAPack, GGAPack2 is executed for one hundred times for

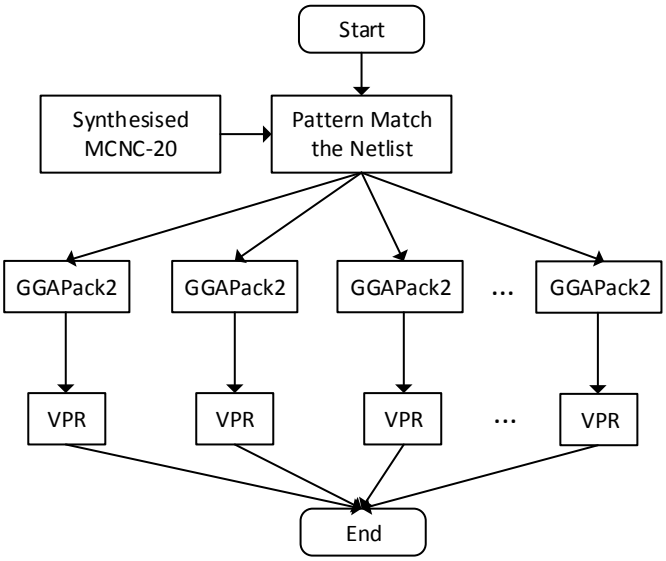


Figure 6.13: GGAPack2 executing and testing flow: Similar to GGAPack, each synthesised and pattern matched MCNC-20 benchmark netlist is processed by GGAPack2. GGAPack2 then produces new netlists to VPR for further testing.

analysing the stochastic features of results. The targeted FPGA architecture is the same as GGAPack, VPack and RVPack. As the GGAPack2 results are improved compared with GGAPack, the results of GGAPack2 not only cover GGAPack2 direct outputs, which are the CLB numbers, CLB interconnect numbers and the execution time, but also include GGAPack2 VPR results. The FPGA model used in VPR is the same as to Section 5.5, the VPR results cover real FPGA area usage, channel width, wire-length and the circuit critical path delay. GGAPack2 testing flow is illustrated in Figure 6.13. All the tests are based on the MCNC-20 benchmarks (Yang, 1991), and GGAPack2 starts from the pattern matched netlist. These tests are carried out on the same computing cluster.

According to GGAPack GA parameter settings, there are some changes in GGAPack2 which allow GGAPack2 fully benefits from RVPack's results – the GA initial conditions. Other than using the population size 10 in GGAPack,

GGAPack2 population size is set to 50, which the diversity of RVPack results can be better preserved. The increase of population size can also increase GA execution time. To compensate the long execution time, GGAPack2 generation number has been reduced from 40,000 to 10,000. The testing shows that this generation reduction does not significantly affect the GGAPack2 performance as GGAPack2 convergence is fast, where large generation number is useless.

The GA parameters for GGAPack2 have been summarised as follows:

1) Population size: 50

2) Crossover rate: 0.6

3) Mutation rate: eliminating 2 CLBs per generation.

4) Generation number: 10,000

6.5.3 GGAPack2 direct outputs

This section presents experimental results from GGAPack2. Apart from real FPGA mapping comparisons which GGAPack2 results are exported to the VPR, GGAPack2 direct outputs are also compared with GGAPack. The experimental result analysis is similar to RVPack and GGAPack. In this analysis, VPack, RVPack–best case results and RPack are used as baselines. The reason is that the direct outputs GGAPack2 are in competition with them. Detailed results and variation box plots have been included in Appendices.

CLB usage

Figure 6.14 shows clustered CLB number comparisons for the MCNC-20 benchmarks, the comparisons covers GGAPack2, GGAPack, VPack, RVPack and RPack. The sums of the MCNC-20 benchmark CLB numbers for above

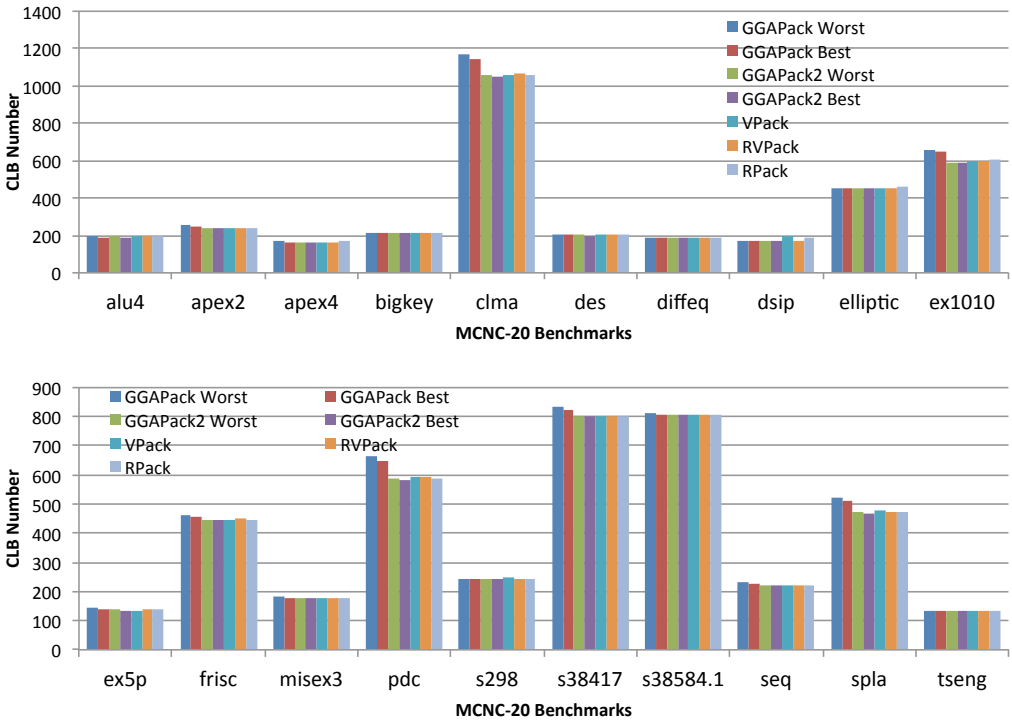


Figure 6.14: GGAPack2 clustered CLB number compared to GGAPack, VPack, RVPack and RPack for MCNC-20 benchmarks, lower is better. Data boxplot and detailed data are provided in Appendices in Figure A.15 and Table A.13.

circuit clustering algorithms are summarised in Table 6.1. The results show that the GGAPack2 can better reduce CLBs number for a clustered circuit. In the best case, GGAPack2 reduces CLB number up 0.79% 0.57% and 0.78% compared to VPack, RVPack and RPack respectively.

CLB interconnect

Figure 6.15 compares CLB interconnects between GGAPack2, GGAPack, VPack, RVPack and RPack. The sums of the MCNC-20 benchmark CLB interconnects for different circuit circuit clustering algorithms are summarised in Table 6.2. The results indicate that, even the worst case results, the GGAPack2 can better include connections in CLBs compared with the VPack

Table 6.1: Sums of clustered CLB numbers for the MCNC-20 benchmarks between different algorithms – sorted in ascending order, lower is better.

Algorithm	CLBs
GGAPack2 Best	7,459
GGAPack2 Average	7,475
GGAPack2 Worst	7,491
RVPack	7,534
RPack	7,550
VPack	7,551
GGAPack Best	7,780
GGAPack Average	7,836
GGAPack Worst	7,892

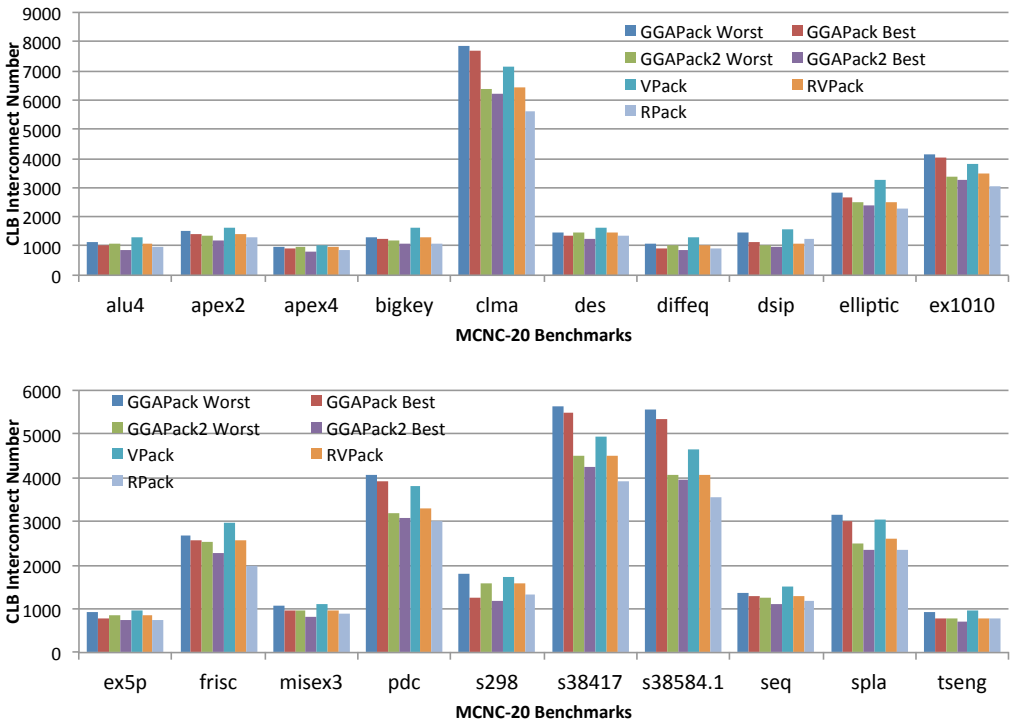


Figure 6.15: GGAPack2 clustered CLB CLB interconnect number compared to GGAPack, VPack, RVPack and RPack for MCNC-20 benchmarks, lower is better. Data boxplot and detailed data are provided in Appendices in Figure A.16 and Table A.14.

Table 6.2: Sums of clustered CLB interconnects for the MCNC-20 benchmarks between different algorithms – sorted in ascending order, lower is better.

Algorithm	CLB Interconnects
RPack	38,300
GGAPack2 Best	39,445
GGAPack2 Average	41,001
GGAPack2 Worst	42,557
RVPack	43,349
GGAPack Best	47,825
GGAPack Average	49,360
VPack	49,840
GGAPack Worst	50,895

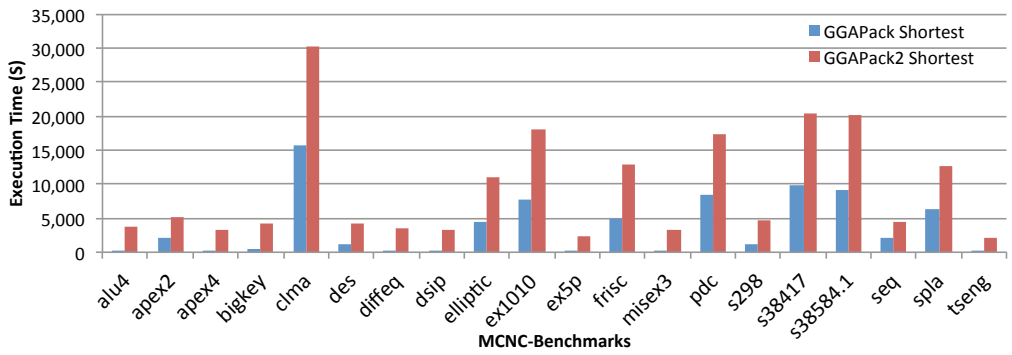


Figure 6.16: Shortest execution time compared to GGAPack and GGAPack2 for MCNC-20 benchmarks, lower is better. Data boxplot and detailed data are provided in Appendices in Figures A.14, A.17 and Tables A.12, A.15.

and RVPack, but not better than RPack. In the best case, GGAPack2 can reduce the CLB interconnects by up to 9.01% compared to RVPack, and 20.86% compared to VPack.

Execution time

Table 6.3 summarises the GGAPack, GGAPack2 longest and shortest (single run), RVPack average and VPack execution time. RVPack average execu-

tion time refers to the RVPack one-hundred-execution average time. The RPack execution time cannot be comparable since the RPack source code is inaccessible.

Table 6.3: Single execution time comparisons for GGAPack, GGAPack2, RVPack (average) and VPack

	GGAPack			GGAPack2				
Benchmark	Long.	Avg.	Short.	Long.	Avg.	Short.	RVPack	VPack
alu4	525	378	231	4,297	3,973	3,648	2.99	2
apex2	2,388	2,215	2,042	6,028	5,653	5,278	4.85	5
apex4	668	428	187	3,860	3,582	3,303	2.01	2
bigkey	2,092	1,318	544	4,922	4,607	4,292	3.78	4
clma	27,929	21,773	15,617	46,679	38,511	30,342	93.1	98
des	2,063	1,646	1,228	4,574	4,365	4,155	3.13	4
diffeq	623	428	233	4,038	3,759	3,480	2.99	3
dsip	737	442	147	3,602	3,490	3,378	2.34	3
elliptic	6,536	5,555	4,574	16,032	13,530	11,027	14.24	16
ex1010	11,745	9,729	7,712	25,590	21,834	18,078	28.47	30
ex5p	174	118	61	3,279	2,841	2,402	1.16	2
frisc	8,156	6,513	4,869	17,833	15,355	12,876	14.86	15
misex3	1,006	660	313	3,590	3,423	3,256	2.49	3
pdc	13,832	11,200	8,568	26,297	21,892	17,487	27.83	29
s298	2,136	1,631	1,125	6,656	5,681	4,705	4.74	5
s38417	14,963	12,417	9,871	31,888	26,198	20,507	48.4	49
s38584.1	13,323	11,250	9,176	29,138	24,666	20,193	45.2	50
seq	2,144	2,080	2,016	5,243	4,843	4,443	4	5
spla	8,738	7,598	6,457	18,790	15,771	12,752	17.67	19
tseng	58	34	9	2,250	2,169	2,088	1.03	1
SUM	119,836	97,408	74,980	264,586	226,138	187,690	325.28	345

Unit: Second
Long. = longest execution time
Avg. = average execution time
Short. = shortest execution time
RVPack = RVPack average execution time
Shorter time is better.

The table shows that the GA based method has a large execution time. However, it is undeniable that the GA consumes so much time because its results are produced by evolution. Figure 6.16 compares the shortest execution time for GGAPack and GGAPack2. If the MCNC-20 benchmark execution time is accumulated for both clustering algorithms, the total time for GGAPack and GGAPack2 will be 74,980 and 187,690 respectively. This shows that the GGAPack2 consumes more than double the amount of time than the GGAPack.

6.5.4 GGAPack2 VPR results

The rest of the experimental results are produced using VPR, and detailed variation box plots are attached in Appendices. The VPR emulates real FPGAs, and allows the clustered circuits to be further tested. In the following comparisons, GGAPack2 results are used. The first comparison is for the FPGA area usages. The area-testing-experiment conditions are similar to the RPack, so the results presented in the RPack literature are used in this comparison.

FPGA area usage

Table 6.4 shows the area usages for the clustered MCNC-20 benchmarks on the targeted FPGA between GGAPack2, VPack (for the RVPack, the best case results are the same as the VPack) and RPack. In both the best and worst cases, GGAPack2 uses the same areas. These results also suggest that GGAPack2 uses the same FPGA area as RPack for each clustered benchmark. The difference is that VPack occupies more area for the benchmark “alu4”.

Table 6.4: GGAPack2 on FPGA area usage, $X \times Y$ arrays, for MCNC-20 benchmarks compared to VPack, RVPack and RPack, lower is better. Data boxplot and detailed data are provided in Appendices in Figure A.18 and Table A.16.

Benchmarks	GGAPack2	VPack (RVPack)	RPack
alu4	14*14	15*15	14*14
apex2	16*16	16*16	16*16
apex4	13*13	13*13	13*13
bigkey	36*36	36*36	36*36
clma	33*33	33*33	33*33
des	42*42	42*42	42*42
diffeq	14*14	14*14	14*14
dsip	36*36	36*36	36*36
elliptic	22*22	22*22	22*22
ex1010	25*25	25*25	25*25
ex5p	12*12	12*12	12*12
frisc	22*22	22*22	22*22
misex3	14*14	14*14	14*14
pdc	25*25	25*25	25*25
s298	16*16	16*16	16*16
s38417	29*29	29*29	29*29
s38584.1	29*29	29*29	29*29
seq	15*15	15*15	15*15
spla	22*22	22*22	22*22
tseng	15*15	15*15	15*15

Channel width

Figure 6.17 shows the FPGA routing channel width between GGAPack2, RVPack and VPack. RPack is not involved as the results presented in its literature cannot be matched to the current VPR testing conditions, so the comparison cannot be set up. The difference in results could be caused by a different version of VPR, and that version VPR is not available. For the MCNC-20 benchmarks, the sums of the channel width for the GGAPack2 best case, worst case, RVPack and VPack are 1,204, 964, 864 and 986 respectively. Compared with VPack, in the best case, GGAPack2 produced solutions save

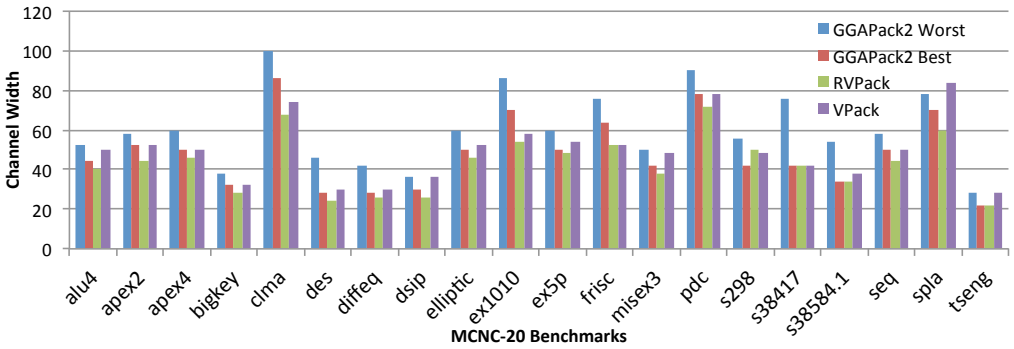


Figure 6.17: GGAPack2 on FPGA channel width compared to RVPack and VPack for MCNC-20 benchmarks, lower is better. Data boxplot and detailed data are provided in Appendices in Figure A.19 and Table A.17.

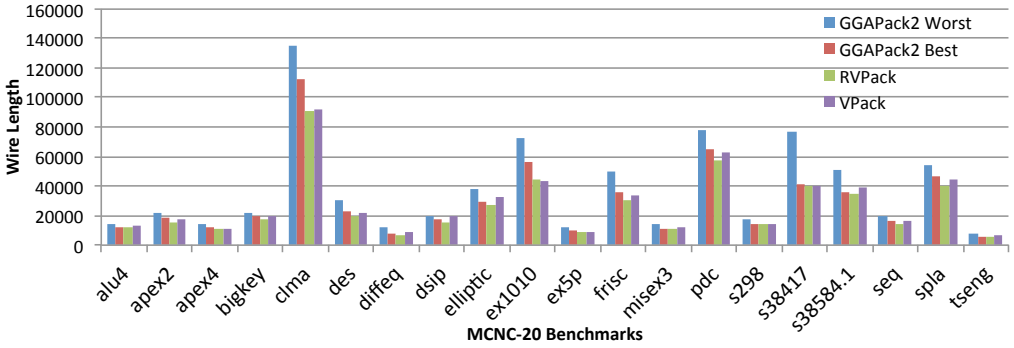


Figure 6.18: GGAPack2 on FPGA wire length compared to RVPack and VPack for MCNC-20 benchmarks, lower is better. Data boxplot and detailed data are provided in Appendices in Figure A.20 and Table A.18.

2.23% channels, or in the worst case, a GGAPack2 solution may use 18.11% more channels. In addition, the results also indicate that GGAPack2's best solution uses more channels than RVPack, which is 14.12% more channels.

Wire length

Figures 6.18-6.19 compare the wire lengths and critical path delays of clustered circuits, and these use RVPack and VPack as references. Figure 6.18 presents

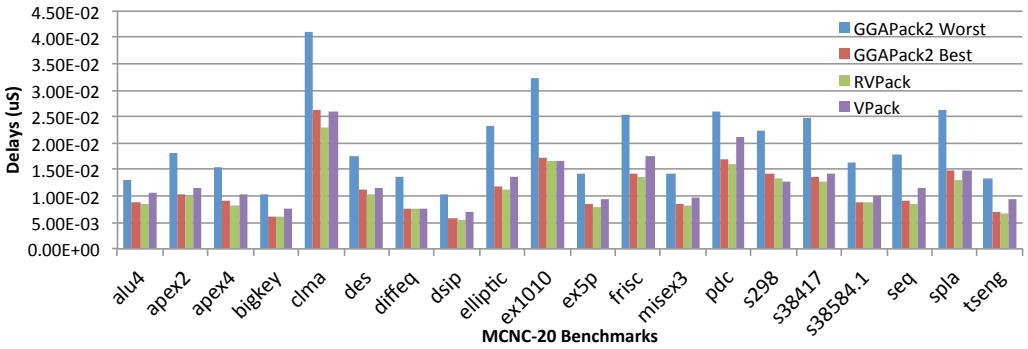


Figure 6.19: GGAPack2 on FPGA critical path delay compared to RVPack and VPack for MCNC-20 benchmarks, lower is better. Data boxplot and detailed data are provided in Appendices in Figure A.21 and Table A.19.

GGAPack2, RVPack and VPack summed wire lengths of the MCNC-20 benchmarks. The wire lengths of GGAPack2 worst case, best case, RVPack and VPack wire lengths are 761,798, 590,573, 518,179 and 558,315 respectively. In both best and worst cases, the GGAPack2 solutions are worse than RVPack and VPack.

Delay

Figure 6.19 presents the mapped-circuit critical path delays for GGAPack2, RVPack and VPack. For the MCNC-20 benchmarks, accumulated delays for GGAPack2 best case, worst case, RVPack and VPack are $3.95224 * 10^{-01}$, $2.30457 * 10^{-01}$, $2.16193 * 10^{-01}$ and $2.53159 * 10^{-01}$ μS respectively. In the best case, the GGAPack2 solution mapped on a FPGA are faster than VPack by 8.97%. However, in the worst case, the delays are higher than VPack by 35.94%. In both cases, the GGAPack2 solutions have poor performances on critical path delays compared with RVPack.

6.6 Discussion

In Section 6.4-6.5, results of GGAPack and GGAPack2 are compared. This section discusses the finds and problems from the experimental results of GGAPack and GGAPack2.

Firstly, the GGAPack direct output comparisons show that building clustered solutions from scratch and using a global perspective are not able to produce better solutions for all the MCNC-20 benchmarks, especially for the large benchmarks. The major cause of the less then optimised results is the short evolution time. As GGAPack searches the solutions from scratch, the GA searching spaces are huge. Therefore, a larger generation number is always expected. In fact, although the number could be increased, the evolution cannot be set to an infinite length. This has to take into the actual computing resources.

On the other hand, the use of the GGAPack clustering method (GGAPack2) to be a second level optimiser can improve the CLB usage, and reduce the CLB interconnects when compared with VPack, RVPack and RPack. If we only consider the circuit separation, which cuts a large circuit into pieces, GGAPack2 will be a useful method for producing better solutions. As a result, GGAPack2 can prove that the design of the GGAPack genetic operations are effective – CLB exchange and CLB elimination methods, as well as the use of multiple fitness functions are also useful for guiding the GA to find expected solutions.

For the execution time comparisons, it is noted that GGAPack2 uses more time than GGAPack. To compare these two algorithms, the obvious difference is that GGAPack2 uses a large population size, which requires more time. Despite GGAPack generation number being greater than GGAPack2, its actual execution time is shorter. This suggests that the fitness calculation and multiobjective sort are the most time-costed sections in the GGAPack algorithm. Moreover, the other reason that GGAPack2 has a longer execution

time is that GGAPack2 optimises the solutions from RVPack, where the GGAPack2 individuals have a number of optimised solutions. It indicates that GGAPack2 individual has more BLEs in a single CLB (since well clustered). Note that evaluating a fewer BLE CLB is faster than a CLB that has more BLEs.

In the real mapping tests using VPR, it shows that the GGAPack2 clustering method does not provide effective improvements for optimising the channel widths, wire lengths and critical delays. This has to recall the GGAPack BLE reinsertion process. To save routing tracks and wires and to reduce the circuit delays, the circuit clustering methods have to form more common connections between CLBs. Unfortunately, the GGAPack reinsertion process is a random operation. This reduces the real-mapping performance of inputted RVPack solutions, which means that the CLB interconnects will not be well arranged after the GGAPack execution. Though the GGAPack2 clustered circuit has less CLB interconnects, the real mapping performances are poor.

Beside the problems that are found in GGAPack, it also highlights a few meaningful aspects. The first is that GGAPack has provided a GA framework for the circuit clustering problem. It introduces the fitness conversion method, especially for circuit evaluations, and the multiobjective selection process. Because the multiobjective (MO) optimisation is used, this allows GGAPack to incorporate any additional objective fitness functions without changing the algorithm. However, to a certain extent, GGAPack also shows that clustering circuits from scratch using the GA and a top-down perspective can be inefficient.

6.7 Summary

This chapter introduced the GGAPack and GGAPack2 algorithms, which were the GA-based top-down circuit clustering algorithms. The motivation and

detailed implementations were presented, and the GGAPack and GGAPack2 results were compared with other state-of-the-art clustering algorithms. The experimental results showed that GGAPack was not as efficient as the greedy algorithms. In contrast, GGAPack2 has better direct-output results compared to some greedy-algorithm-based circuit clustering methods. This indicates that GA based method can partial solve the circuit clustering problem. In addition, these methods (algorithms) also helped define a set of potential processes of using the GA to solve a circuit clustering problem. However, VPR tests showed that the GGAPack2 clustered circuits have worst performances on FPGAs, such as channel widths, wire lengths and delays. On the other hand, by implementing these two methods, the major problems were identified, for example, it could be inefficient by using a GA-based top-down circuit clustering method, especially the circuit scale is large, as well as using the random reinsertion during the genetic operations, it might reduce the performance of a previously clustered circuit, referred to GGAPack2. This work can further inspire the circuit clustering algorithm designs, for example the DBPack in the next chapter.

Chapter 7

DBPack: Bottom-Up Circuit Clustering Approach Using MOGAs

7.1 Introduction

In Chapter 6, the GGAPack and GGAPack2 methods were proposed. These two methods are based on MOGAs, and constructing CLBs using a top-down perspective. The results indicate that GGAPack cannot effectively improve the clustered circuit on-FPGA and its performance. To cope with major problems identified in GGAPack, this chapter introduces a new MOGA based circuit clustering method, known as DBPack, and uses a new perspective to view the circuit clustering problem. DBPack utilises a number of discrete MOGAs to build CLBs, and builds CLBs from scratch. Clustering a circuit using DBPack algorithm assists the GA search ability, and enhances the GA fitness function efficiency for a single CLB. In addition, the use of MOGA can also extend the clustering flexibility.

The organisation of this chapter is followed as: Section 7.2 presents the motivation of DBPack. Section 7.3 explains the implementation of DBPack, including the GA representation, genetic operations, fitness function design and the solution selection. Section 7.4 describes how the experiments are created to all the testing of DBPack. Experimental results are presented in Section 7.5 with the discussion in Section7.6, and a summary in Section 7.7.

7.2 Motivation

Clustering circuits using a top-down perspective can be considered as an effective approach since CLBs are evaluated at the same time and the BLE combinations of CLBs are changed or optimised from a global perspective. However, the solution qualities of GGAPack, a MOGA-based top-down circuit clustering method, in particularly using GA to search an entire solution of clustered circuit, are poor.

The low-quality solution issue is caused by three major problems. First of these is the huge searching space in GGAPack. Building solutions from a global perspective are better for searching global optimal solutions, but possible BLE combinations can be an astronomical number. This results in GA having an extremely low convergence speed. However, such long evolution might not be produced by current computing systems. Second is the design of GGAPack genetic operations, which use a random reinsertion to reinsert freed BLEs back to CLBs after these operations. Unfortunately, the random reinsertion is a major problem that reduces the solution performance for real FPGA mappings as reinsertion process only focuses on CLB hardware constrains without considering CLB interconnects, for example, some circuit critical connections. Third is the design of fitness functions in GGAPack. Due to the fact that GGAPack individuals contain a complete solution, the fitness calculation of a solution has to be achieved from the global perspective. Therefore, it limits the sensitivity of the fitness function for evaluating a single CLB change, and finally results in that a GA is convergent at a local

optimality.

DBPack is a new circuit clustering algorithm based on the MOGA. It utilises a number of discrete MOGAs to build a solution. In the DBPack clustering process, each discrete GA focuses on a single CLB construction. It is similar to other bottom-up circuit clustering algorithms, but not the same – DBPack utilises a GA to directly group BLEs as a CLB instead of incrementally clustering single BLE into a CLB. This particular design means that the GA searching space is reduced, the fitness functions are effective to a single CLB and each CLB can be viewed as an entirety to be optimised. DBPack is short for Database Packer. The database describes that each CLB solution is identified from a large number of individuals – GA population (a set of solutions).

7.3 DBPack implementation

DBPack uses a new bottom-up perspective to build CLBs – clustering BLEs to CLBs. Rather than incrementally adding BLEs to a CLB, which is used in greedy algorithm based methods, DBPack utilises a MOGA to search a group of best BLEs (BLE combination) for a CLB – CLB per GA. When DBPack is in a process of building CLBs, clustered CLB interconnects are known, this means that an optimisation objective can be set up in MOGA to optimise CLB interconnects based on already built CLBs, which is from a global perspective. Although DPBack is a bottom-up algorithm, the global optimisation is considered. Moreover, since the building of CLB is facilitated by the MOGA in DBPack, this indicates that DBPack can work with a number of clustering objectives, which is flexible. Therefore, any clustering related objectives can be defined as fitness functions and added to DBPack without changing its core algorithm, for example, in this implementation, five clustering objectives are used. As with the GGAPack, DBPack reads a synthesised and pattern matched netlist as an input, and produces a new VPR compatible netlist. In this section, the DBPack implementation is introduced.

It includes the GA chromosome representation, genetic operations, fitness functions and solution selection. DBPack genetic operations cannot deal with clustering constraints, unlike the GGAPack, so an extra MOGA constraint-handling approach is required. This is accomplished by invoking penalties, and this method is introduced in Section 7.3.3.

7.3.1 Representation

Binary string has been used to encode DBPack GA's chromosome. The use of this particular encoding style has the following two benefits: First, this encoding scheme is simple, and the classic-binary-string genetic operations (Maulik et al., 2011) can be incorporated without complex conversions. Second, this chromosome representation is sufficient to provide the complexity for DBPack targeted-problem domain, this can be shown with the DBPack experimental results in Section 7.5.

The DBPack GA chromosome has a number of genes, and these genes are used to represent BLE selectivities for a CLB. The number of genes in the chromosome is determined by the unclustered BLE number, hence the chromosome length is variable. Each gene is used to encode each BLE index and its selectivity. A gene that has the binary value "1" means that the BLE index corresponding to that gene position is selected for a CLB. Otherwise, the gene has the binary value "0" suggests that the corresponding BLE is not selected. In DBPack, the circuit clustering is performed by a number of discrete GAs, and each GA starts with different chromosomes, but the longest chromosome appears at the first GA.

Figure 7.1 shows an example chromosome for a 12-BLE circuit clustering problem. In this example, the "1" valued genes, which are *BLE-2*, *BLE-3*, *BLE-6* and *BLE-8*, represent a possible BLE combination for a CLB. If these BLEs are the ultimate solution of the CLB, these BLEs are removed. The next CLB (GA) construction starts from the chromosome shown in Figure 7.2. However, relative BLE indexes are not changed, and these remaining

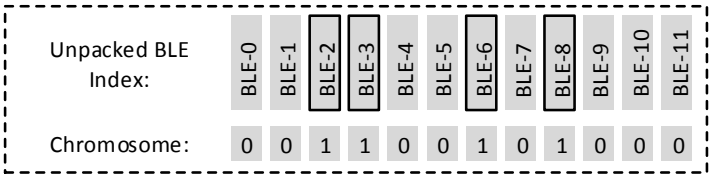


Figure 7.1: DBPack chromosome for 12-BLE circuit clustering problem: In the chromosome, the position of gene is used to encode the BLE index, and the gene value indicates the selectivity of corresponding BLE, where "1" means selecting the BLE, "0" indicates a non-selected BLE.

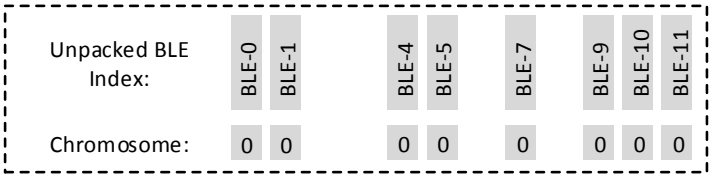


Figure 7.2: The remaining BLE indexes are based on the longest chromosome. If some BLEs are used to build a CLB, these BLE's genes are removed from the chromosome. Next GA chromosome is based on the remaining (unclustered) BLEs, but their indexes still use the longest chromosome.

BLE indexes are still based on the longest chromosome.

7.3.2 Reproduction

Crossover in DBPack

In DBPack, each discrete GA has both crossover and mutation operations implemented as the genetic operation in order to reproduce individuals. The crossover is a typical one-point binary coding crossover, and the crossover operation is controlled by a crossover rate, which determines how many individuals are crossovered or directly copied. In a GA generation, two individuals are randomly selected from the population to perform the crossover,

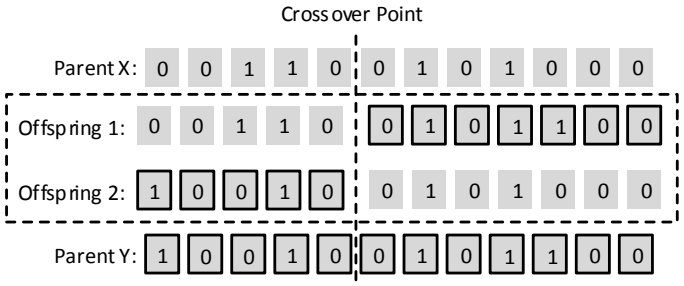


Figure 7.3: DBPack GA crossover operation. This crossover operation creates two offspring. The crossover point is determined randomly. Then original individuals are crossovered to generate offspring

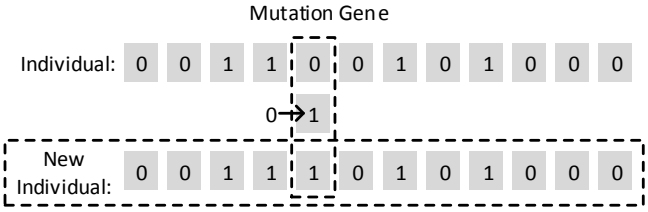


Figure 7.4: DBPack GA mutation operation. The mutation operation is the classic "flipping a bit " mutation, and it occurs to one crossovered offspring. Each mutation operation flips one gene, and only one gene.

and the crossover point of their chromosomes is randomly determined as well. Then these two individuals produce another two individuals. Figure 7.3 illustrates DBPack GA crossover operation.

Mutation in DBPack

The "flipping a bit" mutation operation (Maulik et al., 2011) is utilised in DBPack, and this mutation operation is shown in Figure 7.4. This mutation operates after crossover, and is performed on all crossovered offspring. For each crossovered offspring, DBPack mutation operation is designed to randomly flip one gene, and only one in the individual's chromosome. This is different to a

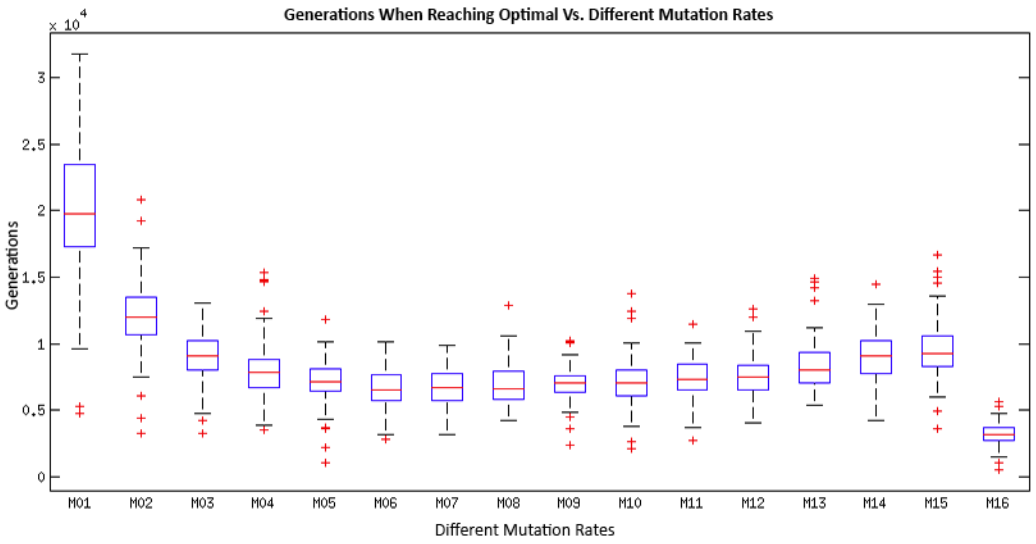


Figure 7.5: Box plot of generations vs. different mutation rates when finding a optimal solution. For each mutation rate, GA executes 100 times. M01: mutation rate 0.01%, M02: mutation rate 0.02% to M15: mutation rate 0.15%. M16 is one gene, and only one gene mutation operation.

standard binary-coding-chromosome mutation operation, where the standard mutation operation uses a mutation rate to control how many genes are flipped. The reason for implementing this mutation is that this unique design can save evolution generations in reaching DBPack clustering requirements. Figure 7.5 illustrates the relationship between mutation rates and generation number when using only mutation operation to find all BLEs of "alu4" benchmark (1152, BLEs, genes) – an "one max" problem – it is also an extreme condition of DBPack. In this test, each chromosome is randomly initialised – "0" or "1", and the "one max" means that all genes require to be "1". Apart from the single gene flip mutation, all others use a rate controlled mutation which the mutation ranges are from 0.01% to 0.15%. Compared with the rate-controlled mutation, this "flip only one gene" mutation reduces the number of generations by an average of 50% when the GA finds theexpected solution.

7.3.3 Multiobjective evaluation

Objective functions

At each CLB construction, DBPack involves a number of fitness functions to guide the GA evolution to search suitable BLEs. These fitness functions not only describe which objectives need to be optimised, but also handle clustering constraints. In DBPack, the MO selection is still based on the NSGA-2 (Deb et al., 2002), which selects the best individuals using the fast-non-dominated sort and crowding distance, and the detailed algorithm pseudocode is shown in Appendices in Algorithms A.2-A.3. To clarify clustering requirements, clustering objectives are preferentially described as objective functions, and are shown in Equations 7.1-7.5. Each function represents one optimisation objective of the searched BLEs, and all functions are defined to return smaller values when the function represented objective is better. The calculation of these objective functions uses BLE pin properties, where the individual chromosome is firstly converted to a BLE set, and these BLE circuit features are further determined by BLE input-and-output-pin relationships. This is similar to GGAPack fitness calculations in Section 6.3.4.

$$f_{\text{BLE}}(x) = (\#\ of\ BLEs)^{-1} \tag{7.1}$$

$$f_{\text{internal connects.}}(x) = \begin{cases} 2,\ (\#\ of\ inter.\ cons. = 0) \\ \\ (\#\ of\ inter.\ cons.)^{-1} \\ ,\ (\#\ of\ inter.\ cons. > 0) \end{cases} \tag{7.2}$$

$$f_{\text{increased nets}}(x) = \#\ of\ increased\ CLB\ interconnect\ nets \tag{7.3}$$

$$f_{\text{input}}(x) = \#\ of\ inputs \tag{7.4}$$

$$f_{\text{output}}(x) = \#\ of\ outputs \tag{7.5}$$

Compared with the global perspective clustering method GGAPack, these objective functions are redesigned to meet the clustering requirements in DBPack. Explanations of these functions are as follows: Equation 7.1 represents the number of BLEs for a CLB. To increase the CLB usage, a CLB has to fill with as many BLEs as possible. Equation 7.2 shows how many circuit connections a CLB can contain, or how many connections a particular BLE combination, the BLEs, can include, to put it another way. Similar to BLE number, if a clustered circuit has fewer interconnects between CLBs, it will mean that there are more connections inside CLBs. Equation 7.2, presents two situations: When the BLEs have no included connection, it returns a large constant. Otherwise, it presents a function relationship of CLB included connections. Equation 7.3 is to set up a global optimisation for CLB interconnects. If there are already clustered CLBs, current clustered CLB interconnects are known. When a new CLB is added, how many new interconnects appeared is calculable. If the "increased net" number has a low value, the clustered circuit has fewer CLB interconnects or more common connections within current CLB interconnects. Equations 7.4-7.5 are the controls of the CLB input and output numbers, and these are inspired by Rent's rule.

Rent's rule

$$T = tg^p \tag{7.6}$$

Rent's rule is an exponential relationship discovered by E. F. Rent (Landman and Russo, 1971). It is a trend between the number of connections T

at the boundaries of an integrated circuit and its internal logic component number g. Equation 7.6 explains the relationship. Apart from T and g, t is a constant. No matter how connection and component numbers are changed, p is a value that is always less than "1", and it is normally between 0.5 and 0.8. When the p is smaller, the integrated circuit boundaries have fewer connections. However, the integrated circuit contains more internal connections. In DBPack, this relationship is used and transferred in the designed objective functions. Equations 7.2, 7.4 and 7.5 represent this relationship. When a BLE combination, treated as an integrated circuit, has fewer inputs and outputs, as well as more internal connections, a smaller p is obtained.

Constraints handling

In the process of searching targeted BLE combinations, DBPack GA's operations cannot control clustering constraints, and solution evolutions are only dependant on GA fitness functions. Beside the objective functions, DBPack GAs have to have an extra mechanism to control the CLB input and BLE numbers when clustering a circuit for CLB bandwidth-constraint FPGAs. However, this cannot be achieved by simply adding new objective functions. If these constraints are presented as objectives in the MO selection, these objectives might dominate other objectives and lead to the GA producing invalid solutions. To handle constraints, penalties are implemented in the fitness functions to eliminate unsatisfied individuals – solutions. Equations 7.7-7.8 are the defined penalty functions. When the evolved BLE combinations are valid to these constraints, both functions have no effects. Once the BLE combinations are invalid for the targeted CLB type, these functions produce penalty violations. In these penalty functions, Equation 7.7 presents the BLE number constraint, and Equation 7.8 is to control the input number of the BLEs. Note that $N = 8$ and $I = 18$ are used in the following DBPack experiments, but these values are changeable to match to different FPGA architectures. A and B are two proportional coefficients. These two parameters adjust penalty violation levels. Experiments show, $A = 7$, $B = 2$ are the

most efficient settings, where the penalty violations have to be small enough. If these penalty violations are too large, penalty violations will degrade the GA population diversity, and result in no useful solutions found in GA.

$$f_{\text{BLE penalty}}(x) = \begin{cases} 0, \ (\#\ of\ BLEs \leq N) \\ \\ \#\ of\ BLEs\ /\ A \\ , \ (\#\ of\ BLEs > N) \end{cases} \tag{7.7}$$

$$f_{\text{input penalty}}(x) = \begin{cases} 0, \ (\#\ of\ inputs \leq I) \\ \\ \#\ of\ inputs * B \\ , \ (\#\ of\ BLEs > I) \end{cases} \tag{7.8}$$

Fitness functions

The objective functions and penalty functions have been set up for DBPack. According to Deb, Kalyanmoy (Deb et al., 2001, 2000b) and Srinivas (Srinivas and Deb, 1995) introduced method, which the penalty needs to add to all objective functions to handle constraints, DBPack fitness functions have been defined as in following Equations 7.10-7.11, where Equation 7.9 shows the sum of the penalties. Designing fitness functions in this way, when the GA has an invalid solution, the individual fitness values are higher. This leads to the individual being eliminated in the following evolutions.

$$f_{\text{penalty}}(x) = f_{\text{BLE penalty}}(x) + f_{\text{input penalty}}(x) \tag{7.9}$$

$$f_{\text{obj1}}(x) = f_{\text{BLE}}(x) + f_{\text{penalty}}(x) \tag{7.10}$$

$$f_{\text{obj2}}(x) = f_{\text{internal connects.}}(x) + f_{\text{penalty}}(x) \tag{7.11}$$

$$f_{\text{obj3}}(x) = f_{\text{increased nets}}(x) + f_{\text{penalty}}(x) \tag{7.12}$$

$$f_{\text{obj4}}(x) = f_{\text{input}}(x) + f_{\text{penalty}}(x) \tag{7.13}$$

$$f_{\text{obj5}}(x) = f_{\text{output}}(x) + f_{\text{penalty}}(x) \tag{7.14}$$

7.3.4 Solution selection

This section describes how a solution individual is identified from the GA final population. In DBPack, each CLB construction uses a MOGA, and the GA executes for a fixed number of generations. At the last generation, all individuals can be considered as possible solutions for a CLB. However, the individuals present a number of solutions. As a feature of MO, these solutions are not identical but form a trade-off Pareto surface in "x" dimensions, where "x" is number of objectives (objective fitness). Therefore, a solution selection process is required. The flow of DBPack implemented solution selection process is shown in Figure 7.6, and the detailed algorithm pseudocode is listed in Appendices in Algorithm A.5.

In order to identify the solution individual, the selection process checks all final generation individuals. Individuals that are on the first Pareto front and having n ($n = N$) BLEs and less than or equal I inputs, are temporarily stored. In practice, the GA might not find any solution which has $n = N$ BLEs, so n is deductible until individuals are found. The key to this process is to find all maximum BLE solutions. Subsequently, these temporarily stored individuals are ranked based on their included connections. The individual

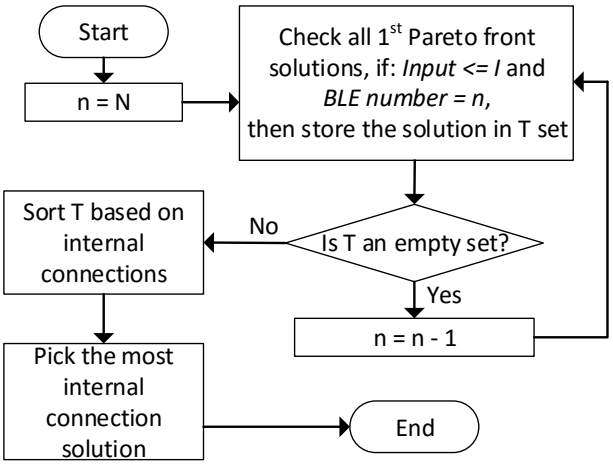


Figure 7.6: The flow of DBPack solution selection process. All 1^{st} Pareto front individuals in the GA's final generation indicate possible solutions for a CLB. If the selection process cannot find the suitable solution where $n = N$; a solution that has n BLEs, it will perform $n - 1$ and try the process again, until it finds the useful solution.

that has the most included connections is selected as a CLB.

7.3.5 Summary of DBPack

The DBPack execution flow is shown in Figure 7.7. The Pseudocode of DBPack algorithm is summarised in Algorithm A.6 in Appendices. DBPack clusters BLEs by using a number of discrete GAs, and the number of GAs are dependent on whether or not the algorithm has unclustered BLEs. Therefore, discrete GA number also indicates the number of CLBs. In each GA, the initial population is randomly generated and based on the unclustered BLEs. Here "random" means that individual chromosome is generated as a random binary string. Then the individuals are evolved under multiple clustering objectives. In DBPack, each GA produces its CLB based on unclustered BLEs. Until all BLEs are clustered, DBPack will translate these clustered CLBs as a new netlist.

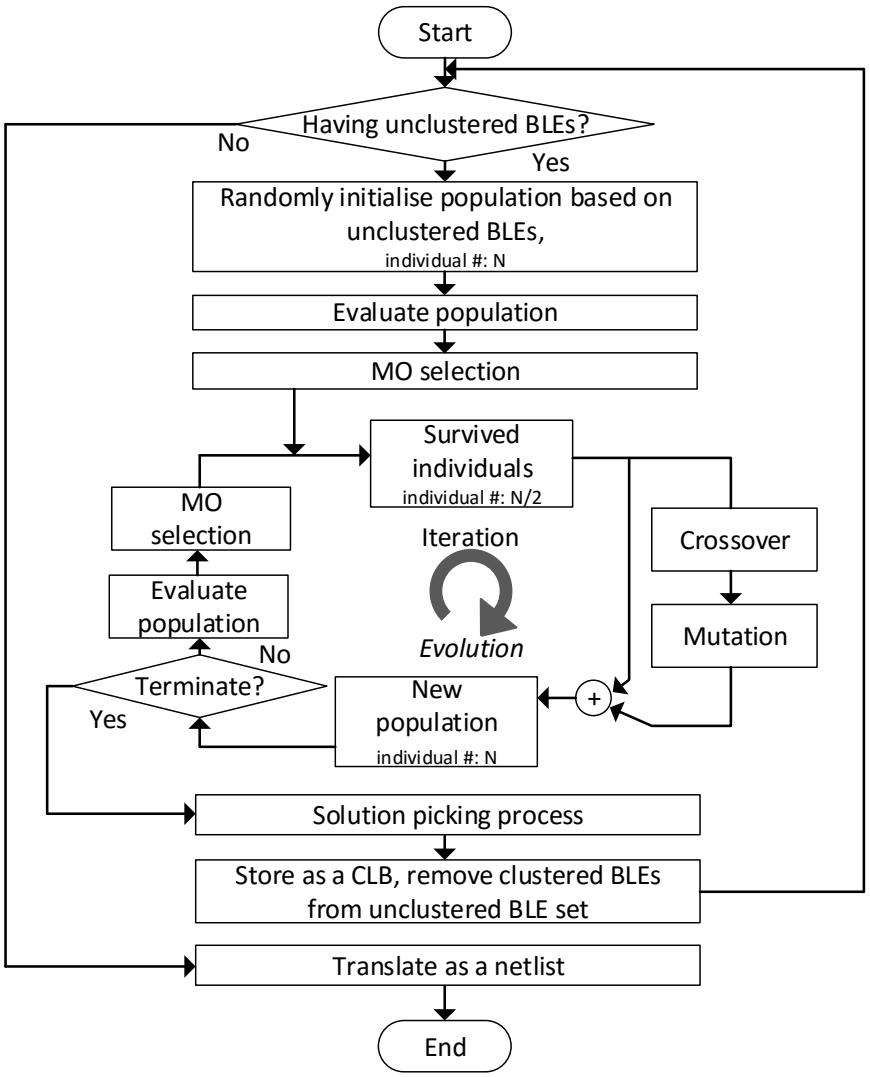


Figure 7.7: DBPack circuit clustering flow. GA evolution is similar to GGAPack. The major difference is that the DBPack uses a number of discrete GAs to deal with CLB constructions. Once all BLEs have been clustered in CLBs, DBPack will translate all CLBs as a netlist. This netlist can be tested using VPR.

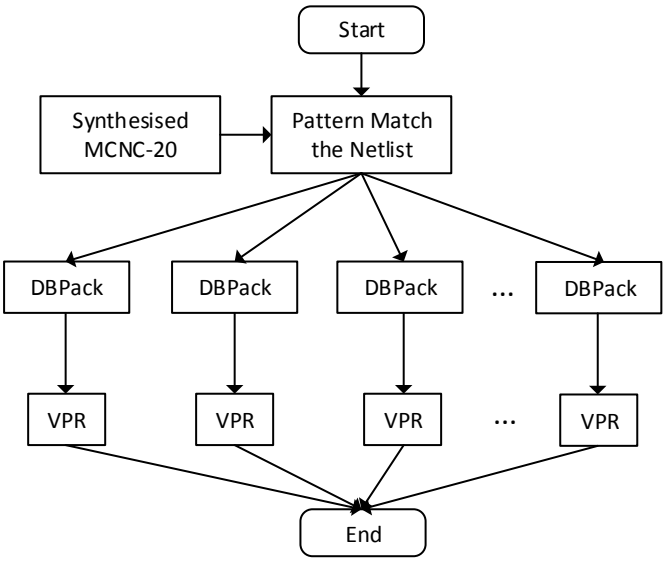


Figure 7.8: DBPack executing and testing flow: before forwarding the synthesised MCNC-20 netlist (LUTs + FFs) to DBPack, the duplicated process – the pattern match, is first performed, so the DBPack deals with synthesised BLEs directly. As these programs are executed on a 128-CPU computing cluster, there are maximum 128 DBPack and VPR programs can be executed simultaneously.

7.4 Experimental setups

To evaluate DBPack clustered circuits, experiments have been set up. The testing is based on the MCNC-20 benchmarks (Yang, 1991). For each benchmark, DBPack is executed one hundred times to investigate its stochastic features. Similar to GGAPack, DBPack uses the synthesised and pattern matched netlist as an input. After applying the DBPack algorithm, a new netlist is generated, and the netlist is readable for VPR (Luu et al., 2011; Betz and Rose, 1997b; Betz et al., 1999). The targeted FPGA architecture is an island style FPGA as shown in Section 5.5. In this FPGA, it contains a number of CLBs, and each CLB has $I = 18$ inputs, $N = 8$ BLEs and outputs. BLE comprises of a 4-input LUT and one reconfigurable FF.

Similar to RVPack and GGAPack, the experiments focus on two aspects: First, DBPack direct output comparisons – this includes the CLB number, CLB interconnects and the execution time. Second is the clustered circuit real-mapping-performance comparisons. DBPack outputs are exported to VPR, and VPR emulates the real FPGA mappings. The real mapping comparison contains FPGA area usage, channel width, wire length and the circuit critical path delay. All experimental results are produced by a computing clustering, referred to Section 5.5. As with previous clustering algorithm tests, the execution time is the computing-cluster-processor-occupying time. The DBPack execution and testing flow is illustrated in Figure 7.8.

To get satisfactory results, the GA parameters are calibrated. During the calibration, the largest MCNC benchmark "clam" is used. Three parameters are set for DBPack GA, which are the GA generation number, population size and the crossover rate.

At the beginning, the GA generation number is set to a large value, for example hundred of thousands, and the other two parameters are determined by random numbers. To speed up the GA, both the population size and crossover rate use small numbers. Using these settings, the results of first executed GAs, clustering the first CLB, are satisfied. Note that when clustering the first CLB, the GA has the largest searching space. "Satisfied" suggests that the GA can have a large number of individuals, the possible CLBs, on the first Pareto front. These individuals also have the most internal connections and their BLE numbers are maximum. However, satisfied individuals are significantly reduced when the unclustered BLEs are decreased – at the end of the clustering process. To maintain a selectable number of individuals on the first Pareto front, the population size is increased. An attempt is also made to reduce the generation number to reduce the run time. Note that the generation number was set to a huge number, so slightly reducing the number does not immediately affect the results. Figure A.4 in Appendices shows that the reduced generation number is still enough for DBPack to produce useful results.

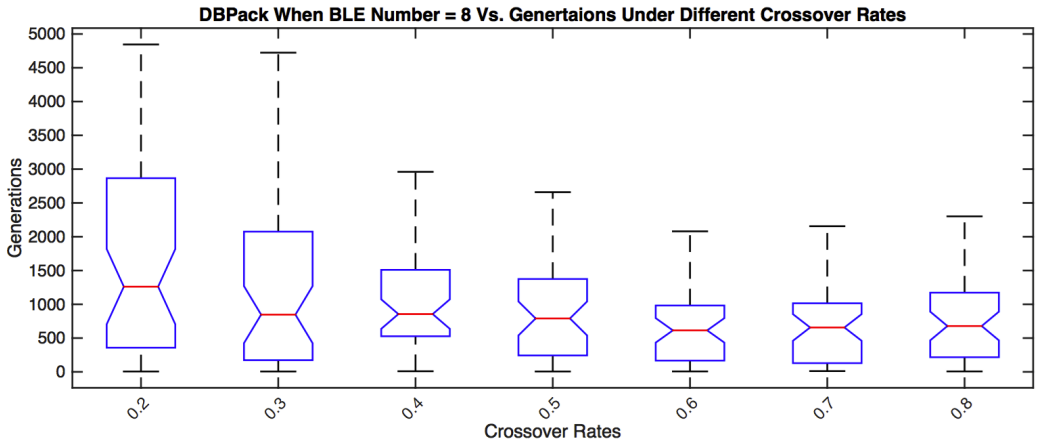


Figure 7.9: Box plot of generation numbers (when BLE number is 8) vs. different crossover rates of DBPack executions – clustering first CLB. The test is based on "clma" – the largest benchmark in MCNC-20. For each crossover rate, DBPack first GA executes for 100 times, and stops at when found BLE number is 8 (a CLB can contain 8 BLEs). When the crossover rate is 0.6, the GA generation numbers and variations are small when a solution has 8 BLEs.

Now keeping the generation number and population size of GAs constant, the crossover rate is further calibrated. The core purpose of this calibration is to adjust its rate to reduce the generation number required when the GAs find suitable solutions compared with uncalibrated GAs. Figure 7.9 shows that GA generation number used when a solution has 8 BLEs under different crossover rates. In this crossover rate test, experiments are set up based on benchmark "clma", and for each mutation rate, GA has executed for 100 times to show result variations. This test reflects when crossover is 0.6, GA is fast to find a solution that has 8 BLEs. Once the rate is determined, the population number is adjusted again. To maximise the GA performance and short evolutionary time, DBPack GA parameters are set as follows:

1) Population size: 200, building CLBs from scratch.

2) Crossover rate: 0.6

3) Mutation rate: one gene, only one per offspring.

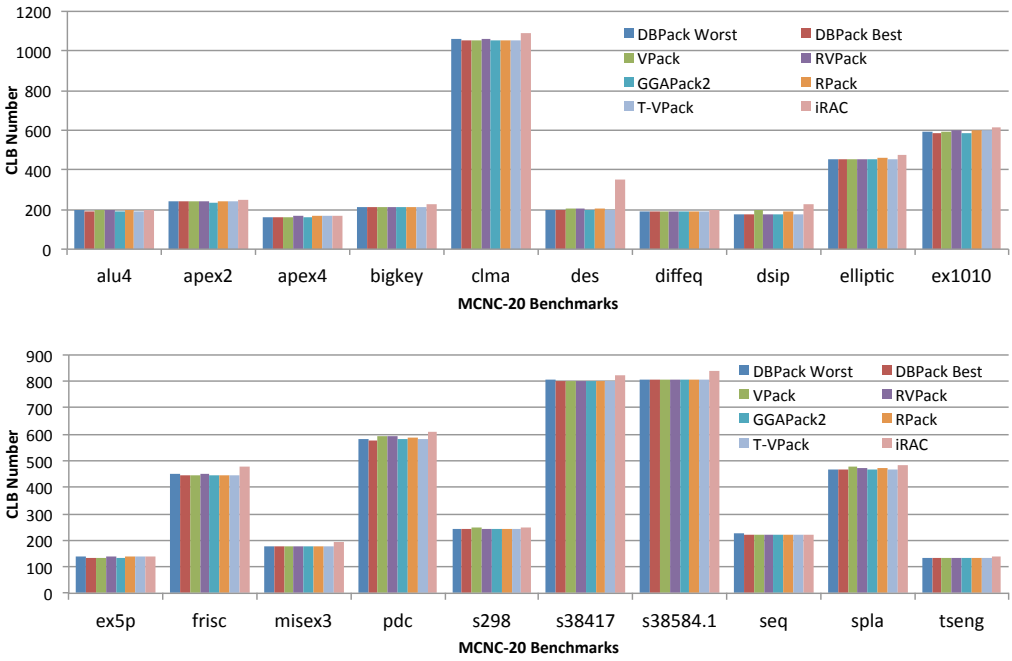


Figure 7.10: DBPack clustered CLB number for MCNC-20 benchmarks compared to VPack, RVPack, GGAPack2, RPack, T-VPack and iRAC, lower is better. Data boxplot and detailed data are provided in Appendices in Figure A.22 and Table A.20.

4) Generation number: 15,000

7.5 Experimental results

7.5.1 DBPack direct outputs

This section shows DBPack clustered results for the MCNC-20 benchmarks, and it also presents how DBPack solution selection works under the MO selection scheme. This is facilitated by a DBPack GA execution which constructs a CLB of the largest "clma" benchmark. For DBPack direct output comparisons, VPack (Betz and Rose, 1997a), RVPack, GGAPack2, RPack (E.Bozorgzadeh et al., 2001; Bozorgzadeh et al., 2004), T-VPack (Marquardt

Table 7.1: Sums of clustered CLB numbers for the MCNC-20 benchmarks between different algorithms – sorted in ascending order, lower is better.

Algorithm	CLB number
GGAPack2	7,459
DBPack Best	7,469
DBPack Average	7,489
T-VPack	7,498
DBPack Worst	7,509
RVPack	7,534
RPack	7,550
VPack	7,551
iRAC	7,971

et al., 1999) and iRAC (Singh and Marek-Sadowska, 2002) clustering results are used as references. Both RVPack and GGAPack2 refer to their best case results. For clustered circuit real-FPGA-mapping tests, VPack, RVPack and T-VPack are used. RPack and iRAC are not involved as no experimental results and source code of these are available. Additionally, GGAPack2 is not compared because its results are poor – worst than VPack. Detailed results and variation box plots are enclosed in Appendices.

CLB usage

The first comparison is for clustered CLB numbers. Figure 7.10 shows the CLB number comparison between different FPGA circuit clustering algorithms for the MCNC-20 benchmarks. The sums of the clustered MCNC-20 benchmark CLB numbers for these algorithms are summarised in Table 7.1. As the table shows, DBPack has fewer number of CLBs for benchmarks if considering the sum, and its CLB usage, in whichever case, is better than most of circuit clustering methods. Only in the worst case, DBPack uses more CLBs than T-VPack. In the best case, the improvements of DBPack to T-VPack, RVPack, RPack, VPack and iRAC are 0.39%, 0.86%, 1.07%, 1.09% and 6.30% respectively. However, GGAPack2 still has the highest CLB usage, although

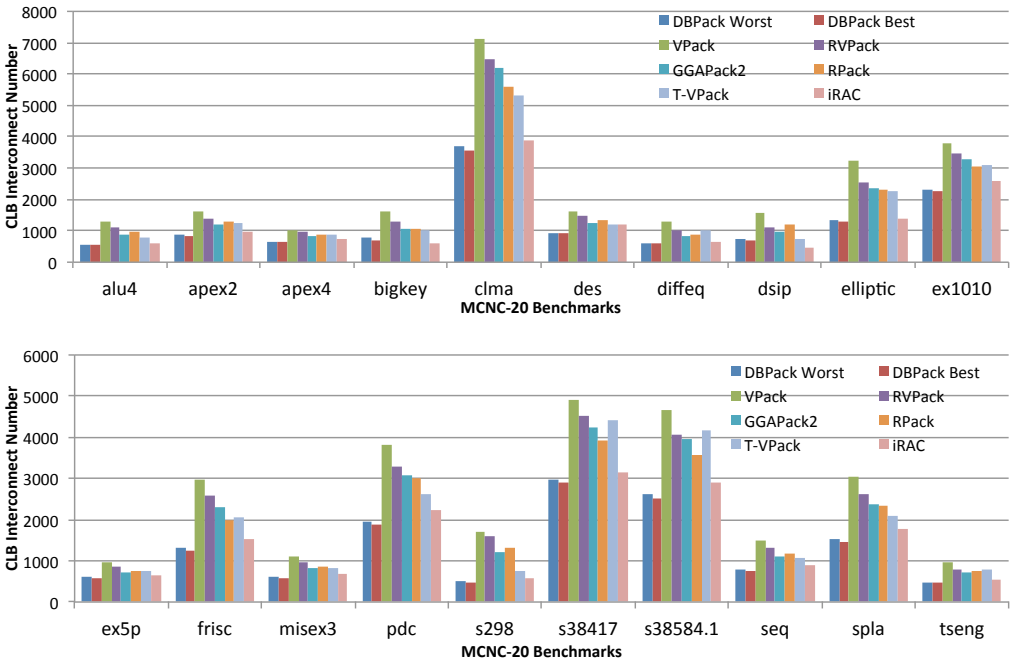


Figure 7.11: DBPack clustered CLB interconnect number for MCNC-20 benchmarks compared to VPack, RVPack, GGAPack2, RPack, T-VPack and iRAC, lower is better. Data boxplot and detailed data are provided in Appendices in Figure A.23 and Table A.21.

the real mapping results are poorer than other algorithms.

Table 7.2: Sums of clustered CLB interconnects and improvements compared to DBPack best case results for MCNC-20 benchmarks between different algorithms – sorted in ascending order, lower is better.

Algorithm	CLB interconnects	Improved
DBPack Best	24,813	0.00% (Reference)
DBPack Average	25,357	2.15%
DBPack Worst	25,901	4.20%
iRAC	28,077	11.62%
T-VPack	37,239	33.37%
RPack	38,300	35.21%
GGAPack2	39,445	37.10%
RVPack	43,349	42.76%
VPack	49,840	50.21%

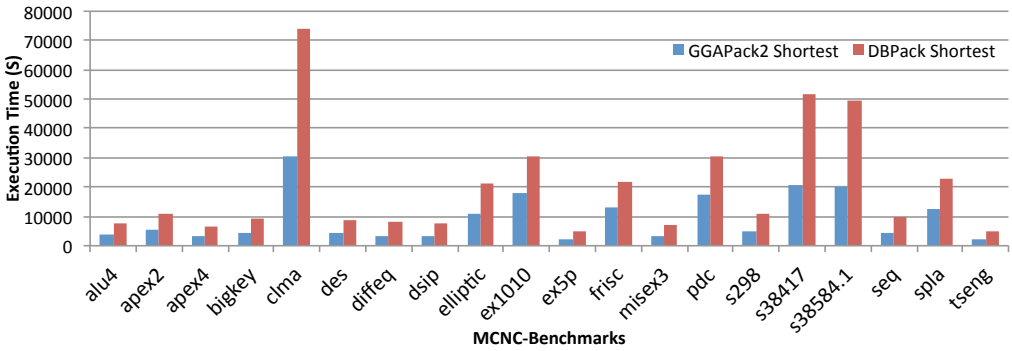


Figure 7.12: DBPack shortest execution time compares to GGAPack2. Data boxplot and detailed data are provided in Appendices in Figure A.24 and Table A.22.

CLB interconnect

Figure 7.11 presents the clustered MCNC-20 benchmark CLB interconnect number comparison for different algorithms. The sums of the clustered MCNC-20 benchmark CLB interconnects and improvements are summarised in Table 7.2. The results indicate that DBPack can include most circuit connections in clustered CLBs, which leads to fewer CLB interconnects after clustering compared with other algorithms. In the best case, DBPack is able to reduce the CLB interconnects by up to 50.21% compared with VPack, and there is also an 11.62% CLB interconnect reduction compared to the outstanding connection-absorption circuit clustering method, iRAC. Moreover, DBPack uses no more CLBs.

Execution time

However, execution time in DBPack is longer than other greedy algorithms, and it is also longer than GGAPack2 – the GA based method. This is a major problem when utilising GAs for performing the clustering. In the execution time comparison, GGAPack2 is used as a baseline. Figure 7.12 shows the shortest execution time for GGAPack2 and DBPack. As can be seen, DBPack

Table 7.3: A clustered CLB in benchmark "clma"

Parameter	Count
# of BLEs	8
# of inputs	14
# of outputs	1
# of clocks	0
# of CLB internal connections	7
# of increased CLB interconnects	4

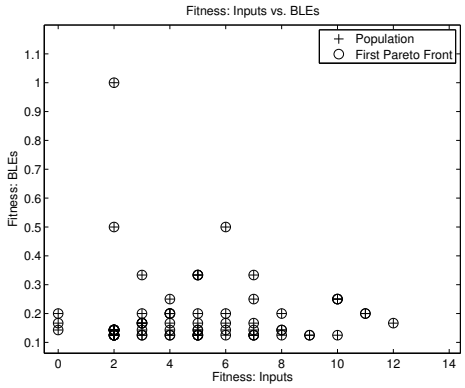
clearly has a longer execution time, and even its shortest clustering time is more than twice that of GGAPack2.

Solution analysis

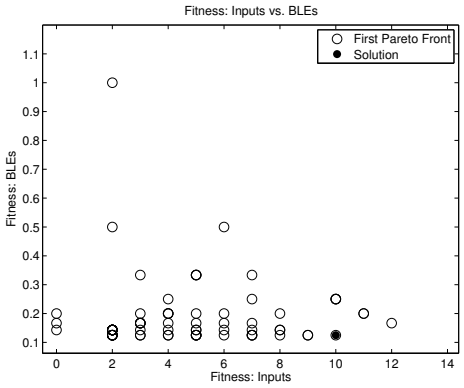
DBPack GA has involved a maximum number of circuit clustering objectives in the construction of a CLB, five objectives, and each GA evolved under the MO selection scheme. As a result, DBPack GA is the best example to show how multiple objectives are incorporated, and how the best individual is selected as a CLB.

To observe this solution selection, the GA's fitnesses are captured. Table 7.3 presents a CLB in the benchmark "clma", and it is constructed by DBPack GA. It is also the best solution, referred to solution selection method, among all solutions – individuals. Figure 7.13 shows six 2-D fitness plots of individuals on the last generation when building this CLB.

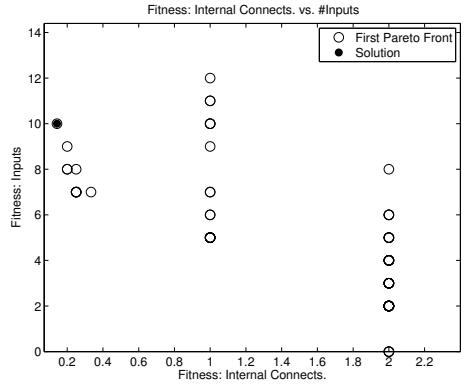
Figure 7.13(a) shows the population fitnesses between the BLE and the input numbers, and also shows the first Pareto front individuals. As these individuals are well evolved, they are all on the first Pareto front. However, these coordinate points are discontinuous, and less than the number of individuals.



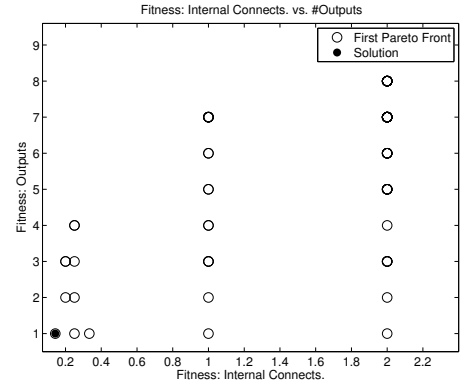
(a) Pop. and 1st Pareto Front at Gen.



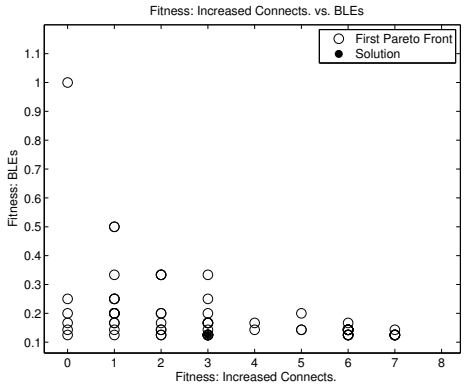
(b) Inputs vs. BLEs at Max. Gen.



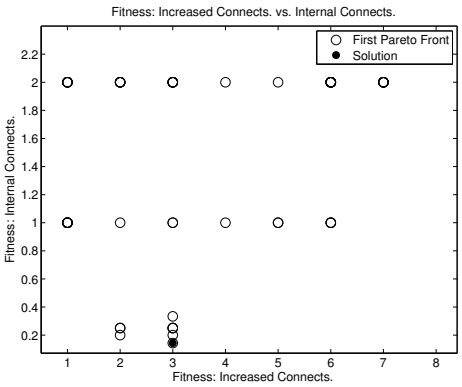
(c) Internal Cons. vs. Input at Max. Gen.



(d) Inter. Cons. vs. Output at Max. Gen.



(e) Inc. Cons. vs. BLEs at Max. Gen.



(f) Inc. Cons. vs. Inter. Cons at Max. Gen.

Figure 7.13: 2-D fitness plots for objectives at the GA last generation, when clustering a CLB in benchmark "clma". Generation number is 15,000 – maximum. The black dot is the selected individual – the determined CLB.

Two factors cause this phenomenon: First, the definition of the fitness functions – DBPack GA fitnesses represent real circuit properties, so the fitness is determined by discontinuous circuit properties. Second, there are duplicated fitness individuals. Although searched BLE combinations are different, the fitness can be the same.

To illustrate DBPack solution selection process, fitness plots, which are shown in Figures 7.13(b)-7.13(f), can be used. Figure 7.13(b) illustrates the individual fitness relationship between the input number and the BLE number. The black dot is the selected individual – the solution, where the fitness value is 0.125 – the smallest fitness value – the best fitness value for this objective, and it has 8 BLE, referred to Equation 7.1 in Section 7.3.3. To clarify why this individual is selected as a solution, consider Figure 7.13(c). This figure indicates the fitness relationship between the CLB internal connections and the input numbers. Similar to Figure 7.13(b), the black dot is the selected individual. Note that all black dots in different figures represent the same individual. As can be seen in Figure 7.13(c), only the black dot individual has the most connections included in the CLB. Under the same principle, in Figure 7.13(d), only the black dot individual represented CLB has fewer output and most included connections. Figures 7.13(e)-7.13(f) help to explain why the individual that increases three CLB interconnects is selected.

7.5.2 DBPack VPR results

This section focuses on clustered circuit real-mapping performance analysis for DBPack.

FPGA area usage

The first comparison is the FPGA area usage. According to VPR mapping reports, when mapping the DBPack clustered MCNC-20 benchmarks in both the best and worst cases, it uses exactly the same FPGA area compared to

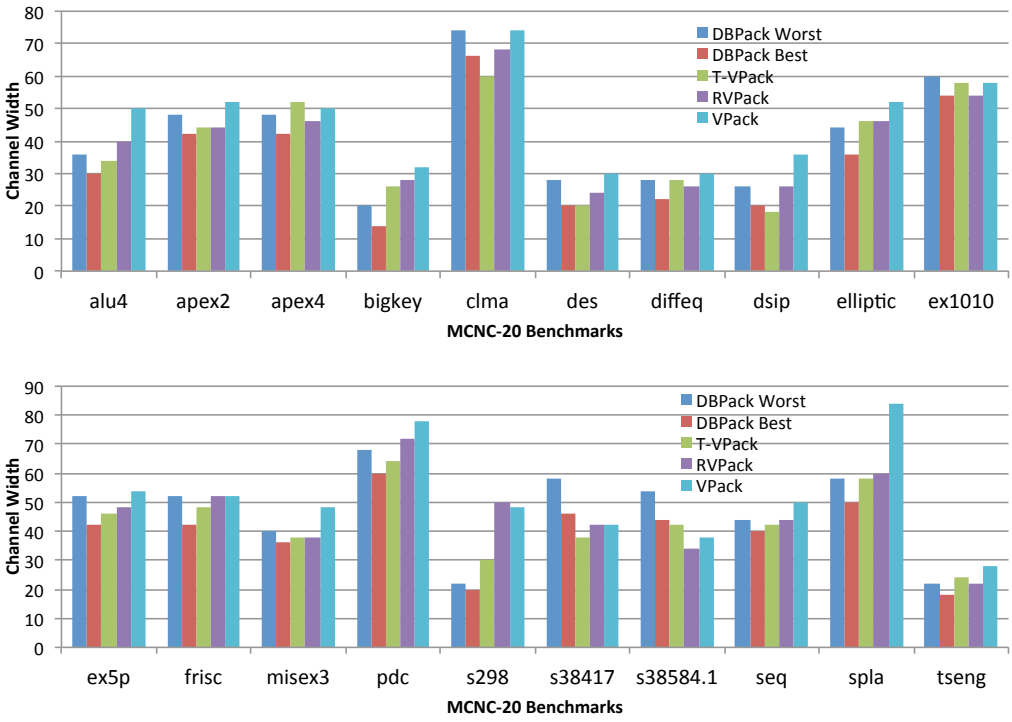


Figure 7.14: DBPack on FPGA channel width for MCNC-20 benchmarks compared to T-VPack RVPack and VPack, lower is better. Data boxplot and detailed data are provided in Appendices in Figure A.26 and Table A.24.

RVPack, RPack and T-VPack. As discussed in the previous Section 2.4.2, although DBPack has fewer CLBs, the actual area usage is not optimised because VPR implements circuit on a $X \times Y$ array, and $X = Y$. If the clustered CLB number is not significantly reduced, the area will not be optimised. Beside this, the other supplementary issue is that VPR has to allocate the input and output (IO) of a clustered circuit. The MCNC-20 benchmarks, for example the benchmark "bigkey", contain many-IO circuits. When mapping these circuits onto the FPGA, no matter how many CLBs are used, the IO pads have to be allocated, and these pads can enlarge the circuit on-FPGA area – the mapping area usage.

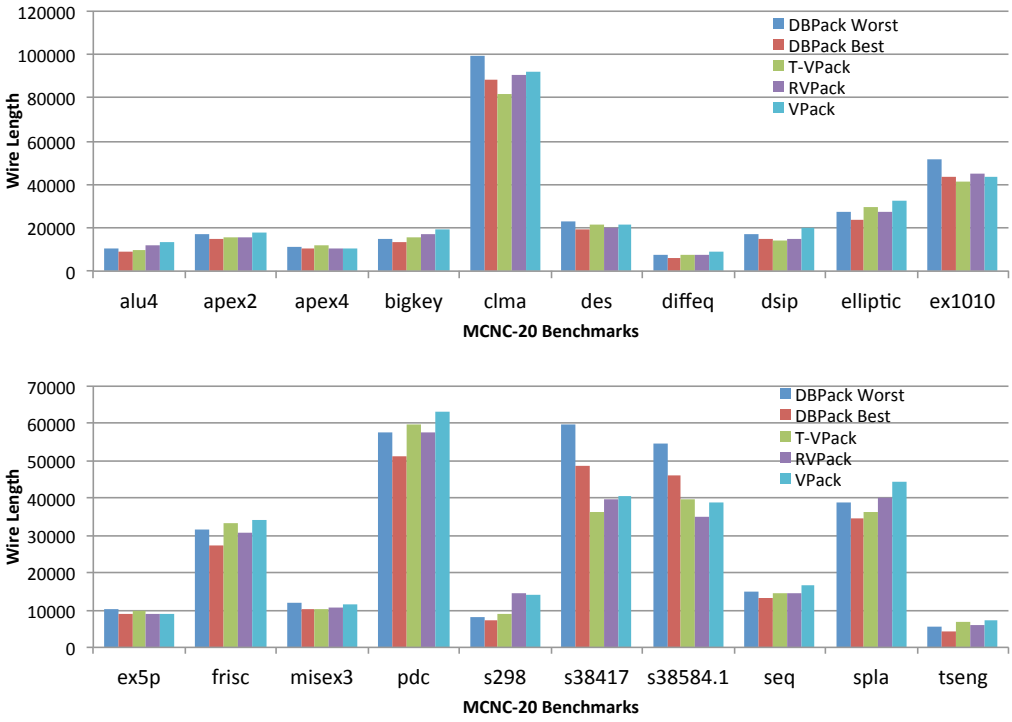


Figure 7.15: DBPack on FPGA wire length for MCNC-20 benchmarks compared to T-VPack RVPack and VPack, lower is better. Data boxplot and detailed data are provided in Appendices in Figure A.27 and Table A.25.

Channel width

As DBPack has reduced the CLB interconnect, when mapping DBPack clustered circuits onto FPGAs, it results in the mapped circuits having narrower channel widths. Figure 7.14 shows the channel width comparison of the clustered MCNC-20 benchmarks between DBPack both cases and main-stream circuit clustering algorithms. The sums of the channel widths are 882, 744, 816, 864 and 986 for DBPack worst and best cases, T-VPack, RVPack and VPack respectively. In both the best or worst cases, DBPack uses less channels than VPack. In the best case, DBPack reduces channels by 24.54% and 8.82% compared with VPack and T-VPack. In contrast to the best case, DBPack occupies 9.68% more channels than T-VPack in its worst case.

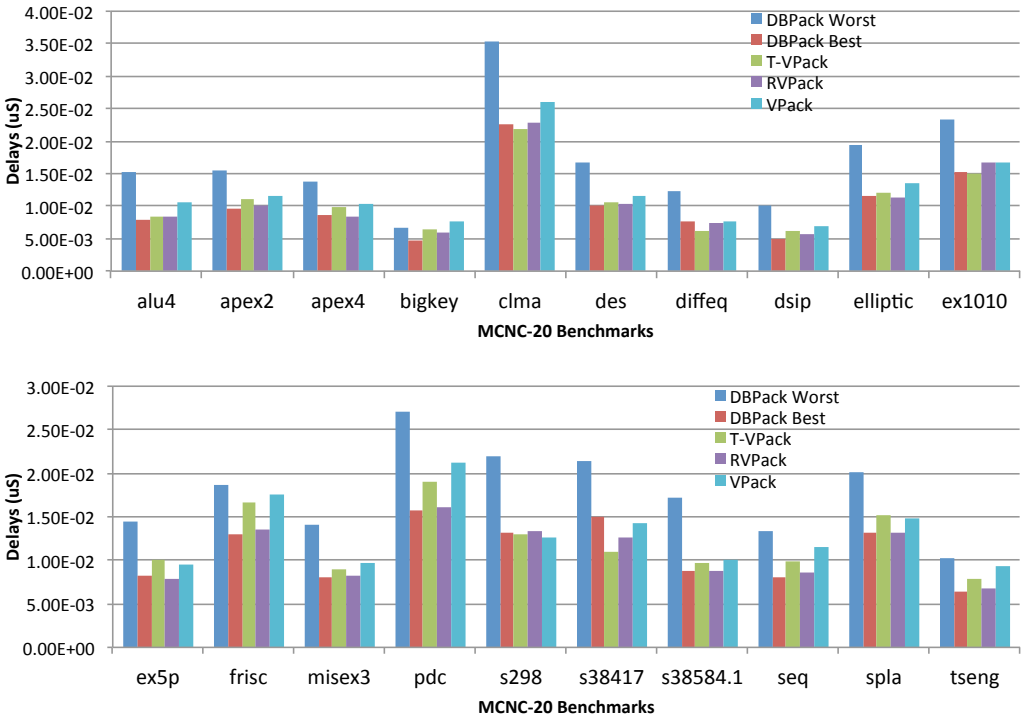


Figure 7.16: DBPack on FPGA circuit-critical-path delay for MCNC-20 benchmarks compared to T-VPack RVPack and VPack, lower is better. Data boxplot and detailed data are provided in Appendices in Figure A.28 and Table A.26.

Wire length

Figure 7.15 illustrates the wire length comparison of the mapped MCNC-20 benchmarks via different circuit clustering algorithms. For large benchmarks, routing DBPack clustered circuits on the FPGA uses more wires than T-VPack, but, for the entire MCNC-20 benchmarks, DBPack wire length is shorter than T-VPack in its best case. The overall wire lengths for DBPack worst case, best case, T-VPack, RVPack and VPack are 572,201, 495,317, 504,230, 518,179 and 558,315 respectively.

Delay

Despite DBPack does not consider the mapped-circuit timing optimisations, the reduced CLB interconnects can actually speed up the mapped circuits. Figure 7.16 is the mapped MCNC-20 benchmark circuit-critical-path delay comparison between different algorithms. DBPack results have better timing performances in the best case. However, DBPack results also have worse timing performances when compared in the worst case. The sums of the clustered and mapped MCNC-20 benchmark circuit-critical-path delays for DBPack worst case, best case, T-VPack, RVPack and VPack are $3.46892 * 10^{-01}$, $2.12634 * 10^{-01}$, $2.28657 * 10^{-01}$ $2.16193 * 10^{-01}$ and $2.53159 * 10^{-01}$ μS respectively.

7.6 Discussion

The DBPack experimental results have been reviewed in the previous section, and notable improvements can be highlighted for DBPack. First of all, DBPack clustered circuits have the minimum CLBs and CLB interconnects. This is better than other algorithms, specially iRAC which was considered as an outstanding connection absorption FPGA circuit clustering algorithm. Benefiting the low CLB and CLB interconnect numbers is the main reason why DBPack clustered circuit mapped onto FPGAs has narrower channels, shorter routing-wire lengths and better timing performances.

In addition, another reason DBPack increases the opportunities of making circuits use fewer tracks, shorter wires and faster speeds is that the DBPack clustering objectives can optimise the common connections for a circuit effectively. On the other hand, the MOGA and multiple fitness functions maintain excellent flexibility for the DBPack. Anything related to the single CLB construction can be represented as fitness functions involved in DBPack.

However, DBPack is not issue free. The obvious issue is the clustering order.

DBPack still uses a bottom-up clustering perspective, which means that the new CLB building is only dependent on the unclustered BLEs. Although there is a global optimisation fitness function – CLB interconnects, the optimisation is limited. Moreover, there is no BLE change or interaction between clustered CLBs and an under clustering CLB. Therefore, this clustering method still limits the quality of the clustered solutions.

One advantage of DBPack is that it deploys a few fitness functions to guide the evolution, but this can also be a negative. First, the fitness functions are designed using the real circuit parameters. This means that the fitnesses are imprecise, and have resolution problems. Another problem with the multiple fitness functions is the longer execution time. It should be emphasised that in the MOGA, the more fitness functions used, the more time is spent in its MO selection, so DBPack uses more evolutionary time than GGAPack2 although the searching space is narrowed. Apart from the fitness functions, the longer execution time is also caused by the discrete GAs and the fixed GA generations. Since the GA settings are based on the biggest MCNC-20 benchmark, this increases the execution time for smaller benchmarks as each GA has to execute a same generation number regardless of when the suitable result is found.

7.7 Summary

This chapter presents the novel DBPack algorithm. It is a GA-based circuit clustering algorithm, and also uses a new bottom-up perspective to cluster circuits for FPGAs. The design and implementation of DBPack solves the major problems in GGAPack, but does not include the clustering perspective. The experimental results show that DBPack can produce better clustering solutions. For the CLB number and CLB interconnect comparisons, DBPack results are outstanding compared with other state-of-the-art algorithms. Unfortunately, DBPack has its own issues – for example, the longer execution time, and the stochastic results due to the use of the GA. However, there

is no doubt that the DBPack provides a new way to use the GA to solve the FPGA clustering problem. In order to facilitate a top-down perspective clustering, the DBPack method can be combined with GGAPack to produce a hybrid circuit clustering method, which will be introduced next chapter.

Chapter 8

HYPack/T-HYPack: Hybrid Circuit Clustering Approach Using MOGAs

8.1 Introduction

In the previous chapters, GGAPack and DBPack were introduced. Both algorithms are based on MOGAs. The obvious difference between these algorithms is the CLB clustering order, where GGAPack clusters a circuit from a global perspective, and DBPack builds CLBs using a new bottom-up perspective. Tests show that GGAPack cannot produce better clustered circuits for optimising real mapping performances, and the major problems are the individual evaluation mechanism (fitness) and the reinsert operation. DBPack builds the CLB by directly searching BLEs, and the fitness functions are effective for a single CLB, so the clustered circuit not only has fewer CLBs and CLB interconnects but also presents better real mapping performances. This chapter introduces the HYPack and T-HYPack circuit clustering methods. HYPack method starts from the CLB number and CLB interconnect number optimisations. The method is then extended to optimising timing of clustered

circuits - this is T-HYPack, which is short for Timing-driven HYPack.

This chapter is organised as follows: Section 8.2 presents the motivation, and explains HYPack and T-HYPack. The implementations of these methods are introduced in Section 8.3. Section 8.4 describes how experiments are created to test HYPack and T-HYPack. The results are compared in Section 8.5, and discussed in Section 8.6. A summary of the chapter is given in Section 8.7.

8.2 Motivation

Previous experiments have shown that the GGAPack2 method, actually referred to as GGAPack, is useful for optimising the CLB number and CLB interconnects. This proves that the designs of GGAPack representation, genetic operations and its fitness functions are feasible for solving the circuit clustering problem – for fewer CLBs and fewer CLB interconnects. However, actual mapping performances are unsatisfactory. This is mainly due to the random reinsertion operation where the freed BLEs are randomly inserted into CLBs only based on hardware constraints, with no common connection considerations.

The HYPack is short for hybrid circuit clustering method, and attempts to solve the circuit clustering problem from a global perspective, similar to GGAPack. The HYPack algorithm is based on GGAPack, but enhances the free BLE reinsertion by incorporating DBPack method – the key word "hybrid" indicates that this algorithm combines two clustering perspectives (methods) together, and also uses DBPack produced solutions as GGAPack GA initial population. Therefore, HYPack contains two clustering phases. Subsequently, the HYPack algorithm has been extended as a timing-driven circuit clustering method, T-HYPack. To fully optimise clustered circuits, T-HYPack involves VPR (Luu et al., 2011; Betz and Rose, 1997b; Betz et al., 1999) in the HYPack evolution process, and uses VPR to emulate real FPGA

mappings. Compared with conventional circuit clustering methods, T-HYPack clustering is no longer an independent procedure, and its implementation can set up interactions between the circuit clustering and the circuit routing of a FPGA which is similar to HDPack (Chen et al., 2007) and Un/DoPack (Tom et al., 2006) clustering methods.

8.3 Implementation

8.3.1 MOGA based hybrid two-phase circuit clustering – HYPack

HYPack is a design to combine the methods of GGAPack and DBPack, and review whether this combination produces better results or not. In HYPack, the circuit clustering is in two phases. HYPack starts optimisations from semi-optimised solutions – the DBPack results, and uses these results as initial population of the GGAPack. This means that HYPack utilises DBPack method as a pre-circuit-clustering tool, which clusters circuits from scratch. This process is called first phase. In this phase, DBPack is executed many times to generate enough stochastic results. Subsequently, a GGAPack-like method is involved as the second phase. To match the second phase GA population size, the first phase results are selected by a MO selector and fitness functions which are the same as in GGAPack. In phase two, only better results are converted to GA individuals. Figure 8.1 presents the working flow of HYPack. As phase one, DBPack, is introduced in Chapter 7, and phase two method is the same as GGAPack which is in Chapter 6, the details of these two methods are not reintroduced. This chapter focuses on how the HYPack method uses DBPack algorithm to reinsert the freed BLEs after the second phase GA genetic operations – referred to GGAPack genetic operations.

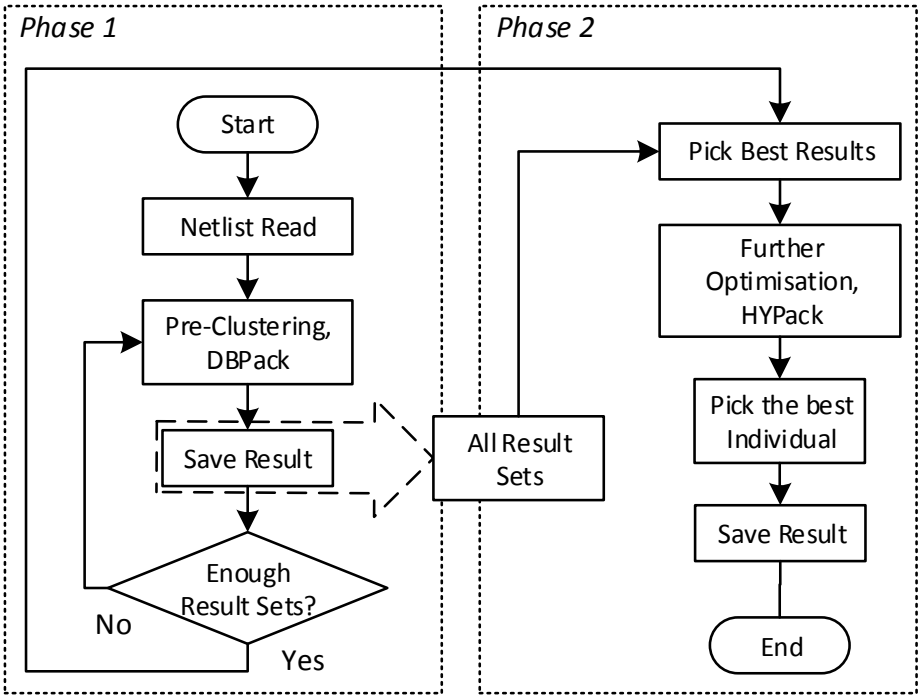


Figure 8.1: HYPack working flow. In phase one, DBPack executes for many times to generate enough stochastic clustering solutions. In phase two, these results are selected and used the GGAPack to further optimise the solution. The free BLE reinsertion process in the GGAPack is replaced and uses the DBPack method, where freed BLEs are represented as a DBPack chromosome, and apply DBPack genetic genetic operations to cluster these BLEs as CLBs.

Figure 8.2 illustrates a situation after-crossover of an individual in the HYPack second phase. Compared with original GGAPack, in which these freed BLEs are directly inserted back into current CLBs. HYPack will reserve the freed BLEs and continue its mutation operation – the same as to GGAPack mutation, which randomly eliminates two CLBs. Figure 8.3 shows an example individual, based on the Figure 8.2, after both crossover and mutation operations. On the right of the figure, these "free BLEs" are the total freed BLEs after two genetic operations. It has to be emphasised that, in real individuals, each individual contains a number of CLBs, so eliminated CLBs are a small proportion to other CLBs. After individuals

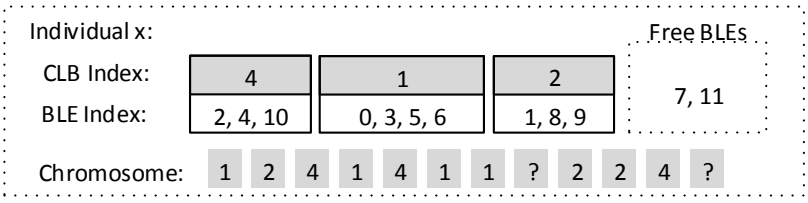


Figure 8.2: An individual after GGAPack crossover operation, some BLEs are freed during this process. Detailed GGAPack crossover operation is introduced in Section 6.3.2. The gene with "?" means the genes represented by this BLEs does not belong to any CLBs.

perform crossover and mutation operations in GGAPack-like GA, these freed BLEs which are reserved in the individual need to be reinserted by using DBPack method. Figure 8.4 shows how these BLEs are converted as a new chromosome for DBPack GAs.

In this reinsertion process, the DBPack method only produces new CLBs with new indexes back to the global-perspective-optimisation GA individuals

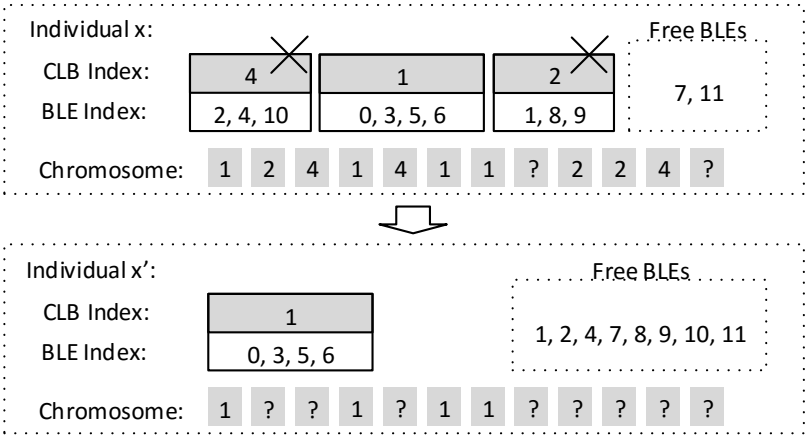


Figure 8.3: An individual after both crossover and mutation operations, where the mutation is designed to randomly eliminate two CLBs. These CLBs are *CLB-2* and *CLB-4*. Note that this figure is based on the Figure 8.2. After these two operations, freed BLEs are reserved in this individual, and waiting for the BLE reinsertion process to cluster them into CLBs.

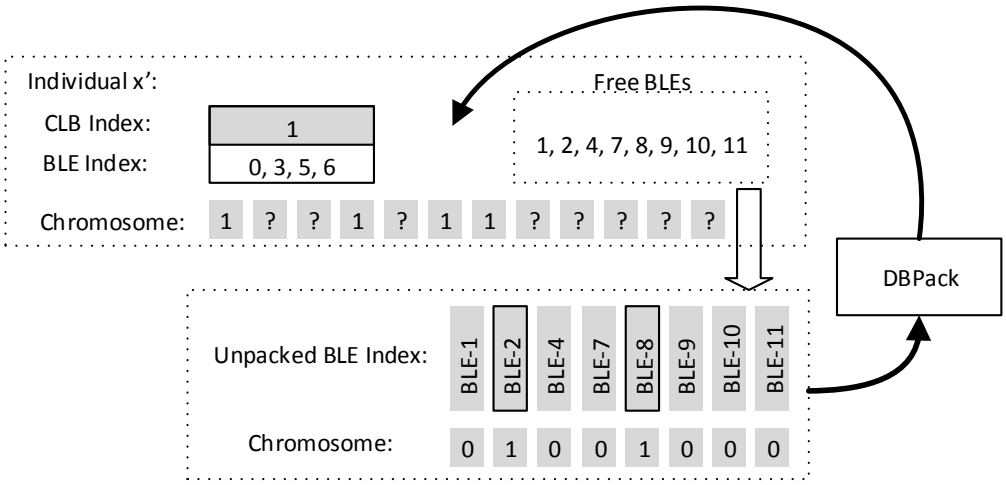


Figure 8.4: Each individual, freed BLEs are reinserted by DBPack method. According to the index of the freed BLEs, a new chromosome is created for DBPack. This chromosome is the longest chromosome, and BLE indexes are based on this chromosome when performing the DBPack to cluster them.

– HYPack phase-two GA individuals. This implies that DBPack method will not reinsert a single BLE into a phase-two-GA individual presented CLBs. In addition, since the reinsertion DBPack deals with a small number of BLEs, so DBPack GA generation number has been reduced to compromise the execution time. The same as GGAPack, in the last generation of the phase-two GA, a best individual will be selected as the final solution, and translated as a netlist.

8.3.2 Timing-driven HYPack – T-HYPack

T-HYPack is an implementation to optimise the timing performance of DBPack clustered circuits, in addition to maintaining other circuit properties such as the CLB number, CLB interconnects and FPGA area usage. In order to optimise the timing of mapped circuit, connections of a clustered circuit should be legitimately arranged. The reason is that the FPGA CLB internal-connection-propagation delays are significantly lower than the connections

between CLBs – CLB interconnects (Marquardt, 1999).

The conventional methods, for example T-VPack (Marquardt et al., 1999), normally cluster target circuits by incorporating the connection criticality. This is achieved by performing a static timing analysis (Hitchcock et al., 1983) before clustering. Subsequently, the circuit clustering method clusters the target circuit, and attempts to arrange the most critical connections inside CLBs while leaving the less critical, or non-critical connections as CLB interconnects. The goal of the method is to avoid unnecessary delays that are produced by certain CLB interconnects.

However, there are many factors that affect the timing of a clustered circuit. First, it is the CLB and CLB interconnect numbers. The fewer CLBs used when mapping the clustered circuit onto a FPGA, the shorter the interconnect wires can be. By the same principle, the smaller the number of CLB interconnects produced, the narrower the routing channels are. Second, is how the CLB interconnects are determined. As different clusterings can form different CLB interconnects, if these connections contain more critical connections, where CLB internal BLEs are badly grouped, the mapped circuit might have larger propagation delays, or the routing wires might be longer – consuming more power. Third, it is the position of the clustering step in the FPGA CAD flow. Since the circuit clustering process is usually regarded as an independent process, the rest of mapping is just using its clustered circuits. This means that the clustered circuit might be satisfiable for the basic circuit clustering requirements – for example CLB and CLB interconnect number, but worse for the real mapping performances.

As DBPack clustered circuits contain many superior timing-performed solutions, this indicates that DBPack clustered circuits can be used or further optimised on timings. T-HYPack is based on HYPack, and an extra step is implemented to perform on-line circuit evaluations. Rather than independently clustering circuits, T-HYPack evaluates and optimises the clustered circuits with VPR, which emulates a real FPGA. When T-HYPack produces a new solution, or receives the solutions from DBPack at the beginning, the solutions

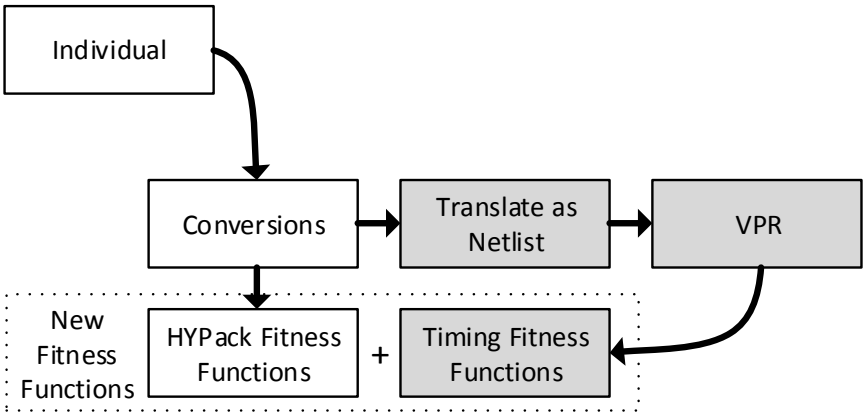


Figure 8.5: T-HYPack involves VPR for evaluating a solution. After the genetic operation and the reinsertion processes, an individual is converted and assigned fitnesses as in HYPack. At the same time, the individual is exported as a netlist to VPR. VPR then is executed, and its outputs are extracted and represented as new fitnesses for this individual. All fitnesses are used in T-HYPack MO selection.

are not only evaluated on the basic circuit clustering requirements, such as the CLB number, CLB interconnects and the CLB internal connection number, but also comparing their mapping performances. Figure 8.5 illustrates how HYPack involves VPR as a real mapping evaluator, and how VPR outputs are composed as fitnesses used in the GA loop.

On the left of the figure is HYPack fitness calculation flow, where an individual is firstly converted as an actual circuit according to the BLE input and output relationships, the method has been shown in Section 6.3.4, and the fitness values are assigned. On the right of the figure, the greyed parts show the VPR flow and the new fitness functions. For each individual, the individual represented circuit, after assigning HYPack fitnesses, is translated as a VPR readable netlist, and VPR emulates a real FPGA mapping. Subsequently, the VPR output is used to compose new fitness criteria. The new fitness functions are represented in Equations 8.1-8.2. Compared with HYPack, T-HYPack has five fitness functions which are shown in Equation 8.1-8.5. Similar to the

GGAPack fitness functions, all these fitness functions to return smaller values when a better solution is found.

$$f_{\text{obj1}}(x) = Critical\ delay\ (Seconds) \tag{8.1}$$

$$f_{\text{obj2}}(x) = Wire\ length \tag{8.2}$$

$$f_{\text{obj3}}(x) = \#\ of\ CLBs \tag{8.3}$$

$$f_{\text{obj4}}(x) = \#\ of\ global\ nets \tag{8.4}$$

$$f_{\text{obj5}}(x) = (\#\ of\ CLB\ absorbed\ nets)^{-1} \tag{8.5}$$

As HYPack and T-HYPack algorithms are similar to GGAPack, Algorithm A.4 in Appendices, HYPack and T-HYPack algorithm pseudocode are not reintroduced. However, the solution picking has to be emphasised; when T-HYPack is executed for a fixed number of generations, on the last generation, the best individual is filtered from the first Pareto front. Unlike HYPack, referred to GGAPack – it is the second-phase method of HYPack, which saves the individual with fewer CLBs and number of CLB interconnects as a solution, T-HYPack selects the individual with the best timing performance and the fewest CLBs. The individual is then converted as a netlist, and regarded as the best solution.

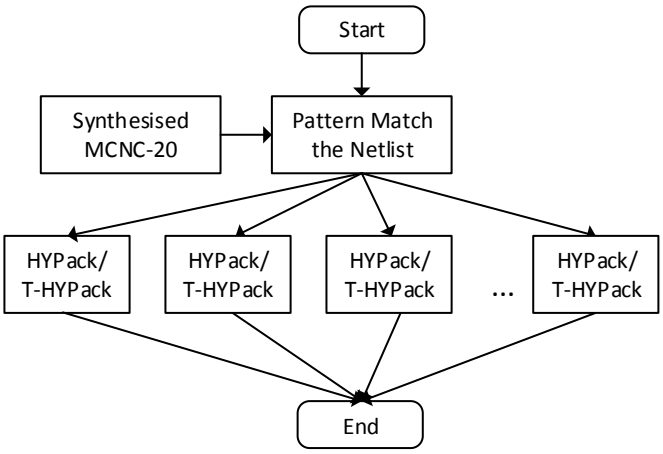


Figure 8.6: HYPack, T-HYPack executing and testing flow

8.4 Experimental setups

The experimental setups are the same as RVPack, GGAPack and DBPack. To test HYPack and T-HYPack, the MCNC-20 benchmarks are used. For each benchmark, the HYPack starts from the synthesised and pattern matched netlist. As HYPack clusters the circuit using both the GGAPack and DBPack methods, this means that HYPack execution time is longer. To get useful results and maintain lower requirements of computing resources, each benchmark in HYPack is managed to execute ten times. In addition, as T-HYPack involves VPR in the evolution where VPR mapping is extremely slow when circuit is large, it significantly increases the entire evolution time, especially for large benchmarks. Therefore, the T-HYPack only uses ten small MCNC-20 benchmarks for testing; "small" refers to a synthesised benchmark that has 1,000-1,500 BLEs. The same as HYPack, ten executions are produced for each benchmark.

The targeted FPGA is the island style FPGA as shown in Section 5.5, which contains a number of CLBs. Each CLB can map 8 4-input-LUT BLEs, and the CLB has 18 inputs and 8 outputs. Like RVPack, HYPack and T-HYPack are executed on the same computing cluster, and their test flows are

shown in Figure 8.6. As T-HYPack has already involved VPR, where the real mapping results can be obtained directly from T-HYPack executions, there is no need to forward T-HYPack results to VPR again. For HYPack, only the CLB number, CLB interconnect number and execution time – HYPack direct outputs, are compared with other circuit clustering algorithms. The results of HYPack are only to show how HYPack meets the basic circuit clustering requirements. For T-HYPack, the result comparisons will cover the FPGA area usage, channel width, wire length and the timing. In contrast to previous circuit clustering method result comparisons, introduced in Chapters 5-7, which only compare one aspect of results at a time, in T-HYPack, the best timing performed solutions are identified, and listed. This helps to show how T-HYPack can solve the circuit clustering problem effectively, and optimise a solution from various aspects – area usage, channel width, wire length and the timing – the MO feature of T-HYPack.

According to GGAPack and DBPack GA settings in Chapters 6-7, the GA parameters of HYPack and T-HYPack are determined. Since HYPack and T-HYPack mainly refer to second level optimisers – the second phase, where both methods start from DBPack solutions and DBPack solutions are already optimised, the generation and population numbers in both GAs are reduced to compensate the execution time. Moreover, the generation and population size reductions are also applied to the reinsertion process – reinsertion DBPack GAs, because the freed BLEs are in a limited number. Although the generation number is small, HYPack and T-HYPack still produce better solutions and the results are shown in Section 8.5. HYPack and T-HYPack GA parameters are summarised as follows:

HYPack GA parameters:

1) Population size: 6

2) Crossover rate: 0.6

3) Generation number: 4,000

T-HYPack GA parameters:

1) Population size: 10
   T-HYPack designs to more benefit from the MO feature.
2) Crossover rate: 0.6
3) Generation number: 1,000

Reinsertion DBPack GA parameters (for both HYPack and T-HYPack):

1) Population size: 200
2) Crossover rate: 0.6
3) Mutation rate: one gene, and only one gene per offspring.
4) Generation number: 2,000

8.5 Experimental results

8.5.1 HYPack direct outputs

This section presents experimental results for HYPack. The CLB number, CLB interconnect number and execution time are compared for HYPack. Reference methods are VPack (Betz and Rose, 1997a), RVPack – best case results, GGAPack2 – best case results, RPack (E.Bozorgzadeh et al., 2001; Bozorgzadeh et al., 2004), T-VPack (Marquardt et al., 1999), iRAC (Singh and Marek-Sadowska, 2002) and DBPack – best case results, as HYPack is competitive to them. Again, to show the variations of GA output, the best and worst case results are compared. The detailed results and variation box plots have been included in Appendices.

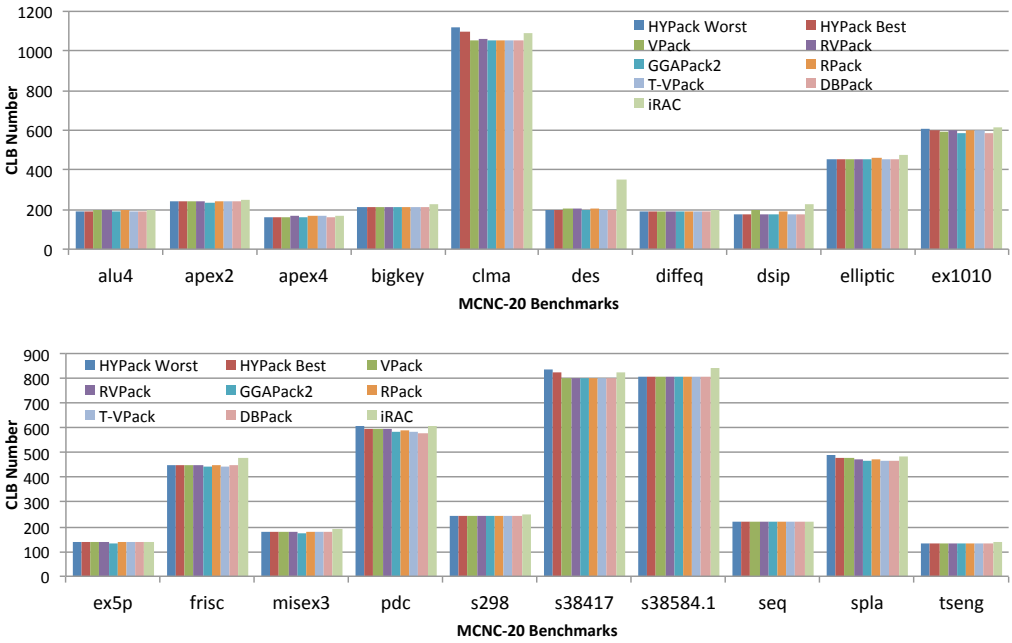


Figure 8.7: HYPack clustered CLB number for MCNC-20 benchmarks compared to VPack, RVPack, GGAPack2, RPack, T-VPack, DBPack and iRAC, lower is better. Data boxplot and detailed data are provided in Appendices in Figure A.29 and Table A.27.

Table 8.1: Sums of clustered CLB numbers for MCNC-20 benchmarks between different algorithms – sorted in ascending order, lower is better.

Algorithm	CLB number
GGAPack2	7,459
DBPack	7,469
T-VPack	7,498
RVPack	7,534
RPack	7,550
VPack	7,551
HYPack Best	7,575
HYPack Average	7,609
HYPack Worst	7,643
iRAC	7,971

CLB usage

The first comparison is for the CLB number comparison; it uses all MCNC-20 benchmarks. Figure 8.7 shows the clustered MCNC-20 benchmark CLB

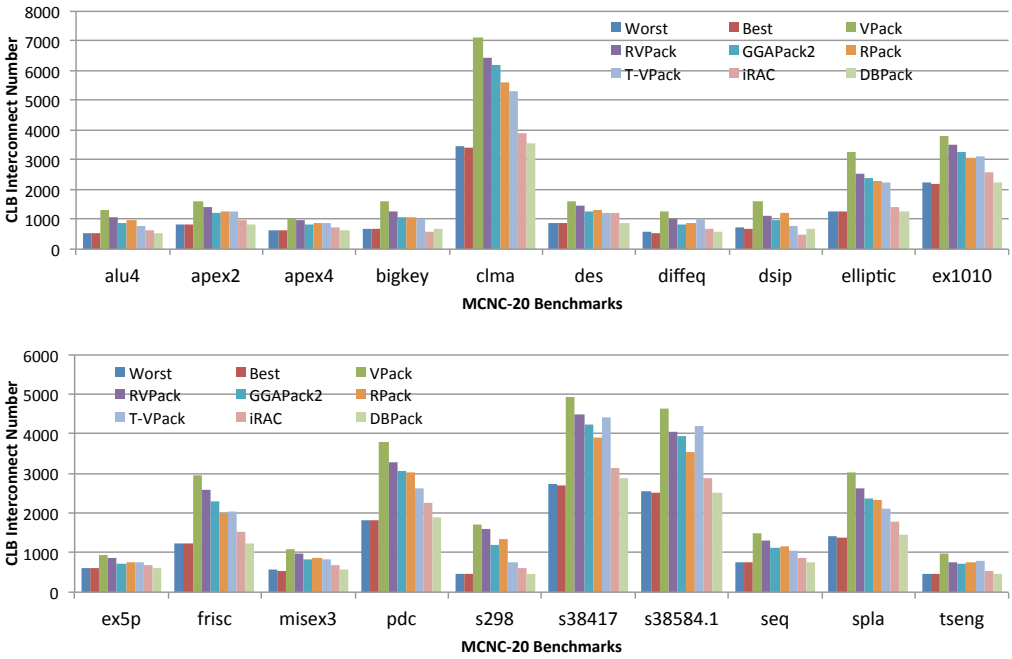


Figure 8.8: HYPack clustered CLB interconnects for MCNC-20 benchmarks compared to VPack, RVPack, GGAPack2, RPack, T-VPack, DBPack and iRAC, lower is better. Data boxplot and detailed data are provided in Appendices in Figure A.30 and Table A.28.

number comparison between HYPack worst and best cases, VPack, RVPack, GGAPack2, RPack, T-VPack, DBPack and iRAC. The figure indicates that HYPack uses no more CLBs apart for some of the large circuits. Table 8.1 summaries the clustered MCNC-20 benchmark CLB numbers between different algorithms. The table shows that HYPack CLB usage is worse than other algorithms when compared with the total CLB numbers. This is due to the DBPack method only deals with freed BLEs by creating new CLBs. However, the CLB usage is still better than iRAC no matter in the worst and best cases. The improvement is by 4.96% compared to iRAC in the HYPack best case.

Table 8.2: Sums of clustered CLB interconnects and improvements compared to HYPack best case results for MCNC-20 benchmarks between different algorithms – sorted in ascending order, lower is better.

Algorithm	CLB interconnects	Improved
HYPack Best	24,074	0.00% (Reference)
HYPack Average	24,244	0.71%
HYPack Worst	24,413	1.39%
DBPack	24,813	2.98%
iRAC	28,077	14.26%
T-VPack	37,239	35.35%
RPack	38,300	37.14%
GGAPack2	39,445	38.97%
RVPack	43,349	44.46%
VPack	49,840	51.70%

CLB interconnect

Figure 8.8 shows the clustered MCNC-20 benchmark CLB interconnect comparison between HYPack worst case and best case, VPack, RVPack, GGAPack2, RPack, T-VPack, iRAC and DBPack. This comparison indicates that HYPack creates the fewest CLB interconnects. The sums of the clustered MCNC-20 benchmark CLB interconnects and improvements of different algorithms are listed in Table 8.2. The data presented in the table indicates that HYPack is the best circuit clustering algorithm for absorbing circuit connections; it is 50.70% better than VPack, and 14.28% better than iRAC.

Execution time

Figure 8.9 compares the shortest execution time of GA-based circuit clustering methods. This includes GGAPack2, DBPack and HYPack. The figure shows that HYPack is the most time-consumed-clustering method. In the comparison of the total shortest execution time, HYPack uses about eight times the execution time of DBPack. Furthermore, another feature is found from the

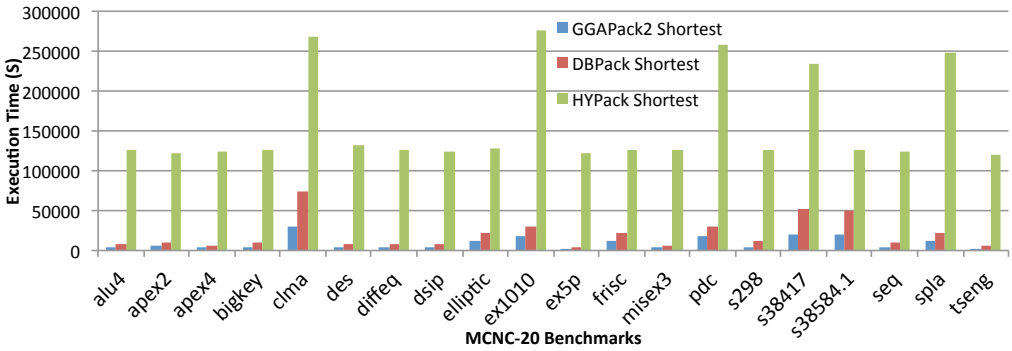


Figure 8.9: The shortest execution time comparison between GGAPack2, DBPack and HYPack for MCNC-20 benchmarks, lower is better. Data boxplot and detailed data are provided in Appendices in Figure A.31 and Table A.29.

Table 8.3: Sums of shortest, average and longest execution time for MCNC-20 benchmarks between GGAPack2, DBPack and HYPack – sorted in ascending order.

Algorithm	Shortest Exe. Time	Avg. Exe. Time	Longest Exe. Time
GGAPack	74,980	97,408	119,836
GGAPack2	187,690	226,138	264,586
DBPack	398,367	689,345	980,322
HYPack	3,157,988	3,445,669	3,733,349

Unit: Second
Exe. = execution
Avg. = average
Shorter time is better.

figure – HYPack has almost identical execution time for small benchmarks. This means that, for these small benchmarks, each GA generation execution time in HYPack is nearly the same – the number of freed BLEs are nearly the same. Rather than only comparing each benchmark shortest execution time, Table 8.3 summarises the MCNC-20 benchmark total execution time for GA based methods. This includes the longest, average and shortest execution time, and lists as a reference.

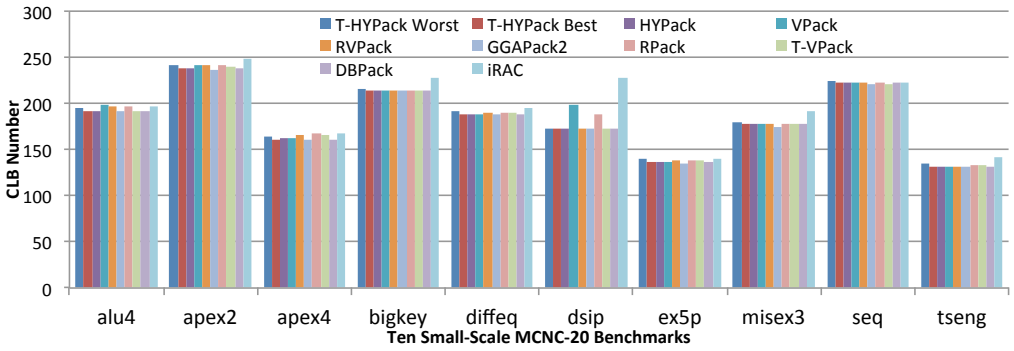


Figure 8.10: T-HYPack clustered CLB number for selected ten MCNC-20 benchmarks compared to HYPack, VPack, RVPack, GGAPack2, RPack, T-VPack, DBPack and iRAC. lower is better. Data boxplot and detailed data are provided in Appendices in Figure A.32 and Table A.30.

8.5.2 T-HYPack outputs

This section presents experimental results for T-HYPack. For the CLB number and CLB interconnects comparisons, VPack, RVPack, GGAPack2, RPack, T-VPack, iRAC, DBPack and HYPack results are used. The used results of RVPack, GGAPack2, DBPack and HYPack refer to their best case results. For the real mapping FPGA area usage, channel width, wire length and delay comparisons are facilitated by VPack, RVPack – best case results, T-VPack and DBPack – best case results. The detailed results and variation box plots have been included in Appendices.

CLB usage

As introduced before, T-HYPack only uses ten of the smaller MCNC-20 benchmarks. Figure 8.10 shows the T-HYPack clustered circuit CLB numbers compared with other algorithms for the ten-small MCNC-20 benchmarks. Table 8.4 lists the sums of CLB numbers of ten selected benchmarks between these algorithms. The table suggests that T-HYPack average results are better than some greedy-algorithm-based clustering methods. In the best

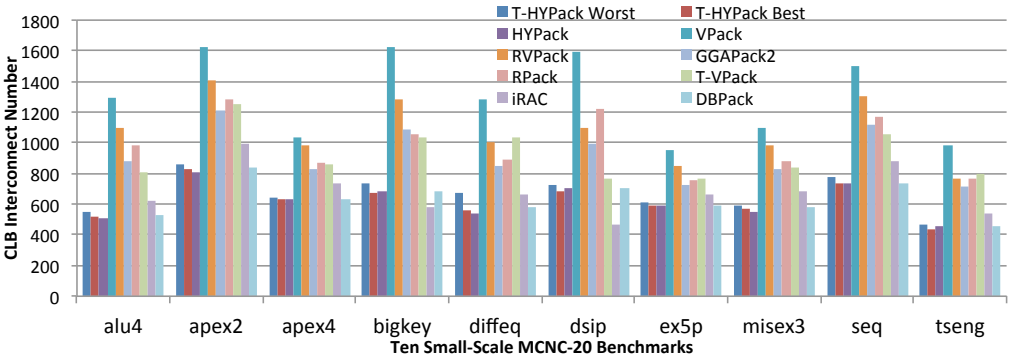


Figure 8.11: T-HYPack clustered CLB interconnect number for selected ten MCNC-20 benchmarks compared to HYPack, VPack, RVPack, GGAPack2, RPack, T-VPack, DBPack and iRAC. lower is better. Data boxplot and detailed data are provided in Appendices in Figure A.33 and Table A.31.

case, T-HYPack results are better than HYPack, DBPack and all greedy algorithm based clustering methods except GGAPack2. However, in the worst case, T-HYPack results is poor compared with recent methods. This table also points out that the result variation of T-HYPack is wider.

Table 8.4: Sums of clustered CLB numbers for ten selected MCNC-20 benchmarks between different algorithms – sorted in ascending order, lower is better.

Algorithm	CLB number
GGAPack2	1,823
T-HYPack Best	1,832
DBPack	1,832
HYPack	1,833
T-VPack	1,843
T-HYPack Average	1,845
RVPack	1,849
T-HYPack Worst	1,858
RPack	1,869
VPack	1,871
iRAC	1,958

Table 8.5: Sums of clustered CLB interconnects for ten selected MCNC-20 benchmarks between different algorithms – sorted in ascending order, lower is better.

Algorithm	CLB interconnects
HYPack	6,215
T-HYPack Best	6,234
DBPack	6,338
T-HYPack Average	6,435
T-HYPack Worst	6,635
iRAC	6,831
T-VPack	9,214
GGAPack2	9,236
RPack	9,875
RVPack	10,777
VPack	12,977

CLB interconnect

Figure 8.11 shows the comparison of T-HYPack clustered ten-smaller MCNC-20 benchmark CLB interconnect number between T-HYPack worst case and best case, HYPack, VPack, RVPack, GGAPack2, iRAC and DBPack. In general, T-HYPack has fewer clustered CLB interconnects. However, for the benchmark "bigkey" and "dsip", DBPack, HYPack and T-HYPack have more CLB interconnects than iRAC. Table 8.5 summarises the total CLB interconnects for the ten smaller MCNC-20 benchmarks among different algorithms. The table indicates that T-HYPack CLB interconnect number in the best case is greater than HYPack. The table also indicates that T-HYPack average and worst case results have more CLB interconnects than DBPack, but the results are better than greedy algorithm based algorithms.

Execution time

T-HYPack involves VPR in the evolution loop, which causes in a longer execution time. Because the working conditions are different, to compare the

Table 8.6: Single execution time comparisons for ten small MCNC-20 benchmarks

	HYPack			T-HYPack		
Benchmk.	Long.	Avg.	Short.	Long.	Avg.	Short.
alu4	148,362	136,782	125,202	556,099	537,515	518,931
apex2	230,977	176,156	121,334	1,334,968	1,267,149	1,199,329
apex4	130,651	127,027	123,403	865,983	831,007	796,030
bigkey	134,457	130,495	126,533	1,509,886	1,250,763	991,640
diffeq	155,831	141,010	126,188	592,586	558,935	525,284
dsip	132,564	127,748	122,932	3,562,884	3,271,641	2,980,398
ex5p	135,992	128,826	121,659	935,335	849,473	763,610
misex3	184,403	154,916	125,429	851,424	805,957	760,490
seq	129,889	126,835	123,780	1,144,545	1,103,508	1,062,471
tseng	133,184	126,852	120,520	377,341	360,632	343,923

Unit: Second
Benchmk. = benchmarks
Long. = longest execution time
Avg. = average execution time
Short. = shortest execution time
Shorter time is better.

Table 8.7: T-HYPack, RVPack, T-VPack, DBPack and the VPack on FPGA area usages, $X \times Y$ arrays, for ten small MCNC-20 benchmarks, lower is better.

Benchmark	T-HYPack	RVPack	T-VPack	DBPack	VPack
alu4	14*14	14*14	14*14	14*14	15*15
apex2	16*16	16*16	16*16	16*16	16*16
apex4	13*13	13*13	13*13	13*13	13*13
bigkey	36*36	36*36	36*36	36*36	36*36
diffeq	14*14	14*14	14*14	14*14	14*14
dsip	36*36	36*36	36*36	36*36	36*36
ex5p	12*12	12*12	12*12	12*12	12*12
misex3	14*14	14*14	14*14	14*14	14*14
seq	15*15	15*15	15*15	15*15	15*15
tseng	15*15	15*15	15*15	15*15	15*15

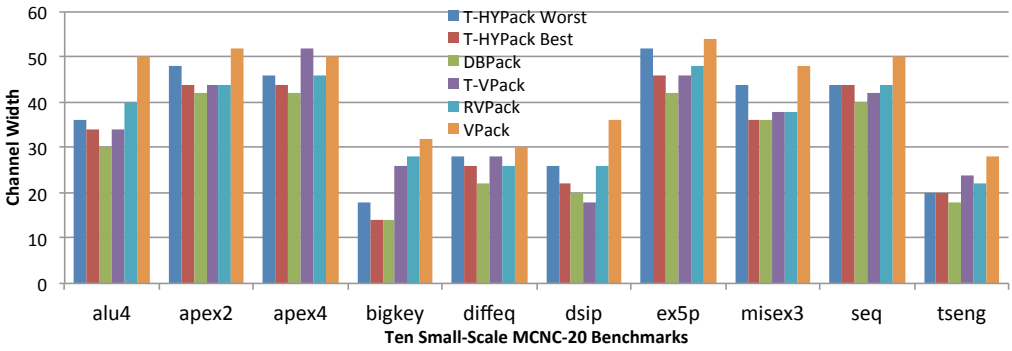


Figure 8.12: T-HYPack on FPGA channel widths for ten small MCNC-20 benchmarks compared to DBPack, T-VPack, RVPack and VPack, lower is better. Data boxplot and detailed data are provided in Appendices in Figure A.36 and Table A.34.

execution time can be meaningless. Therefore, T-HYPack execution time comparison chart is not shown, but each benchmark single execution time is still provided for reference. Table 8.6 shows single HYPack and T-HYPack longest, average and shortest execution time for the ten selected MCNC-20 benchmarks.

FPGA area usage

Table 8.7 shows T-HYPack, RVPack, T-VPack, DBPack and VPack on FPGA area usages for the ten smaller MCNC-20 benchmarks. The results imply that T-HYPack uses the same area as, for example, RVPack, T-VPack and DBPack, but a smaller area usage than VPack, which is in the benchmark "alu4".

Channel width

Figure 8.12 presents the routed FPGA channel width of the ten selected MCNC-20 benchmarks between T-HYPack, DBPack, T-VPack, RVPack and

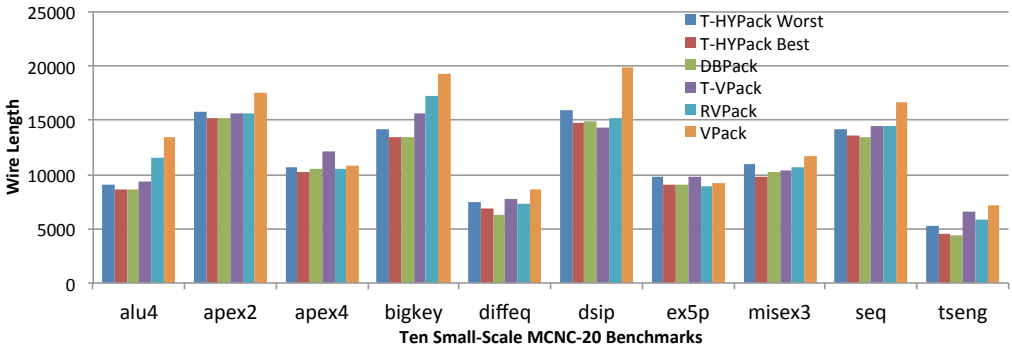


Figure 8.13: T-HYPack on FPGA wire lengths for ten small MCNC-20 benchmarks compared to DBPack, T-VPack, RVPack and VPack, lower is better. Data boxplot and detailed data are provided in Appendices in Figure A.37 and Table A.35.

VPack. The sums of the channels are 362, 330, 306, 352, 362 and 430 for T-HYPack worst case, best case, DBPack, T-VPack, RVPack and VPack respectively. The figure indicates that DBPack has the fewest channels used in its best case compared to other algorithms. Therefore, DBPack can be the algorithm in reducing the mapping channel width, while improving the FPGA routability. In the best case, T-HYPack can reduce channels by 6.25%, 8.84%, 23.26% compared to T-VPack, RVPack and VPack respectively. However, in the worst case, T-HYPack uses 2.76% more channels than T-VPack.

Wire length

Figure 8.13 compares clustered-circuit-routed wire lengths between T-HYPack, DBPack, T-VPack, RVPack and VPack. For the ten selected MCNC-20 benchmarks, the sums of the wire lengths are 113,135, 106,259, 106,048, 116,157, 117,336 and 134,404 respectively for T-HYPack worst case, best case, DBPack, T-VPack, RVPack and VPack. The results suggest that T-HYPack achieves the shortest wires compared to other algorithms in both the worst and best cases. In T-HYPack best case, it reduces the wire lengths by up to 8.51%, 9.44% and 20.94% compared to T-VPack, RVPack and VPack.

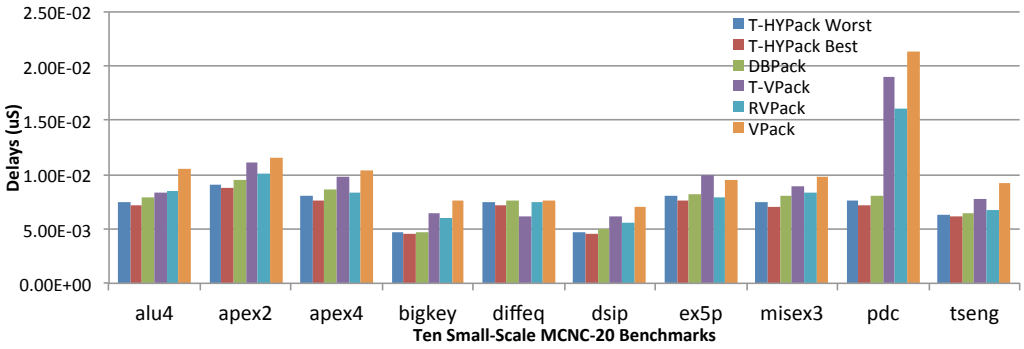


Figure 8.14: T-HYPack on FPGA circuit-critical-path delay for ten small MCNC-20 benchmarks compared to DBPack, T-VPack, RVPack and VPack, lower is better. Data boxplot and detailed data are provided in Appendices in Figure A.38 and Table A.36.

However, T-HYPack uses 0.20% more wire lengths compared to DBPack.

Delay

The next comparison is for the timing of clustered and mapped circuits. Figure 8.14 shows the mapped-circuit critical-path delays of T-HYPack worst case, best case, DBPack, T-VPack, RVPack and VPack for the ten selected MCNC-20 benchmarks. Based on the comparison, T-HYPack has the best timing performance for its clustered circuits. The sums of critical path delays of ten benchmarks for T-HYPack worst case, best case, DBPack, T-VPack, RVPack and VPack are $7.08908 * 10^{-02}$, $6.78532 * 10^{-02}$, $7.41984 * 10^{-02}$, $9.37445 * 10^{-02}$, $8.49561 * 10^{-02}$ and $1.04538 * 10^{-01}$ μS respectively. In the best case, T-HYPack improves the mapped circuit speed by up to 8.55% 27.62%, 20.13%, 35.09% compared to DBPack, T-VPack, RVPack and VPack.

Table 8.8: Best timing performed T-HYPack results compared to T-VPack

Benchmk.	Algo.	CLBs	CLB inter.	CH	Wire-Len.	Delay (nS)
alu4	T-VP.	192	804	34	9410	8.33
	T-HYP.	192	528	34	8864	7.15
apex2	T-VP.	240	1249	44	15681	11.05
	T-HYP.	238	834	44	15824	8.75
apex4	T-VP.	165	863	52	12072	9.74
	T-HYP.	162	645	46	10588	7.60
bigkey	T-VP.	214	1040	26	15619	6.44
	T-HYP.	214	669	14	13640	4.49
diffeq	T-VP.	189	1033	28	7686	6.22
	T-HYP.	188	565	26	6817	7.14
dsip	T-VP.	172	762	18	14368	6.21
	T-HYP.	172	704	22	15157	4.58
ex5p	T-VP.	139	767	46	9780	10.01
	T-HYP.	136	592	46	9618	7.68
misex3	T-VP.	178	840	38	10429	8.93
	T-HYP.	178	579	42	9925	7.08
seq	T-VP.	221	1055	42	14480	8.93
	T-HYP.	225	758	44	13800	7.22
tseng	T-VP.	133	801	24	6632	7.80
	T-HYP.	132	456	20	4545	6.15

Benchmk. = benchmark, Algo. = algorithm
CLB inter. = CLB interconnects, CH = channel widths
Wire-len. = wire lengths
T-VP. = T-VPack
T-HYP. = T-HYPack
Lower is better.

Comprehensive comparison of best timing solutions

Previous chapters only compare each aspect of the clustered circuits. However, this is not able to present all features of a solution, as a clustered circuit might have one aspect superior and other features worse. Table 8.8 lists the best timing performed solutions between T-HYPack and T-VPack. This table shows all features of clustered circuits, and also indicates that T-HYPack

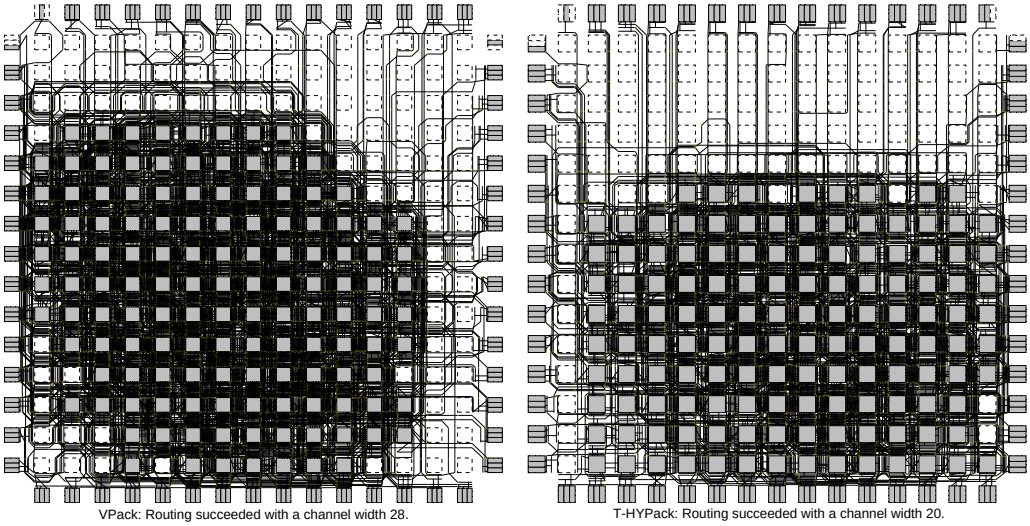


Figure 8.15: Different routings of "tseng" benchmark when using VPack and T-HYPack circuit clustering methods. Routing VPack clustered "tseng" circuit on FPGA uses up to 28 tracks in the routing channel. Routing T-HYPack clustered "tseng" circuit on FPGA uses up to 20 tracks in the routing channel.

will better solve the circuit clustering problem. To clearly show the routing congestion and wire length reductions in T-HYPack, Figure 8.15 presents FPGA routing examples – the smaller benchmark "tseng" final routings on FPGA when using VPack and T-HYPack circuit clustering methods.

8.6 Discussion

Section 8.5 has reviewed HYPack and T-HYPack results. In HYPack, full MCNC-20 benchmarks are used in tests. In T-HYPack, tests are carried out by ten smaller-scale MCNC-20 benchmark circuits. Compared to HYPack, T-HYPack clustered circuits have been further tested for their real FPGA mapping performances.

According to HYPack results, the CLB usage of clustered circuits is low. This means that HYPack has more clustered CLBs for targeted circuits. The

problem is produced by the phase-two GA reinsertion process. In order to speed up the evolution time, DBPack reinsertion only regroups the freed BLEs as new CLBs, rather than reinserting these BLEs back into individual current CLBs. This is why the similar method, the GGAPack, has fewer CLBs. However, HYPack is the best method to absorb circuit connections compared to reference algorithms, which produces a clustered circuit with fewer CLB interconnects.

In the real mappings, T-HYPack, which is based on HYPack, involves VPR as a mapping emulator and uses the Pareto optimality principle (MOGA), so T-HYPack generated results have a number of aspects, such as CLB number, CLB interconnect, channel width, etc., which are better than other algorithms. In not affecting the FPGA mapping area, as well as optimising the routability, T-HYPack has the strongest optimisation on timing of its clustered circuits. Due to T-HYPack incorporating with VPR as an extra plugin, this suggests that this plugin, VPR, can be enhanced or replaced, where anything related to the circuit clustering can be considered without the change of T-HYPack algorithm.

It is notable that GA-based HYPack and T-HYPack take much longer to produce results. It is a major issue, which is also the reason why the larger benchmarks cannot be tested, and has not been addressed in current designs and implementations. On the other hand, one feature in HYPack execution is identified, which is the almost identical execution time for smaller benchmarks. This indicates that the GA-based reinsertion process uses the same time; fixed generation number is used in DBPack-based reinsertion, and the unchangeable time extends entire algorithm execution time. If an adaptive generation number can be implemented to both the BLE reinsertion GA and even in HYPack phase-two GA, the entire evolution time can be reduced. Moreover, the designs of current HYPack, T-HYPack are single-thread programs. If a method can be achieved which parallels these program codes on a process basis to run them on a high-performance computing cluster, solutions can be produced quicker.

8.7 Summary

This chapter introduced HYPack and T-HYPack. These methods are defined as hybrid clustering algorithms, which combine the top-down and bottom-up perspectives when clustering a circuit. The "hybrid" is also apparent in the use of VPR in GA evolution loop. The design motivations are clarified, and detailed implementations are presented. In Section 8.5, test results show that HYPack and T-HYPack have superior clustering performances. However, these tests also point to the drawbacks of these algorithms - for example, the execution time - which have not been solved. In short, the HYPack and T-HYPack provides a useful GA-based methods to solve the circuit clustering method, and the results are better than other existing conventional approaches, for example VPack and T-VPack in terms of CLB usage, CLB interconnects and timing.

Chapter 9

Conclusion and Future Work

This chapter summarises and concludes the key findings of the proposed methods. The research hypothesis is then reviewed, and the conclusion of this research is made. In addition, the proposed methods reveal a set of new opportunities to solving FPGA CAD problems, so the future works are highlighted.

9.1 Key findings

The summaries and conclusions of key findings are based on the proposed methods:

RVPack

1. Greedy algorithm based FPGA circuit clustering methods use a bottom-up clustering perspective, and these methods have to rely on seed selection and cost function in the clustering process. Hence, a highly optimised solution can be difficult to produce.

2. Although seed selection and cost function are improved in new greedy algorithm based clustering methods, the seed and selectable BLEs are not unique. In the worst case, for example in VPack, when dealing with the "clma" benchmark, there are 30.24% BLEs which can be a seed, and an average of 39 BLEs can be selected into a CLB. This is another reason that affects the solution quality of the greedy algorithm based methods.

3. A hill-climbing algorithm is used in most greedy algorithm based methods. The efficiency of hill-climbing is low. The efficiency rate is also unstable when using different seed or BLE selections.

4. RVPack is inspired by EAs, which use stochastic features in seed and BLE selections and also in hill-climbing. Though RVPack is based on VPack, it produces some superior solutions. On the targeted FPGA model, the best RVPack solution can reduce CLB number by up to 0.23%, and CLB interconnect number by up to 13.02% compared with VPack. In VPR tests, RVPack best solution also reduces channel width, wire length and circuit delay by up to 12.37%, 7.19% and 14.60% respectively compared with VPack.

5. RVPack single execution time is similar to VPack, however, it uses a random search. This means that, to find a better solution, RVPack might be required to execute multiple times. In this thesis, 100 RVPack executions are processed, and this is the time-cost.

GGAPack and GGAPack2

1. MO GGA based method, GGAPack, can represent the problem of FPGA circuit clustering, and clustering constraints can be controlled in GGA genetic operations. Due to the use of Pareto optimality, GGAPack and GGAPack2 produced solutions are usually the best trade off. Unfortunately, GGA genetic operations and its MO selection scheme are complex which use more computing time, therefore a larger number

of GA generations cannot be tested with the computing resources available.

2. GGAPack is assumed to have better solutions as it uses a top-down clustering perspective. The solution quality of GGAPack is low which is unexpected. Since it is a top-down GA based clustering method, the huge search space is the major issue that affects the quality of solutions. This can be viewed from its experimental results – the worse solutions are obtained for the larger benchmark.

3. GGAPack2 uses RVPack solutions as initial conditions of the GGAPack GA. A per-selection process is required which uses extra time. On the other hand, RVPack solutions can significantly affect the ultimate solution quality of GGAPack2. GGAPack2 produced solutions are better than RVPack and GGAPack, but the real mapping shows that GGAPack2 solutions perform worse. This is caused by its random reinsertion processes as common circuit connections cannot be reasonably arranged.

4. Both GGAPack and GGAPack2 use three fitness functions (objectives) in their MO section schemes. The resolution, sensitivity to a single CLB, of these functions is relatively low as these functions require the evaluation of an entire solution. The fitness values are also not smooth as they directly use the circuit parameters, for example, CLB number and CLB interconnects. Therefore, designing fitness functions in this way cannot efficiently solve the clustering problem, which a solution usually gets stuck at a local optimum.

5. GGAPack and even GGAPack2 have longer execution times than the greedy algorithm based methods, and GGAPack2 uses about twice the execution time compared with GGAPack. This means if a clustered circuit is close to the optimal, which means a clustered circuit has fewer CLBs and CLB interconnects, these will increase fitness function calculation time – it is caused by the design of GGAPack program. As both methods are based on GAs, their single execution time is

extremely long compared to greedy algorithm and graph-partitioning based methods.

6. Experimental results indicate that GGAPack cannot produce a solution that uses fewer CLBs than VPack and RVPack. However, in the best case, GGAPack solution can reduce CLB interconnects by up 4% compared with VPack – this is worse than the RVPack best case. GGAPack2 is better at reducing the CLB number, where it reduces CLB number by up to 0.79% 0.57% and 0.78% compared with VPack, RVPack best case and RPack respectively. This is better than most of the greedy algorithm based methods. In the best case, GGAPack2 can reduce the CLB interconnects by up to 9.01% compared with RVPack best case, and 20.86% compared with VPack. In the real mapping, GGAPack2 uses the same area compared with RVPack best case solutions. This is slightly better than VPack. Testing results indicate GGAPack2 best case solution reduces 2.23% channel widths compared to VPack, but no improvement on the wire length. GGAPack2 best solution mapped on a FPGA is faster than VPack by 8.97%, but in both cases, GGAPack2 solutions have poor performances on critical path delays compared with RVPack best case solutions.

7. Experimental results are based on 100 executions of GGAPack and GGAPack2 respectively. Result quality can be limited by the number of GA executions.

DBPack

1. To solve the huge search space problem that is identified in GGAPack and GGAPack2, DBPack is proposed. Instead of searching all CLBs at the same time, DBPack focuses on the clustering of a single CLB via MOGAs. In this design, fitnesses are more precise on evaluating one CLB. Although this method still uses a bottom-up clustering perspective, because there is no seed and BLE selections, the quality of

solutions is relatively higher than most greedy-algorithm-based FPGA circuit clustering methods.

2. DBPack GA uses the binary coding representation. This means that standard binary coding genetic operations to be easily incorporated. Experiments show that, in the binary coding representation GA, it is efficient to use the "flipping only one bit" mutation. As the program complexity of genetic operation is simple, the execution time of each GA generation is shorter. However, due to each GA only being able to cluster one CLB, a number of GAs are required for clustering all BLEs in CLBs as well as each GA has to execute a large number of generations. This actually increases execution time – referred to as the completed circuit clustering time.

3. In DBPack, there are five fitness functions (objectives). These fitness functions represent real circuit properties, and use the Rent's rule to optimise solutions. In addition, there is also a fitness function to set up a semi-global perspective optimisation. The constraint controlling is achieved by incorporating penalties in these functions. The penalty level has to be low, otherwise, the population diversity of the GA can be reduced. Experiments indicate that these fitness functions can guide a GA to find useful solutions. Meanwhile, at the end of evolution, there is no individual solution that goes against the constraints. This means that the penalty method can effectively control the clustering constraints.

4. There are still issues – for example, the clustering of the current CLB has no strong interaction to other clustered CLBs; there are a number of best trade off solutions at the end of a CLB clustering but only one is used as a final solution of a CLB. The suitable solutions are rich at the beginning of clustering. However, the number of useful solutions is significantly decreased at end of clustering, as most BLEs are clustered in CLBs. These issues can cause the DBPack to still produce low quality solutions and non-optimal solutions.

5. Experimental results show that DBPack can produce high quality circuit clustering solutions. In the best case, DBPack clustered CLB number is fewer than most greedy algorithms and graph-partitioning based methods, but it is slightly higher than GGAPack2. The DBPack best case solution can reduce CLB interconnects by up to 11.62%, 33.37%, 50.21% compared with iRAC, T-VPack and VPack respectively. In real mapping tests, DBPack solutions use the same FPGA areas compared with RVPack best case, RPack, T-VPack. A best DBPack solution can reduce channel widths by up to 24.54% and 8.82% compared with VPack and T-VPack, and all DBPack solutions have shorter wire lengths compared with VPack and T-VPack. By benefiting from the narrowed channel widths and shorter wire lengths, though DBPack is not intended to speed up a circuit on the FPGA, its produced solutions can also optimise the timing. In the best case, the DBPack solution has better timing performances than T-VPack.

6. These experimental results are based on 100 executions of DBPack. No matter in the best case or worse case, DBPack can produce a solution that has fewer CLBs, and CLB interconnects compared with most circuit clustering methods, such as VPack, T-VPack, RPack, T-RPack, HDPack and iRAC.

HYPack and T-HYPack

1. HYPack and T-HYPack prove that it is possible to combine DBPack and GGAPack methods together as a hybrid FPGA circuit clustering method. The produced solutions are first generated by DBPack, and then optimised by GGAPack method, which uses GGAPack as a second optimiser. The design of HYPack and T-HYPack can be considered as the top-down circuit clustering method. However, the produced solutions are not yet optimal.

2. When combining DBPack and GGAPack methods together, this increases the program complexity. This also means that more computing

resources are required, and a longer execution time is needed. This is the main reason that there are only 10-execution solutions presented in this thesis. The DBPack and GGAPack use different sets of fitness functions. In order to produce better clustered circuits in HYPack or T-HYPack, two sets of fitness functions should be designed properly.

3. T-HYPack presents a feasible approach to incorporate real mappings in a GA based circuit clustering method – this thesis names it as on-line optimisation. The testing results indicate this method is powerful enough to optimise clustering metrics for a solution, and this can be extended to optimise more clustering metrics and objectives, for example power consumption, or the effects of CMOS variability. However, these require accessing the placement and routing processes in each GA generation. This costs a large amount of computing resources. Under the current computing facility, the long execution time results in only 10 small MCNC-20 benchmarks being tested.

4. When HYPack is dealing with MCNC-20 benchmark circuits, for some small benchmarks, their single execution time is similar, and most of the execution time is shared by the DBPack reinsertion process. The reinsertion DBPack GA uses a fixed number of generations – if a circuit is small and is only required to process a small number of freed BLEs, the fixed GA generation number will cost time, but without finding any improved solutions.

5. The best HYPack solutions can reduce clustered CLB interconnects of DBPack by a further 2.98%. In terms of the clustered CLB number, HYPack uses slightly more CLBs, but its solutions are still better than iRAC no matter whether in the worst or best cases. T-HYPack tests use 10 small MCNC-20 benchmarks. In the best case, T-HYPack solution has a similar CLB number compared with GGAPack2, which is smaller than all other methods. The CLB interconnect number is also small, and it is just behind HYPack. In the real mapping, T-HYPack solution uses the same areas compared with other state-of-the-art FPGA circuit

clustering methods. T-HYPack best solutions can reduce channels by 6.25%, 8.84%, 23.26% compared to T-VPack, RVPack best case and VPack respectively, and also reduces the wire lengths by up to 8.51%, 9.44% and 20.94% compared with the previous three methods. As T-HYPack considers timing in the GA loop, the T-HYPack best solution can speed up a mapped circuit by up to 8.55% 27.62%, 20.13%, 35.09% compared with DBPack best case, T-VPack, RVPack best case and VPack respectively. This improvement is huge.

6. These results are only based on 10 executions of HYPack and T-HYPack, so the performance investigation of these methods might be limited.

9.2 Hypothesis and thesis conclusion

In the previous section, the key findings of proposed methods are highlighted. The limitations and issues are also discussed. These limitations and issues cover the method implementations and experiment setups. This section reviews the research hypothesis, and emphasises how these proposed methods tackle the hypothesis and draw the conclusion of this thesis.

The hypothesis of this doctoral research is as follows:

> The quality and performance of a multiobjective circuit mapped to a cluster based FPGA can be improved through the use of evolutionary algorithms during the circuit clustering stage of a FPGA computer aided design flow.

Apart from RVPack, which uses the evolutionary algorithm inspired and greedy algorithm based method to cluster circuits, GGAPack (GGAPack2), DBPack and HYPack (T-HYPack) are all based on fully customised multiobjective genetic algorithms, which belong to the set of evolutionary algorithms. The research first targets clustered circuit quality, which includes the CLB

Table 9.1: Comprehensive comparisons for proposed methods – RVPack, GGAPack2, DBPack and HYPack via full MCNC-20 benchmarks, figures indicate improvements and higher is better.

Method	CLBs	CLB interc.	Areas	CH	W. Len.	Delays	Flex.	Exe. Time
VPack	0%	0%	-	0%	0%	0%	-	+
RPack	0.01%	23.15%	+	N.A.	N.A.	N.A.	-	+
T-VPack	0.70%	25.28%	+	17.24%	9.69%	9.68%	-	+
iRAC	-5.56%	43.67%	N.A.	N.A.	N.A.	N.A.	-	+
RVPack	0.23%	23.15%	+	12.37%	7.19%	14.60%	-	+
GGAPack2	1.22%	21.30%	+	2.23%	-5.78%	8.97%	+	-
DBPack	1.09%	50.21%	+	24.54%	11.28%	16.01%	+	-
HYPack	-0.32%	51.70%	N.A.	N.A.	N.A.	N.A.	+	-

CLB interc. = CLB interconnects
CH = Channel widths
W. Len. = Wire Lengths
Flex. = Flexibility
Exe. = Execution
+ = Positive
- = Negative,
N.A. = Data not available

number, and CLB interconnects. The performance is reflected by the clustered circuit real mappings, for example the circuit delays on a FPGA. The overview comparisons and conclusions of proposed methods and other methods are summarised in Tables 9.1-9.2. Table 9.1 compares the clustered circuits based on the entire MCNC-20 benchmarks, and Table 9.2 compares the clustered circuits based on 10 selected MCNC-20 benchmarks.

According to these tables, it can be concluded that these proposed methods, especially DBPack, HYPack and T-HYPack, can produce a clustered circuit with fewer CLBs and CLB interconnects. In HYPack, the CLB interconnect reduction is by up to 51.70% compared with VPack. The proposed T-HYPack method can even speed up a clustered circuit by up to 35.09% compared with VPack as well.

Table 9.3 is a general comparison, and this comparison covers more well-

Table 9.2: Comprehensive comparisons for proposed methods – RVPack, GGAPack2, DBPack, HYPack and T-HYPack via 10 selected MCNC-20 benchmarks, figures indicate improvements and higher is better.

Method	CLBs	CLB interc.	Areas	CH	W. Len.	Delays	Flex.	Exe. Time
VPack	0.00%	0.00%	-	0.00%	0.00%	0.00%	-	+
RPack	0.11%	23.90%	+	N.A.	N.A.	N.A.	-	+
T-VPack	1.50%	29.00%	+	18.14%	13.58%	10.33%	-	+
iRAC	-4.65%	47.36%	N.A.	N.A.	N.A.	N.A.	-	+
RVPack	1.18%	16.95%	+	15.81%	12.70%	18.73%	-	+
GGAPack2	2.57%	28.83%	+	6.98%	2.59%	15.20%	+	-
DBPack	2.08%	51.16%	+	28.84%	21.10%	29.02%	+	-
HYPack	2.03%	52.11%	N.A.	N.A.	N.A.	N.A.	+	-
T-HYPack	2.08%	51.96%	+	23.26%	20.94%	35.09%	+	-

CLB interc. = CLB interconnects
CH = Channel widths
W. Len. = Wire Lengths
Flex. = Flexibility
Exe. = Execution
+ = Positive
- = Negative
N.A. = Data not available

known circuit clustering methods from 1999 to 2014 – first timing-drive FPGA circuit clustering method, T-VPack, was published in 1999. Note that these methods do not appear in result analysis parts in this thesis as these methods real mapping tests are slightly different from this thesis and their source codes are not available. However, these methods are all compared with T-VPack, and this data is collected from their literatures (Marquardt et al., 1999; Singh and Marek-Sadowska, 2002; Bozorgzadeh et al., 2004; Chen et al., 2007; Rajavel and Akoglu, 2011; Feng, 2012; Feng et al., 2014b).

Based on Table 9.3, it shows that DBPack and T-HYPack are better at including circuit connections in clustered CLBs. The circuit speed improvements are 20.85% and 27.62% in DBPack and HYPack respectively. The improvements are outstanding and beyond all other timing-driven FPGA circuit clustering methods. Currently, DBPack and T-HYPack are the best

Table 9.3: A general comparison for well-known FPGA circuit clustering methods, figures indicate improvements and higher is better.

Method	CLBs	CLB interc.	Channel widths	Wire lengths	Delays
T-VPack	0.00%	0.00%	0.00%	0.00%	0.00%
T-RPack	N.A.	7.01%	2.66%	N.A.	5.00%
iRAC	-6.24%	25.86%	16.10%	25.00%	-4.35%
DPack	N.A.	9.20%	N.A.	17.70%	7.80%
HDPack	N.A.	12.70%	N.A.	23.20%	6.10%
MO-Pack	N.A.	10.73%	11.44%	12.60%	-1.44%
PPack*	N.A.	N.A.	19.80%	17.20%	-4.30%
T-PPack*	N.A.	N.A.	17.00%	15.10%	3.60%
DBPack	0.59%	31.21%	13.00%	8.70%	20.85%
T-HYPack	0.54%	32.34%	6.25%	8.52%	27.62%

CLB interc. = CLB interconnects
N.A. = Data not available or not compatible
* = PPack and T-PPack cluster circuits for input-bandwidth-free CLB FPGAs.

methods to improve the performance of a clustered circuit based on MCNC-20 benchmarks. However, the channel width and wire length are not the best as T-HYPack only considers delays in its MO schemes. If channel width and wire length are involved and represented as clustering objectives, better results could be obtained.

By reviewing previous tables, it can be concluded that evolutionary algorithm based circuit clustering methods can optimise clustered circuits in FPGA CAD flows. Although these proposed methods have to execute for a longer time, the method flexibilities and solution qualities are outstanding - even beyond most of the well-known and state-of-the-art methods.

9.3 Future work

This section highlights the future works.

1. This thesis used VPack as an example and proposed RVPack FPGA circuit clustering method. The results are better than the original VPack. As all greedy algorithm based methods can be affected by the multiple seed and absorbable BLE issues, the solution can be improved. iRAC has huge improvements on VPack; therefore, if randomnesses are injected to iRAC, even better solutions could be produced by iRAC. The implementation can be similar to RVPack, which uses a random generator to select available BLEs, hence a RiRAC (Random iRAC) might be worth developing.

2. This thesis introduced a set of MOGA based FPGA circuit clustering methods, such as GGAPack, DBPack and HYPack. Each method has to be dependent on precise GA parameters in order to produce high quality solutions. However, these GA parameters in this thesis might not be appropriately tuned as there is a computing resource limitation. These parameters can be either adjusted by hand or using another program. Once these parameters are best tuned, better results might be obtained. Therefore, an automatic GA tuning algorithm is worth developing.

3. The proposed methods are all multiobjective, and benefiting from the "model free" feature of GAs. This implies that these methods can incorporate a number of clustering objectives. As electronics nowadays tend to require low power and use deep-nano technologies, power consumption and CMOS variability can be considered in a FPGA circuit clustering method. These optimisations can help a circuit to further reduce the requirement of power, and increase the tolerance to CMOS variabilities. In the short term, these proposed methods are worth having the ability of optimising the channel width and routing wire

length. This can be facilitated by extracting the information in VPR, and adding these key information in GA MO schemes.

4. EAs in general solve problems by using artificial evolution, so there is no model. The results show that GA based methods have excellent solutions for the targeted problem. Once these solutions are obtained, an investigation can be set up to review how BLEs are arranged in CLBs, for example in the FPGA circuit clustering process. This might be able to illustrate a model, where it shows which types of BLEs are grouped. Based on the model, similar to RVPack, GA inspired greedy algorithm based circuit clustering method might be possible to design, and this can significantly reduce the execution time when dealing with a large circuit.

5. GA based methods use population models, and this population represents a larger number of solutions – individuals. When the GA is evolving, these individuals are processed sequentially. This means that GA can be easily parallelised. If these proposed methods are parallelised and executed on a larger computing cluster with a larger number of processors and memories, the single execution time of a GA based circuit clustering method can be significantly reduced. Therefore, parallelised DBPack and HYPack (T-HYPack) are worth developing. Especially in T-HYPack, if these time costly VPR executions are parallelised, large population and generation numbers can be tested in T-HYPack so better results can be obtained.

6. A CAD flow contains a set of algorithms. Especially in post-synthesis process, these clustering, placement and routing can be treated as a set of optimisation problems. These processes either use random search or heuristic algorithms, for example greedy algorithm. As EAs are powerful for solving these complex problems, EA based CAD flow can be produced, or used as a second optimiser. Conventional methods produce a quick solution, and then EAs are used for further optimising the solution. As introduced before, EAs or GAs use population model.

On the other hand, EA based FPGA CAD flow can produce a set of circuit mapping solutions. Some solutions might have fast speeds, and others might save more power. These mappings can be stored, and mapped to a FPGA reconfigurable fabric depending on the requirement of applications. A performance or power-saving adaptive system can therefore be produced.

Appendices

Appendices present supplementary graphs, tables, algorithms and experiment results for main chapters.

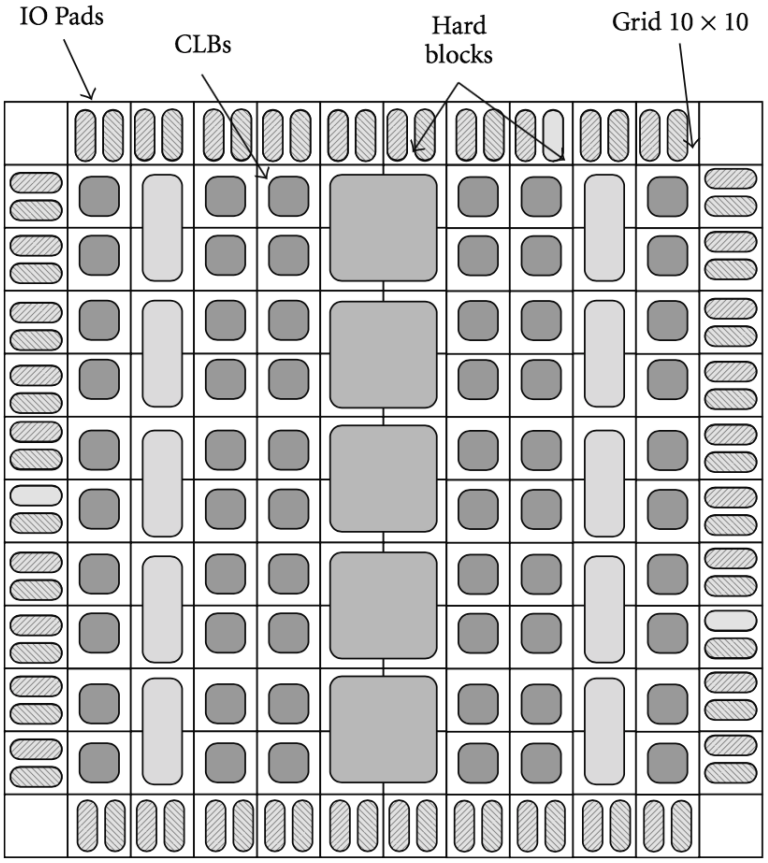


Figure A.1: An example of the heterogeneous FPGA structure (Farooq et al., 2011)

Table A.1: Synthesised MCNC-20 benchmarks

Benchmark	Func.	Inputs	Outputs	LUTs	FFs	Nets
alu4	ALU	14	8	1522	0	1536
apex2	Misc. Func.	39	3	1878	0	1917
apex4	Misc. Func.	9	19	1262	0	1271
bigkey	Key Encription	263	197	1707	224	2194
clma	Bus Interface	383	82	8381	33	8797
des	Data Encription	256	245	1591	0	1847
diffeq	Application	64	39	1494	377	1935
dsip	Encription	229	197	1370	224	1823
elliptic	Application	131	114	3602	1122	4855
ex1010	Misc. Func.	10	10	4598	0	4608
ex5p	FSM	8	63	1064	0	1072
frisc	CPU	20	116	3539	886	4445
misex3	Misc. Func.	14	14	1397	0	1411
pdc	Misc. Func.	16	40	4575	0	4591
s298	Logic	4	6	1930	8	1942
s38417	Logic	29	106	6096	1463	7588
s38584.1	Logic	39	304	6281	1260	7580
seq	Arithmetic Func.	41	35	1750	0	1791
spla	Logic	16	46	3690	0	3706
tseng	Application	52	122	1046	385	1483

Func. = Function
Misc. = Miscellaneous
FSM = Finite-State Machine

Table A.2: Synthesised MCNC-20 benchmarks after the pattern match

Benchmark	Inputs	Outputs	BLEs	Nets	P2P Cons.
alu4	14	8	1522	1536	5408
apex2	39	3	1878	1916	6692
apex4	9	19	1262	1271	4497
bigkey	263	197	1707	1936	6313
clma	383	82	8383	8445	30462
des	256	245	1591	1847	6110
diffeq	64	39	1497	1561	5296
dsip	229	197	1370	1599	5645
elliptic	131	114	3604	3735	12634
ex1010	10	10	4598	4608	16078
ex5p	8	63	1064	1072	4002
frisc	20	116	3556	3576	12772
misex3	14	14	1397	1411	4968
pdc	16	40	4575	4591	17139
s298	4	6	1931	1935	6951
s38417	29	106	6406	6435	21344
s38584.1	39	304	6447	6485	20840
seq	41	35	1750	1791	6193
spla	16	46	3690	3706	13808
tseng	52	122	1047	1099	3760

P2P = Point-to-Point
Cons. = Connections

Algorithm A.1 Pseudocode for VPack

```
Let: UnclusteredBLEs be the number of BLEs that are not in any CLB
     C be the set of BLEs that are in the present CLB
     CLBs be the clustered CLBs, each CLB contains a set of BLEs
Input: Netlist of LUTs and FFs (Registers)
       N = CLB Size
       I = Input number per CLB
Output: clustered CLBs
UnclusteredBLEs = PatternMatchToBLEs(LUTs, FFs); /*Pair LUTs,
FFs as BLEs*/
CLBs = ∅;
while (UnclusteredBLEs != ∅) { /*More BLEs to cluter*/
    C = GetBLEwithLargestInputs(UnclusteredBLEs); /*Seed selec-
tion*/
    while (|C| < N && |Inputs of C| < I) {
        SelectedBLE = MaxGainLegalBLE(C, UnclusteredBLEs);
        if (SelectedBLE == ∅)
            break;
        else {
            if (Gain of SelectedBLE == 0 && |CLBs| != 0) {
                if (hill-climbing(CLBs, SelectedBLE) NOT succeed)
                    C = C ∪ SelectedBLE;
            } else
                C = C ∪ SelectedBLE;
            UnclusteredBLEs = UnclusteredBLEs - SelectedBLE;
        }
    }
    CLBs = CLBs ∪ C;
}
SaveNetlist(CLBs);
```

Algorithm A.2 Pseudocode for fast-non-dominated sort

Let: $\boldsymbol{P}$ be the set that needs to be sorted.
for(each $p \in \boldsymbol{P}$){
    $S_p = \emptyset$; $n_p = 0$;
    for(each $q \in \boldsymbol{P}$){
        if($p \prec q$){ /*If p dominates q*/
            $S_p \cup \{q\}$; /*Adding q to the set of solutions dominated by p*/
        }else if($q \prec p$){
            $n_p = n_p + 1$; /*Incrementing the domination counter of p*/
        }
    }
    if($n_p == 0$){ /*p belongs to the first Pareto front*/
        $P_{rank} = 1$;
        $F_1 \cup \{p\}$;
    }
}
$i = 1$; /*Initialising the Pareto front counter*/
while($F_i \; != \emptyset$){
    $Q = \emptyset$; /*Used for storing the numbers of the next Pareto front*/
    for(each $p \in F_i$){
        for(each $q \in S_p$){
            $n_q = n_q - 1$;
            if($n_q == 0$){ /*q belongs to the next front*/
                $q_{rank} = i + 1$;
                $Q = Q \cup \{q\}$;
            }
        }
    }
    $i = i + 1$;
    $F_i = Q$;
}

Algorithm A.3 Pseudocode for crowding-distance assignment

Let: $\boldsymbol{L}$ be the set of solutions on the same Pareto front.
$l = |\boldsymbol{L}|$;
for($i = 0;\ i < l;\ i++$){
    $L[i]_{distance} = 0;$ /*Initialising distance*/
}
for(each objective m){
    $L = sort(L, m);$ /*Sort using each objective value (fitness)*/
    $L[0]_{distance} = L[l-1]_{distance} = \infty;$ /*Boundary points are always selected*/
    for($i = 1;\ i < l-1;\ i++$) { /*Other points*/
        $L[i]_{distance} = L[i]_{distance} + (L[i+1].m - L[i-1].m)/(f_m^{max} - f_m^{min});$
    }
}

Algorithm A.4 Pseudocode for GGAPack

Let: ***P*** be a storage for population individuals.
P' be a temporary storage for individuals, or mating pool. /* 2****P*** size */
S be a solution individual.
CLBs be the clustered CLBs, each CLB contains a set of BLEs
Input: ***UnclusteredBLEs*** = Pattern matched netlist from LUTs and FFs (Registers)
G = Maximum generation number
PopSize = Population size
Output: clustered CLBs

P' = GenerateInitialPopulation(***PopSize*2***, ***UnclusteredBLEs***);

$i = 0$;
while ($i <$ ***G***) { /*Evolving loop*/
$i = i + 1$;
P' = FitnessCalculation(***P'***); /*Evaluating individuals*/
P = MultiobjectiveSelect(***P'***); /*Fast-non-dominated sort and crowding distance*/
P' = ***P*** + GeneticOperations(***P***); /*Genetic operations, crossover and mutation*/
}

S = PickBestIndividual(***P'***); /*First Pareto front, and fewer CLBs, interconnects*/
CLBs = TranslateToCLBs(***S***);
SaveNetlist(***CLBs***);

Algorithm A.5 Pseudocode for DBPack solution picking process

Let: $\boldsymbol{P}_s$ be the set of last generation population.
$\boldsymbol{T}$ be the maximum BLE solution individuals.
$\boldsymbol{N}$ = CLB Size
$\boldsymbol{I}$ = Input number per CLB
Output: $\boldsymbol{F}_t$, the solution individual.

$\boldsymbol{T} = \emptyset$;
$n = \boldsymbol{N}$;

while($\boldsymbol{T}$ is $\emptyset$){
    for(each $p \in \boldsymbol{P}_s$){
        if(p on 1st Parto front && BLE(p) == n && Input(p) <= $\boldsymbol{I}$){
            $\boldsymbol{T} = \boldsymbol{T} \cup p$;
        }
    }
    if($\boldsymbol{T}$== $\emptyset$){
        $n = n - 1$; /*Reducing one BLE, collect suitable individuals again.*/
    }else{
        break;
    {
}

$\boldsymbol{F}_t$ =MostInternalConnect($\boldsymbol{T}$); /*Return the one that has the most internal connections*/

Algorithm A.6 Pseudocode for DBPack

Let: $\boldsymbol{P}$ be a storage for population individuals.
$\boldsymbol{P'}$ be a temporary storage for individuals, or mating pool. /* 2*$\boldsymbol{P}$ size */
$\boldsymbol{S}$ be the set of BLEs from a solution individual.
$\boldsymbol{CLBs}$ be the clustered CLBs, each CLB contains a set of BLEs
Input: $\boldsymbol{UnclusteredBLEs}$ = Pattern matched netlist from LUTs and FFs (Registers)
$\boldsymbol{G}$ = Maximum generation number
$\boldsymbol{PopSize}$ = Population size
Output: clustered CLBs

$\boldsymbol{CLBs} = \emptyset$;

while ($\boldsymbol{UnclusteredBLEs} != \emptyset$){
    $\boldsymbol{P'}$ = GenerateInitialPopulation($\boldsymbol{PopSize}$, $\boldsymbol{UnclusteredBLEs}$);
    $\boldsymbol{P'}$ = FitnessCalculation($\boldsymbol{P'}$); /*Evaluating individuals*/
    $\boldsymbol{P}$ = MultiobjectiveSelect($\boldsymbol{P'}$); /* Fast-non-dominated sort and crowding distance*/
    $i = 0$;
    while($i < \boldsymbol{G}$){
        $i = i + 1$;
        $\boldsymbol{P'}$ = $\boldsymbol{P}$ + GeneticOperations($\boldsymbol{P}$); /*Combine population and offspring*/
        $\boldsymbol{P'}$ = FitnessCalculation($\boldsymbol{P'}$); /*Evaluating individuals*/
        $\boldsymbol{P}$ = MultiobjectiveSelect($\boldsymbol{P'}$);
    }
    $\boldsymbol{S}$ = SolutionPickAlgorithm($\boldsymbol{P}$);
    $\boldsymbol{UnclusteredBLEs}$ = $\boldsymbol{UnclusteredBLEs}$ - $\boldsymbol{S}$;
    $\boldsymbol{CLBs}$ = $\boldsymbol{CLBs} \cup \boldsymbol{S}$;
}

SaveNetlist($\boldsymbol{CLBs}$);

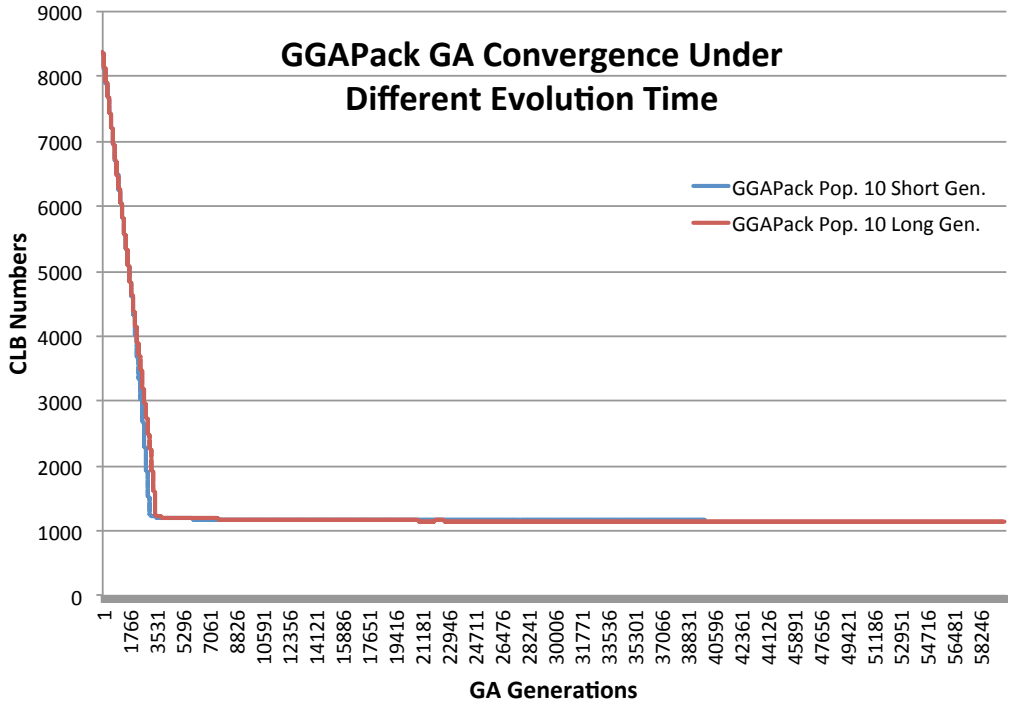


Figure A.2: GGAPack GA convergence under different evolution time – CLB numbers vs. GA generations. Gen. is short for generation number. The benchmark is "clma" – the largest benchmark in MCNC-20. Test shows that a large generation number is not able to further improve result quality. Short GA stops at 40,000 generations, long (large) GA stops at 60,000 generations. Since $25{,}000^{th}$ generation, there is no change in the results in both GAs.

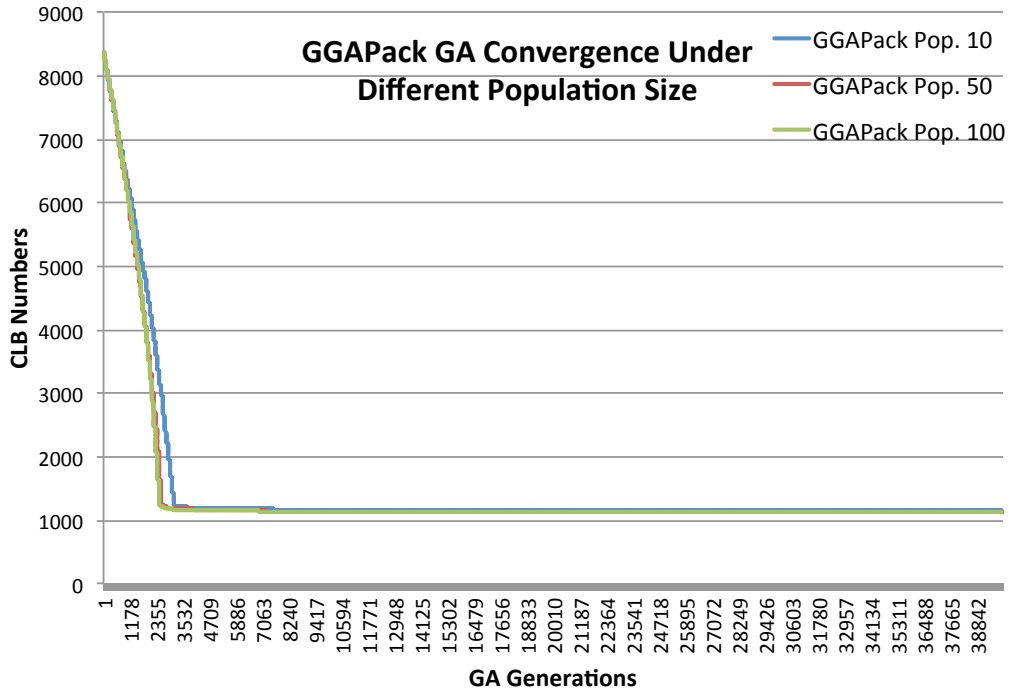


Figure A.3: GGAPack GA convergence under different population sizes – CLB numbers vs. GA generations. Pop. is short for population size. The benchmark is "clma" – the largest benchmark in MCNC-20. There is no huge difference, maximum is 2% - 3% in CLB numbers, when the population size is large, but a large population size can significantly slow down a GA execution time - a generation execution time is equal to individual evolution time by population size - when the population size is 100, entire GA execution time will be at least 10 times (1,000%) than the one that has population size 10.

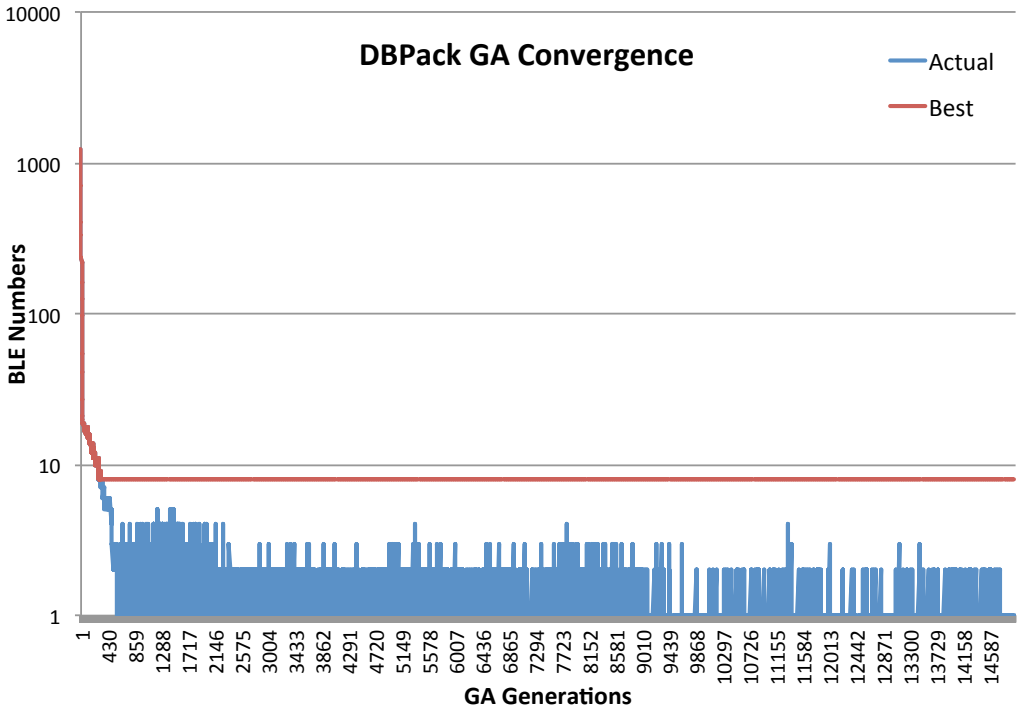


Figure A.4: DBPack GA convergence – BLE numbers (smallest BLE number among entire population) vs. GA generations when clustering the first CLB for benchmark "clma" – the largest benchmark in MCNC-20. There are two curves: The "Actual" curve shows the smallest BLE number found in GA population – one or a few individuals have this feature. Note that, in order to reduce the clustered CLB number for a clustered circuit, the BLE number is required to match or close to the CLB's BLE number, which is 8 (one CLB contains, $N = 8$, 8 BLEs) in this DBPack test. The other curve "Best" (best to a CLB) shows when individual has BLE number as 8 – this indicates the required BLE number individual (solution) is found. When generation number is equal to around 500, "BLE=8" solutions are appeared. Larger generation number designs to fully evolve individuals, where more Pareto optimal solutions can be selected.

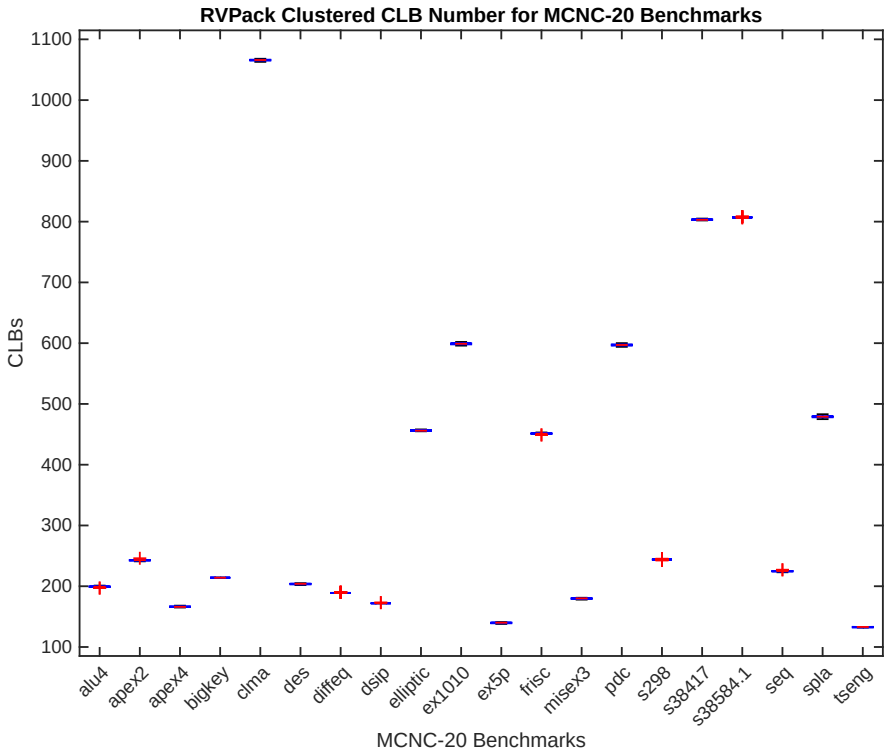


Figure A.5: Boxplot of RVPack clustered CLB number for MCNC-20 benchmarks

Table A.3: Medians of Figure A.5 – boxplot of RVPack clustered CLB number for MCNC-20 benchmarks

Benchmark	Median	Benchmark	Median
alu4	199	ex5p	140
apex2	243	frisc	451
apex4	166	misex3	180
bigkey	214	pdc	597
clma	1,066	s298	244
des	204	s38417	803
diffeq	189	s38584.1	807
dsip	172	seq	225
elliptic	456	spla	479
ex1010	599	tseng	133

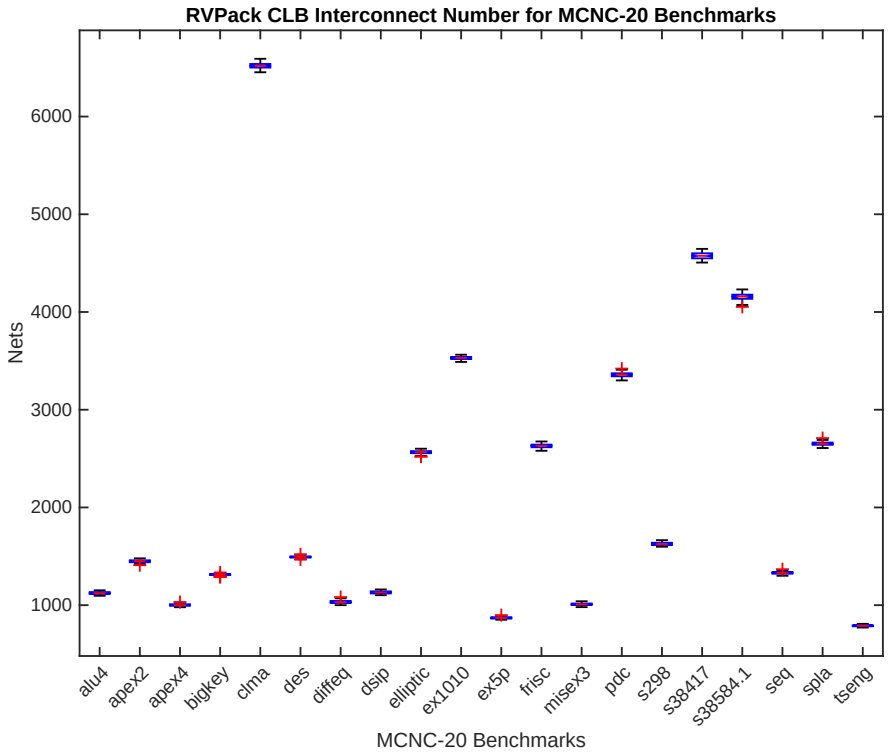


Figure A.6: Boxplot of RVPack clustered CLB interconnect number for MCNC-20 benchmarks

Table A.4: Medians of Figure A.6 – boxplot of RVPack clustered CLB interconnect number for MCNC-20 benchmarks

Benchmark	Median	Benchmark	Median
alu4	1,123	ex5p	869
apex2	1,451	frisc	2,631
apex4	1,001	misex3	1,010
bigkey	1,313	pdc	3,360
clma	6,519	s298	1,624
des	1,493	s38417	4,575
diffeq	1,031	s38584.1	4,159
dsip	1,131	seq	1,331
elliptic	2,567	spla	2,652
ex1010	3,531	tseng	790

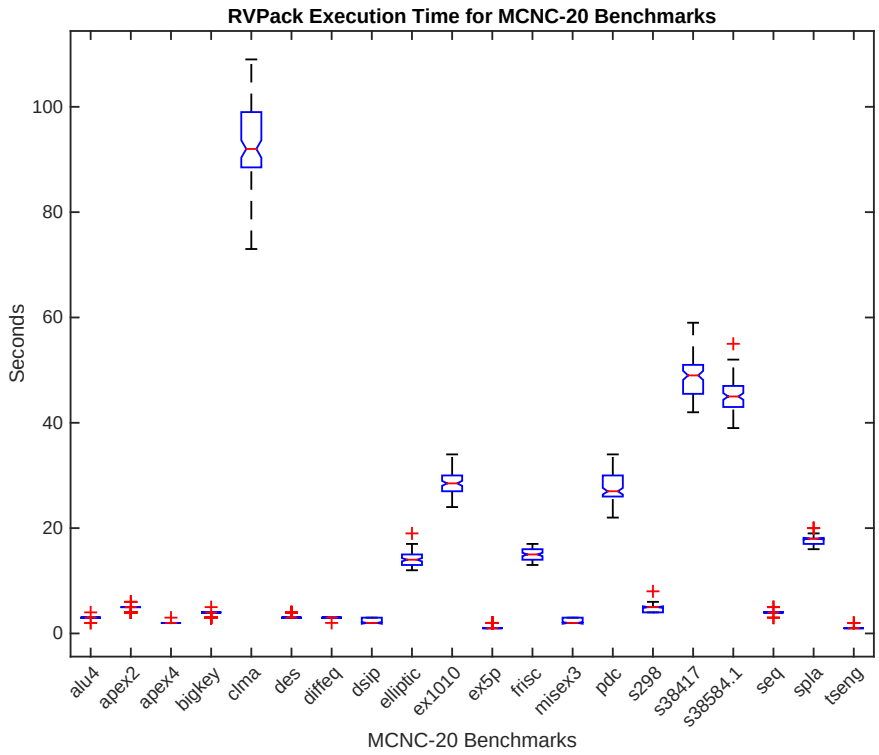


Figure A.7: Boxplot of RVPack execution time for MCNC-20 benchmarks

Table A.5: Medians of Figure A.7 – boxplot of RVPack execution time for MCNC-20 benchmarks

Benchmark	Median	Benchmark	Median
alu4	3.00	ex5p	1.00
apex2	5.00	frisc	15.00
apex4	2.00	misex3	2.00
bigkey	4.00	pdc	27.00
clma	92.00	s298	5.00
des	3.00	s38417	49.00
diffeq	3.00	s38584.1	45.00
dsip	2.00	seq	4.00
elliptic	14.00	spla	18.00
ex1010	28.50	tseng	1.00

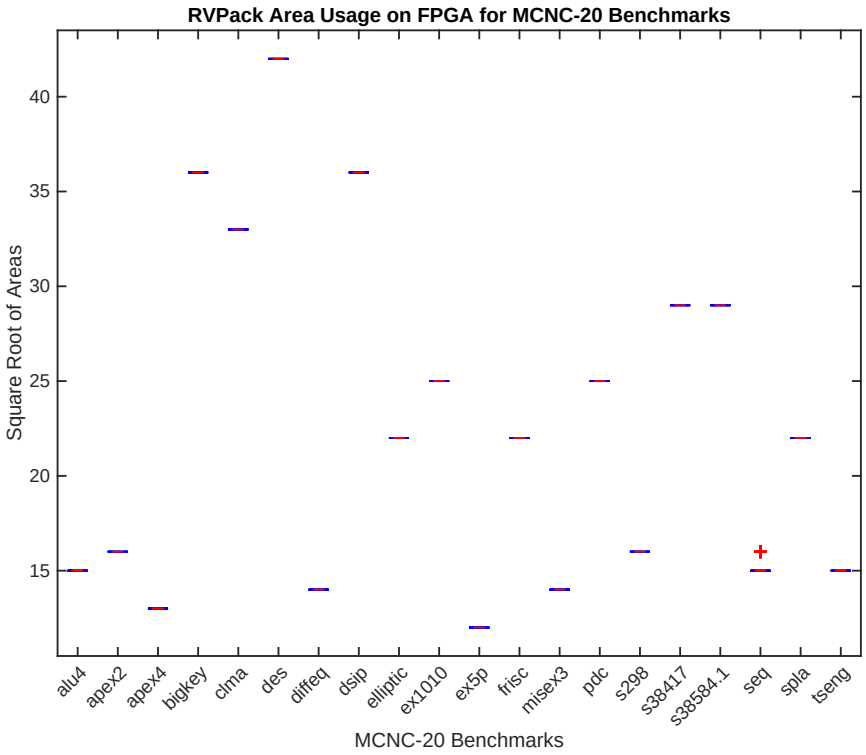


Figure A.8: Boxplot of RVPack on FPGA area usages for MCNC-20 benchmarks

Table A.6: Medians of Figure A.8 – boxplot of RVPack on FPGA area usages for MCNC-20 benchmarks

Benchmark	Median	Benchmark	Median
alu4	15	ex5p	12
apex2	16	frisc	22
apex4	13	misex3	14
bigkey	36	pdc	25
clma	33	s298	16
des	42	s38417	29
diffeq	14	s38584.1	29
dsip	36	seq	15
elliptic	22	spla	22
ex1010	25	tseng	15

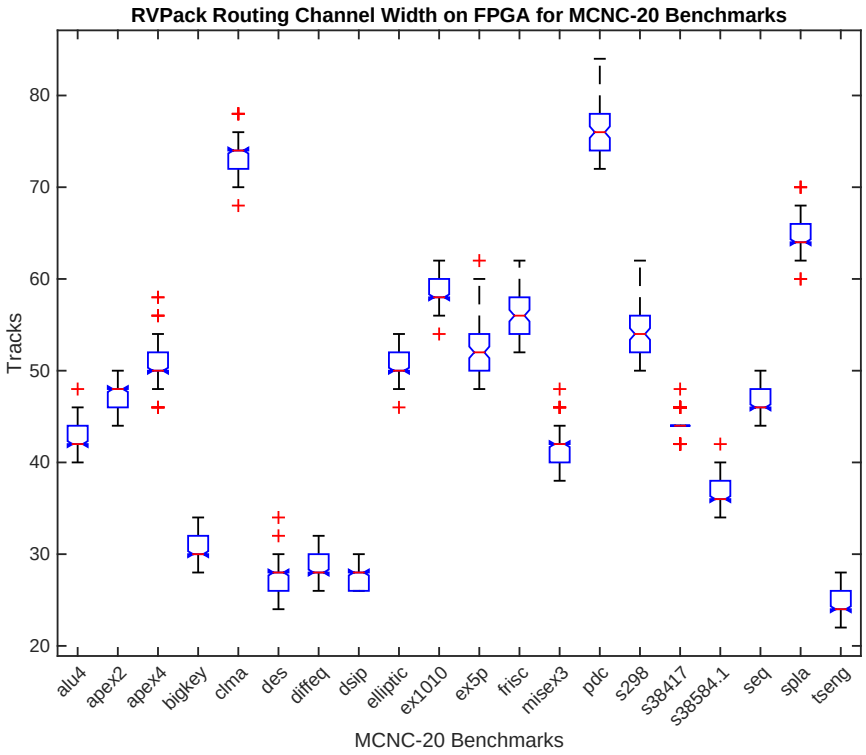


Figure A.9: Boxplot of RVPack on FPGA channel widths for MCNC-20 benchmarks

Table A.7: Medians of Figure A.9 – boxplot of RVPack on FPGA channel widths for MCNC-20 benchmarks

Benchmark	Median	Benchmark	Median
alu4	42	ex5p	52
apex2	48	frisc	56
apex4	50	misex3	42
bigkey	30	pdc	76
clma	74	s298	54
des	28	s38417	44
diffeq	28	s38584.1	36
dsip	28	seq	46
elliptic	50	spla	64
ex1010	58	tseng	24

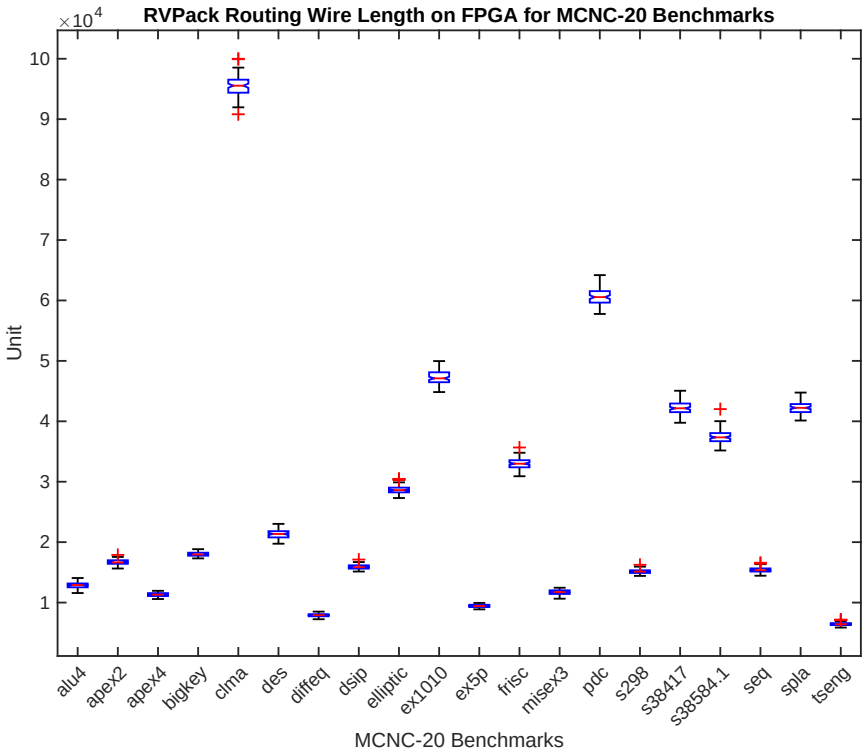


Figure A.10: Boxplot of RVPack on FPGA wire lengths for MCNC-20 benchmarks

Table A.8: Medians of Figure A.10 – boxplot of RVPack on FPGA wire lengths for MCNC-20 benchmarks

Benchmark	Median	Benchmark	Median
alu4	12,905	ex5p	9,427
apex2	16,637	frisc	32,983
apex4	11,318	misex3	11,734
bigkey	17,970	pdc	60,554
clma	95,542	s298	15,122
des	21,355	s38417	42,152
diffeq	7,906	s38584.1	37,346
dsip	15,853	seq	15,353
elliptic	28,607	spla	42,229
ex1010	47,109	tseng	6,419

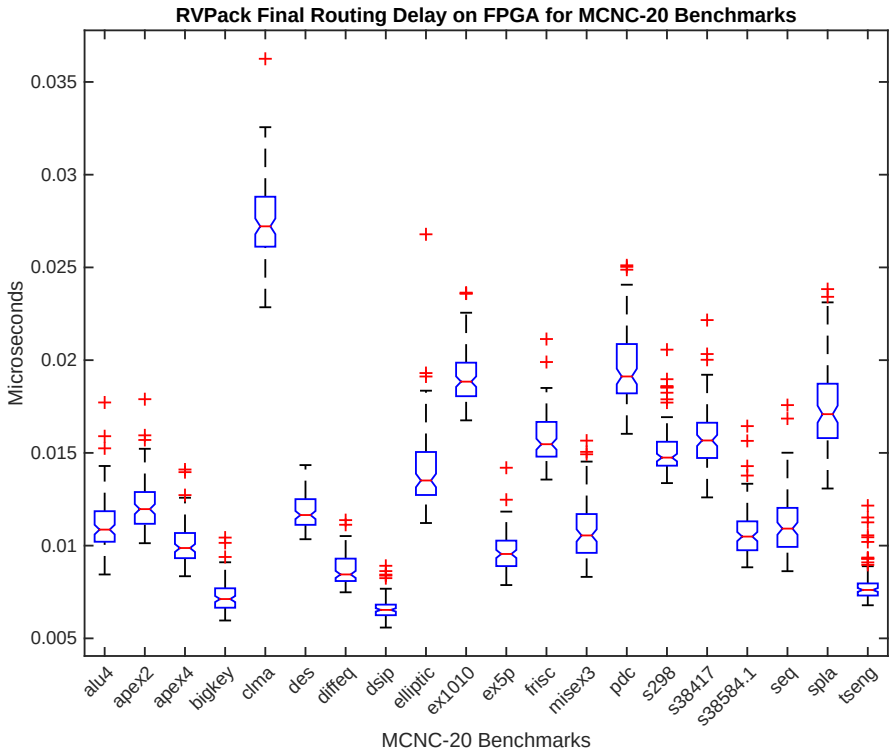


Figure A.11: Boxplot of RVPack on FPGA circuit-critical-path delays for MCNC-20 benchmarks

Table A.9: Medians of Figure A.11 – boxplot of RVPack on FPGA circuit-critical-path delays for MCNC-20 benchmarks

Benchmark	Median ($*10^{-02}\mu S$)	Benchmark	Median ($*10^{-02}\mu S$)
alu4	1.09	ex5p	0.95
apex2	1.20	frisc	1.55
apex4	0.99	misex3	1.05
bigkey	0.71	pdc	1.91
clma	2.72	s298	1.47
des	1.16	s38417	1.57
diffeq	0.84	s38584.1	1.05
dsip	0.65	seq	1.09
elliptic	1.35	spla	1.71
ex1010	1.88	tseng	0.76

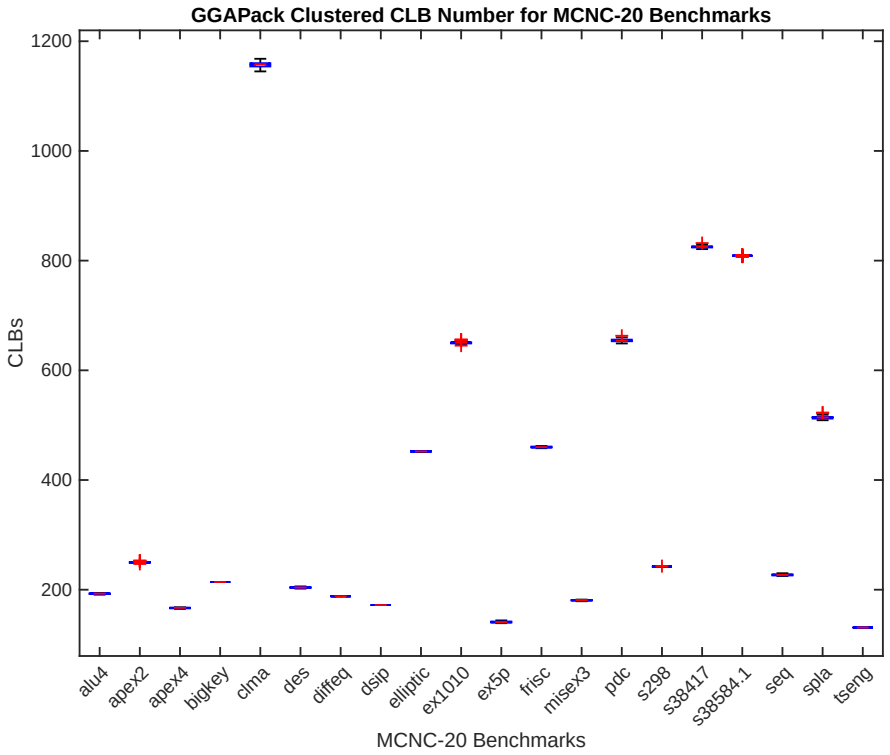


Figure A.12: Boxplot of GGAPack clustered CLB number for MCNC-20 benchmarks

Table A.10: Medians of Figure A.12 – boxplot of GGAPack clustered CLB number for MCNC-20 benchmarks

Benchmark	Median	Benchmark	Median
alu4	193	ex5p	141
apex2	250	frisc	460
apex4	167	misex3	181
bigkey	214	pdc	654
clma	1,158	s298	242
des	204	s38417	825
diffeq	188	s38584.1	809
dsip	172	seq	227
elliptic	452	spla	514
ex1010	651	tseng	131

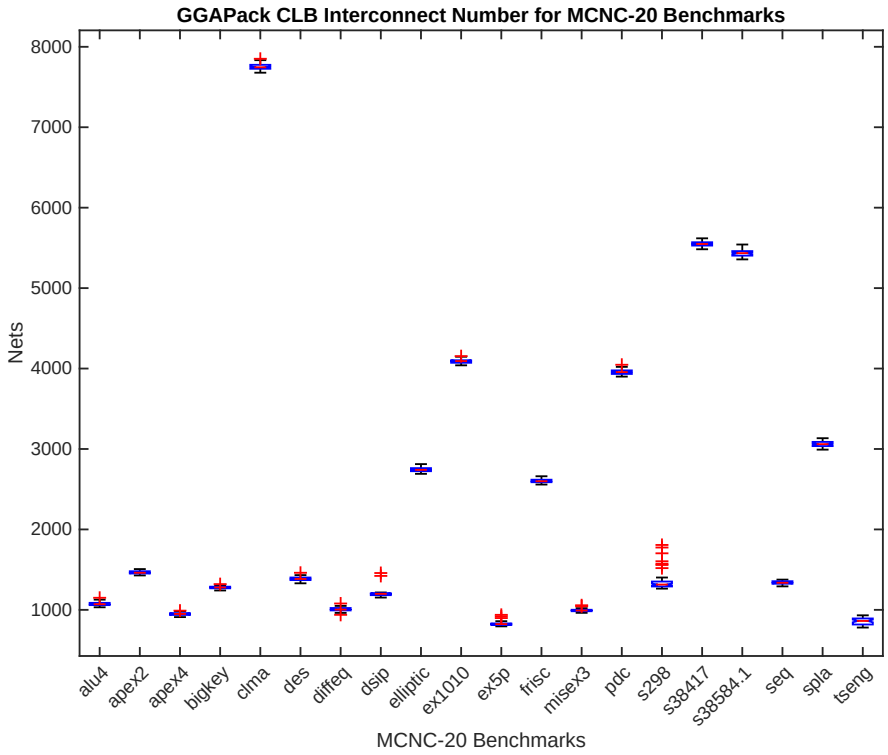


Figure A.13: Boxplot of GGAPack clustered CLB interconnect number for MCNC-20 benchmarks

Table A.11: Medians of Figure A.13 – boxplot of GGAPack clustered CLB interconnect number for MCNC-20 benchmarks

Benchmark	Median	Benchmark	Median
alu4	1,071	ex5p	823
apex2	1,461	frisc	2,599
apex4	950	misex3	992
bigkey	1,279	pdc	3,956
clma	7,746	s298	1,315
des	1,384	s38417	5,549
diffeq	1,004	s38584.1	5,435
dsip	1,199	seq	1,337
elliptic	2,741	spla	3,060
ex1010	4,093	tseng	862

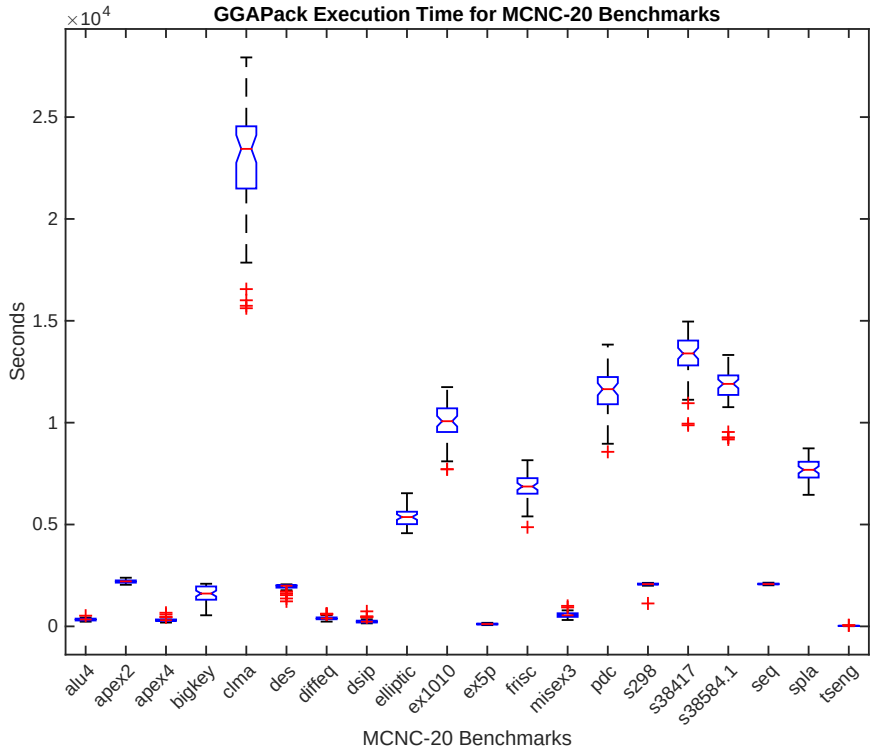


Figure A.14: Boxplot of GGAPack execution time for MCNC-20 benchmarks

Table A.12: Medians of Figure A.14 – boxplot of GGAPack execution time for MCNC-20 benchmarks

Benchmark	Median	Benchmark	Median
alu4	339.00	ex5p	115.50
apex2	2,207.00	frisc	6,865.50
apex4	301.00	misex3	543.00
bigkey	1,611.50	pdc	11,644.00
clma	23,439.00	s298	2,075.00
des	1,993.50	s38417	13,398.00
diffeq	388.00	s38584.1	11,903.00
dsip	230.00	seq	2,077.50
elliptic	5,366.00	spla	7,684.50
ex1010	10,071.00	tseng	25.00

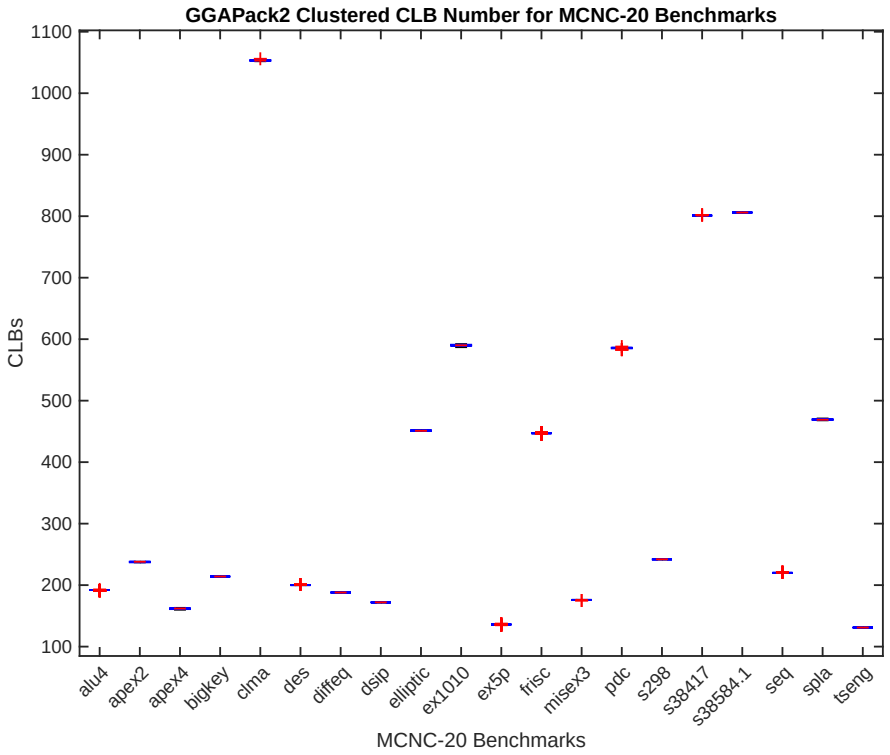


Figure A.15: Boxplot of GGAPack2 clustered CLB number for MCNC-20 benchmarks

Table A.13: Medians of Figure A.15 – boxplot of GGAPack2 clustered CLB number for MCNC-20 benchmarks

Benchmark	Median	Benchmark	Median
alu4	192	ex5p	136
apex2	238	frisc	447
apex4	162	misex3	176
bigkey	214	pdc	586
clma	1,053	s298	242
des	200	s38417	801
diffeq	188	s38584.1	806
dsip	172	seq	220
elliptic	451	spla	469
ex1010	590	tseng	131

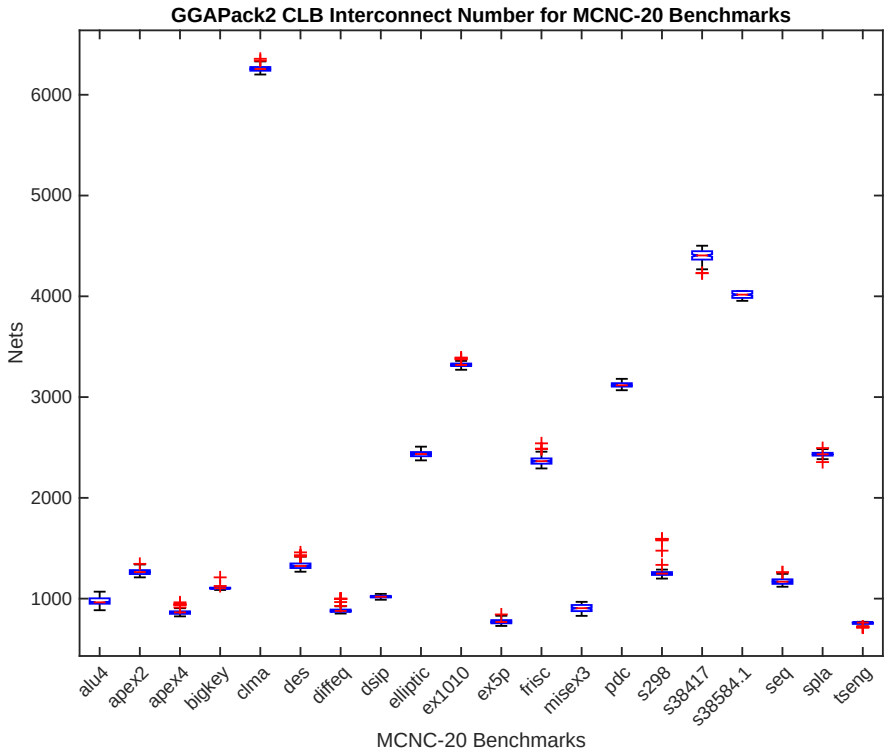


Figure A.16: Boxplot of GGAPack2 clustered CLB interconnect number for MCNC-20 benchmarks

Table A.14: Medians of Figure A.16 – boxplot of GGAPack2 clustered CLB interconnect number for MCNC-20 benchmarks

Benchmark	Median	Benchmark	Median
alu4	963	ex5p	768
apex2	1,264	frisc	2,364
apex4	864	misex3	905
bigkey	1,103	pdc	3,117
clma	6,256	s298	1,248
des	1,324	s38417	4,406
diffeq	878	s38584.1	4,017
dsip	1,019	seq	1,168
elliptic	2,435	spla	2,432
ex1010	3,319	tseng	762

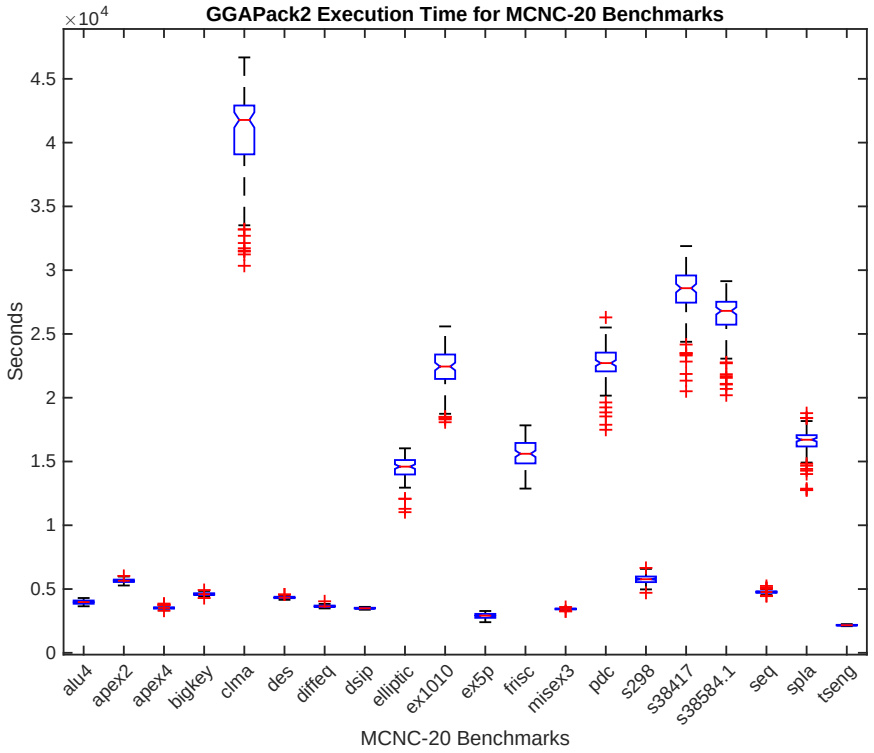


Figure A.17: Boxplot of GGAPack2 execution time for MCNC-20 benchmarks

Table A.15: Medians of Figure A.17 – boxplot of GGAPack2 execution time for MCNC-20 benchmarks

Benchmark	Median	Benchmark	Median
alu4	3,978.00	ex5p	2,916.50
apex2	5,651.50	frisc	15,605.00
apex4	3,518.00	misex3	3,437.00
bigkey	4,596.00	pdc	22,714.00
clma	41,776.00	s298	5,769.50
des	4,326.50	s38417	28,586.50
diffeq	3,641.50	s38584.1	26,808.50
dsip	3,492.00	seq	4,768.50
elliptic	14,597.00	spla	16,705.00
ex1010	22,445.50	tseng	2,165.50

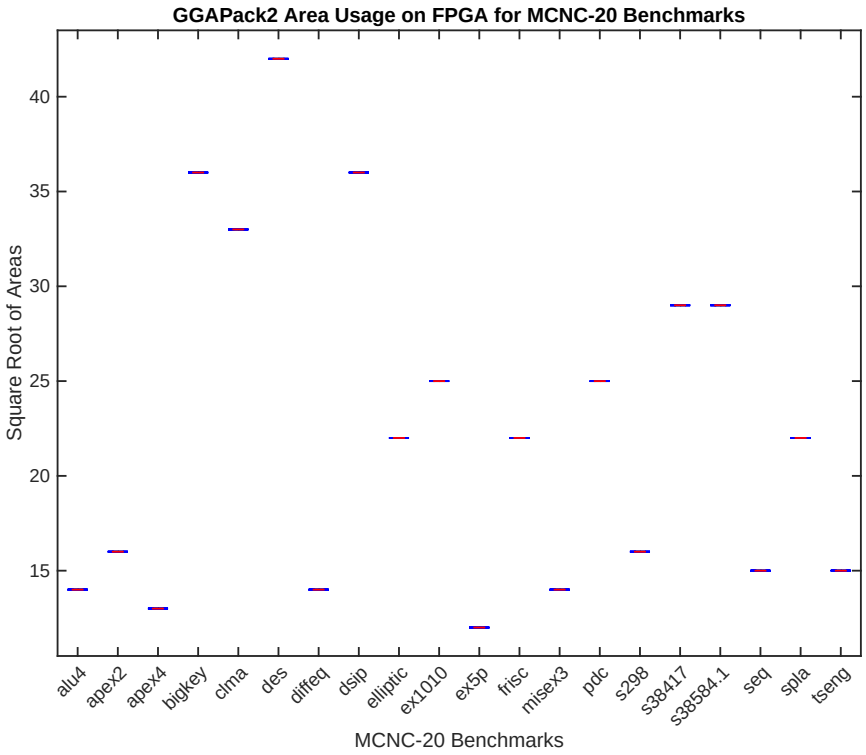


Figure A.18: Boxplot of GGAPack2 on FPGA area usages for MCNC-20 benchmarks

Table A.16: Medians of Figure A.18 – boxplot of GGAPack2 on FPGA area usages for MCNC-20 benchmarks

Benchmark	Median	Benchmark	Median
alu4	14	ex5p	12
apex2	16	frisc	22
apex4	13	misex3	14
bigkey	36	pdc	25
clma	33	s298	16
des	42	s38417	29
diffeq	14	s38584.1	29
dsip	36	seq	15
elliptic	22	spla	22
ex1010	25	tseng	15

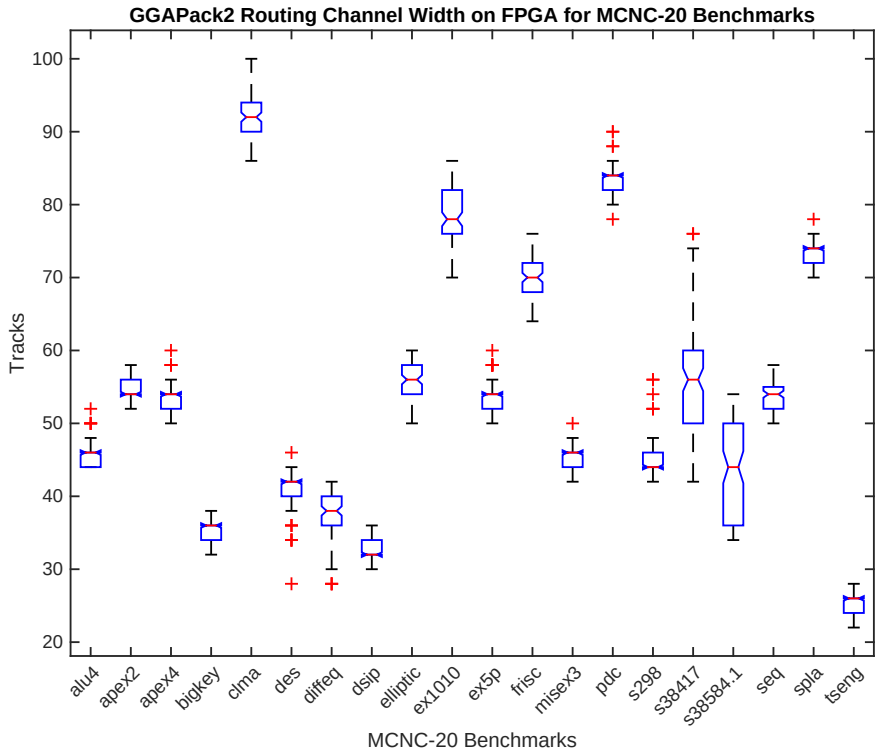


Figure A.19: Boxplot of GGAPack2 on FPGA channel widths for MCNC-20 benchmarks

Table A.17: Medians of Figure A.19 – boxplot of GGAPack2 on FPGA channel widths for MCNC-20 benchmarks

Benchmark	Median	Benchmark	Median
alu4	46	ex5p	54
apex2	54	frisc	70
apex4	54	misex3	46
bigkey	36	pdc	84
clma	92	s298	44
des	42	s38417	56
diffeq	38	s38584.1	44
dsip	32	seq	54
elliptic	56	spla	74
ex1010	78	tseng	26

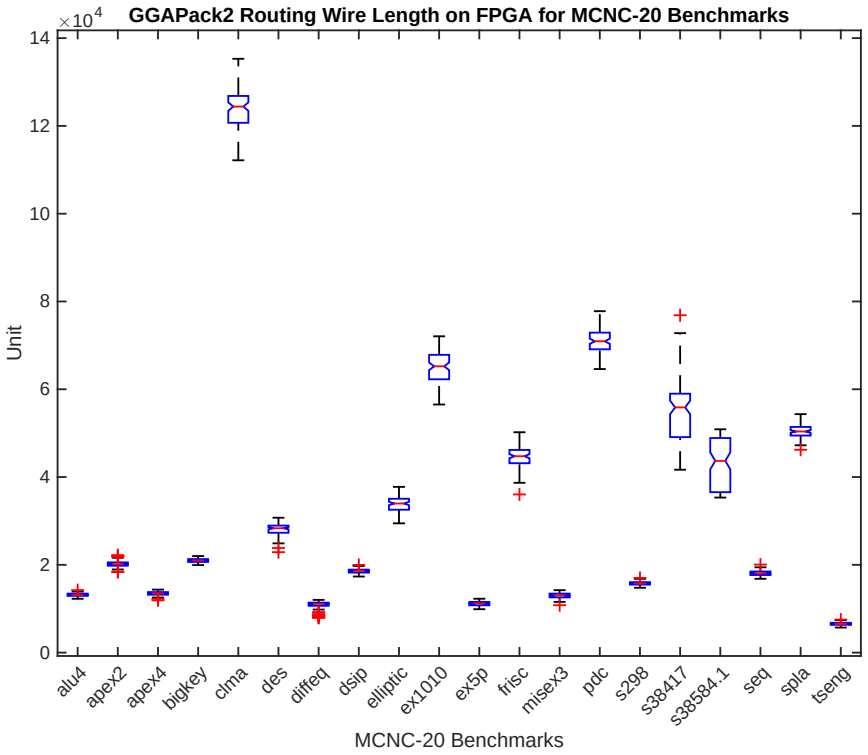


Figure A.20: Boxplot of GGAPack2 on FPGA wire lengths for MCNC-20 benchmarks

Table A.18: Medians of Figure A.20 – boxplot of GGAPack2 on FPGA wire lengths for MCNC-20 benchmarks

Benchmark	Median	Benchmark	Median
alu4	13,153	ex5p	11,202
apex2	20,130	frisc	44,745
apex4	13,474	misex3	13,048
bigkey	20,990	pdc	70,933
clma	124,399	s298	15,748
des	28,335	s38417	55,864
diffeq	11,093	s38584.1	43,684
dsip	18,628	seq	18,066
elliptic	33,975	spla	50,377
ex1010	65,231	tseng	6,532

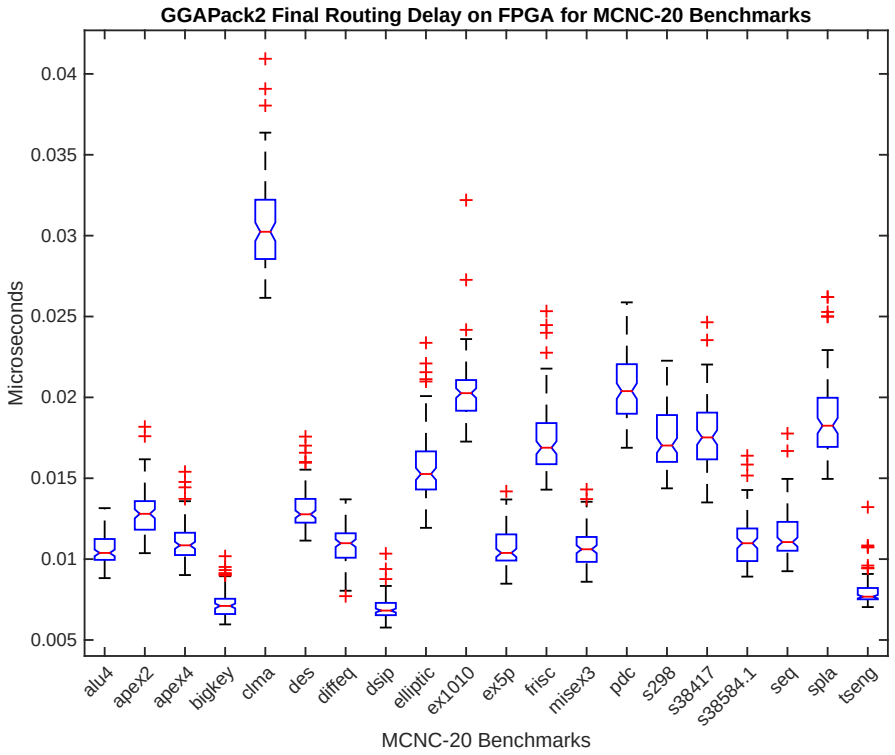


Figure A.21: Boxplot of GGAPack2 on FPGA circuit-critical-path delays for MCNC-20 benchmarks

Table A.19: Medians of Figure A.21 – boxplot of GGAPack2 on FPGA circuit-critical-path delays for MCNC-20 benchmarks

Benchmark	Median ($*10^{-02}\mu S$)	Benchmark	Median ($*10^{-02}\mu S$)
alu4	1.04	ex5p	1.04
apex2	1.28	frisc	1.69
apex4	1.09	misex3	1.06
bigkey	0.71	pdc	2.03
clma	3.02	s298	1.70
des	1.28	s38417	1.75
diffeq	1.10	s38584.1	1.10
dsip	0.68	seq	1.11
elliptic	1.53	spla	1.82
ex1010	2.03	tseng	0.77

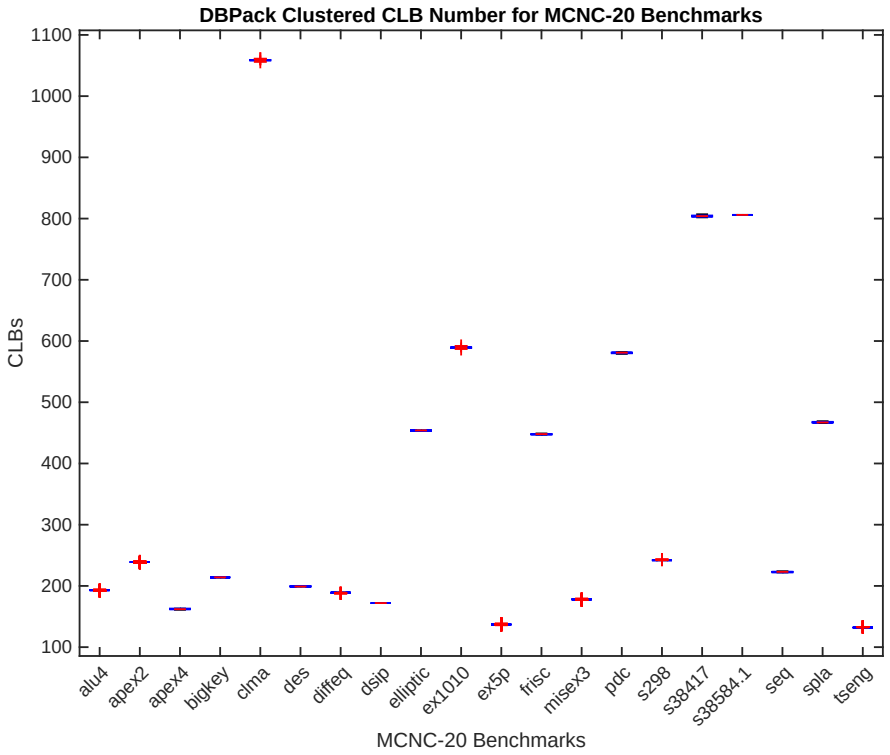


Figure A.22: Boxplot of DBPack clustered CLB number for MCNC-20 benchmarks

Table A.20: Medians of Figure A.22 – boxplot of DBPack clustered CLB number for MCNC-20 benchmarks

Benchmark	Median	Benchmark	Median
alu4	193	ex5p	137
apex2	239	frisc	448
apex4	162	misex3	178
bigkey	214	pdc	581
clma	1,059	s298	242
des	199	s38417	804
diffeq	189	s38584.1	806
dsip	172	seq	223
elliptic	454	spla	467
ex1010	589	tseng	132

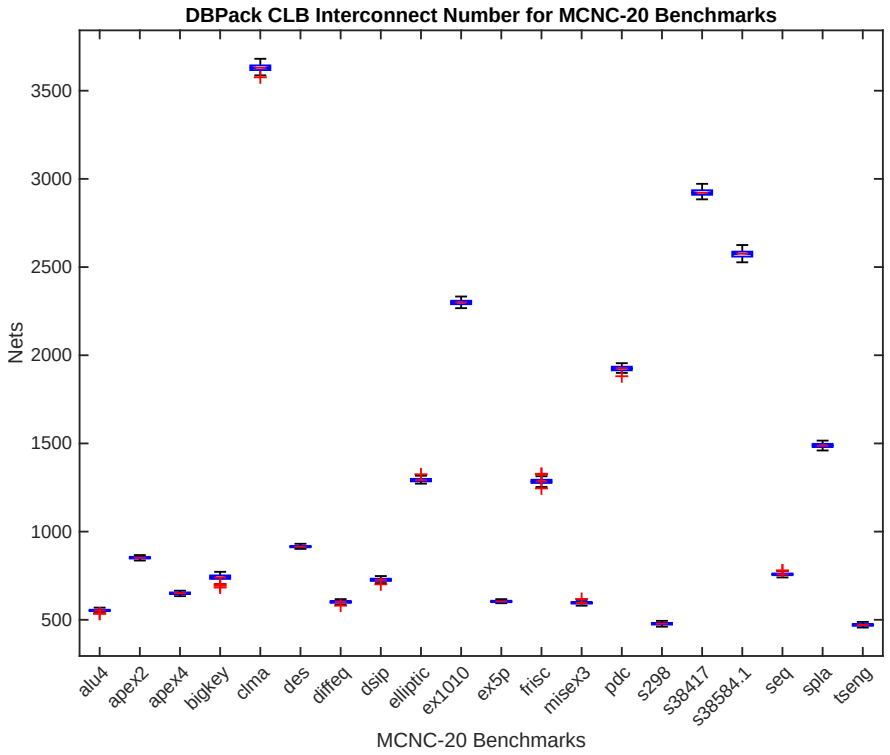


Figure A.23: Boxplot of DBPack clustered CLB interconnect number for MCNC-20 benchmarks

Table A.21: Medians of Figure A.23 – boxplot of DBPack clustered CLB interconnect number for MCNC-20 benchmarks

Benchmark	Median	Benchmark	Median
alu4	553	ex5p	604
apex2	852	frisc	1,284
apex4	651	misex3	596
bigkey	740	pdc	1,927
clma	3,630	s298	478
des	915	s38417	2,921
diffeq	601	s38584.1	2,576
dsip	727	seq	758
elliptic	1,292	spla	1,487
ex1010	2,300	tseng	471

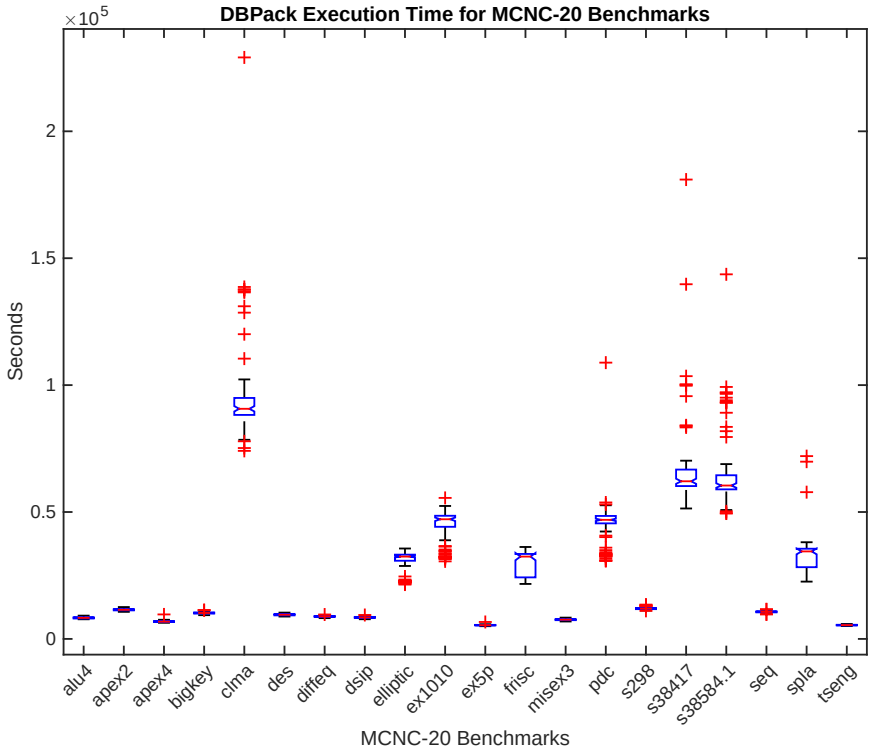


Figure A.24: Boxplot of DBPack execution time for MCNC-20 benchmarks

Table A.22: Medians of Figure A.24 – boxplot of DBPack execution time for MCNC-20 benchmarks

Benchmark	Median	Benchmark	Median
alu4	8,295.00	ex5p	5,421.00
apex2	11,469.50	frisc	32,439.50
apex4	6,804.50	misex3	7,585.50
bigkey	10,213.50	pdc	46,917.50
clma	90,665.50	s298	11,928.50
des	9,541.50	s38417	62,108.50
diffeq	8,857.50	s38584.1	60,437.00
dsip	8,477.00	seq	10,656.50
elliptic	32,442.50	spla	34,476.50
ex1010	47,142.50	tseng	5,445.00

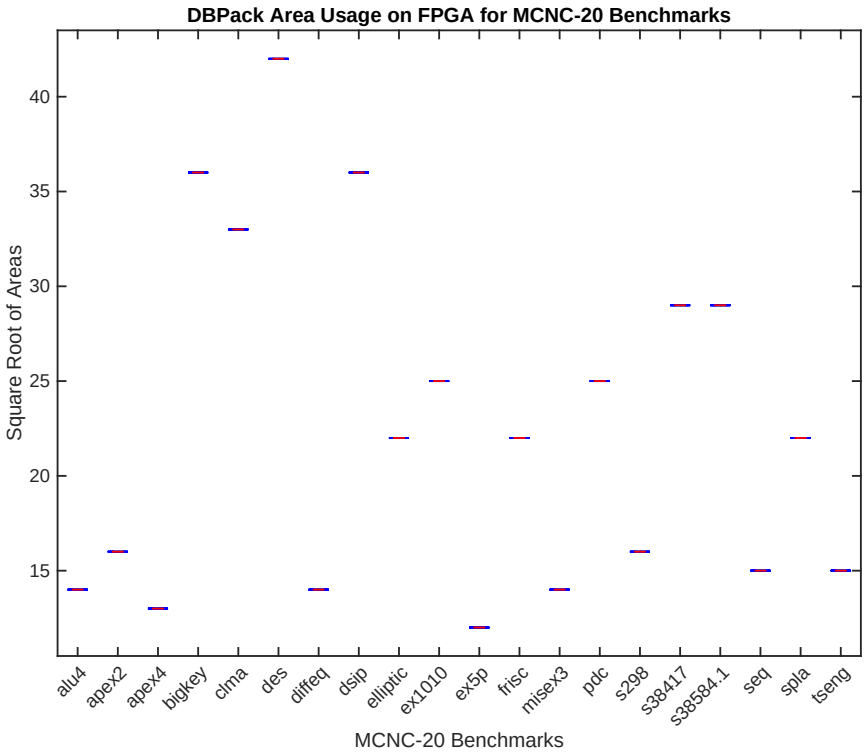


Figure A.25: Boxplot of DBPack on FPGA area usages for MCNC-20 benchmarks

Table A.23: Medians of Figure A.25 – boxplot of DBPack on FPGA area usages for MCNC-20 benchmarks

Benchmark	Median	Benchmark	Median
alu4	14	ex5p	12
apex2	16	frisc	22
apex4	13	misex3	14
bigkey	36	pdc	25
clma	33	s298	16
des	42	s38417	29
diffeq	14	s38584.1	29
dsip	36	seq	15
elliptic	22	spla	22
ex1010	25	tseng	15

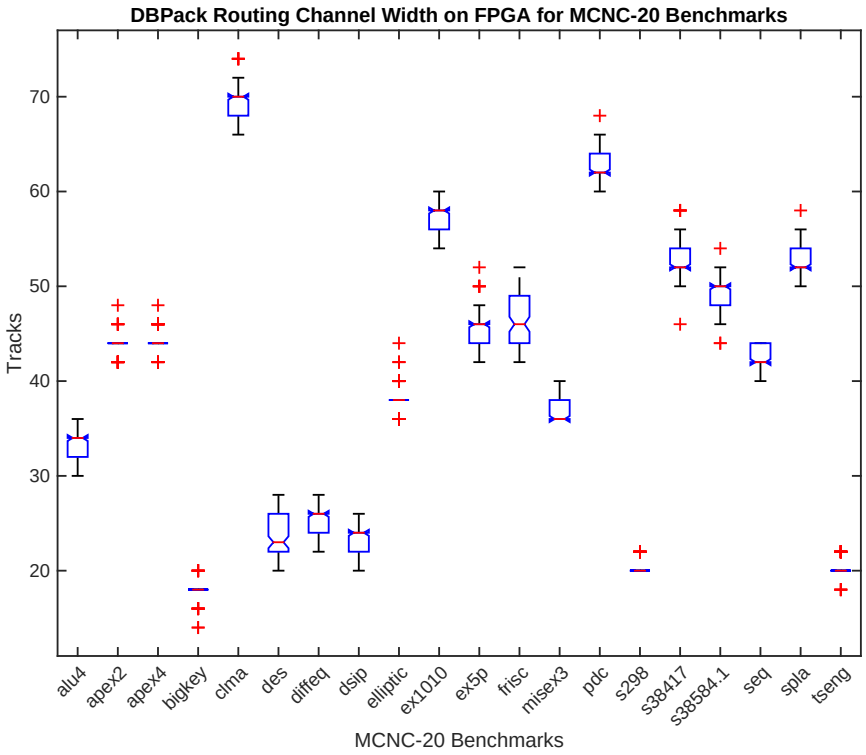


Figure A.26: Boxplot of DBPack on FPGA channel widths for MCNC-20 benchmarks

Table A.24: Medians of Figure A.26 – boxplot of DBPack on FPGA channel widths for MCNC-20 benchmarks

Benchmark	Median	Benchmark	Median
alu4	34	ex5p	46
apex2	44	frisc	46
apex4	44	misex3	36
bigkey	18	pdc	62
clma	70	s298	20
des	23	s38417	52
diffeq	26	s38584.1	50
dsip	24	seq	42
elliptic	38	spla	52
ex1010	58	tseng	20

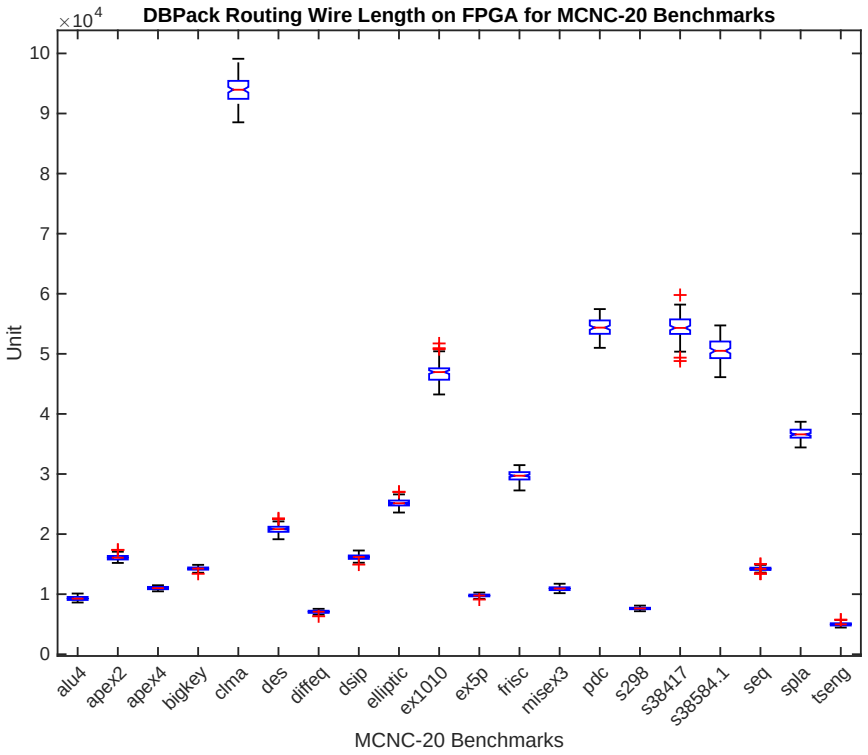


Figure A.27: Boxplot of DBPack on FPGA wire lengths for MCNC-20 benchmarks

Table A.25: Medians of Figure A.27 – boxplot of DBPack on FPGA wire lengths for MCNC-20 benchmarks

Benchmark	Median	Benchmark	Median
alu4	9,258	ex5p	9,766
apex2	16,069	frisc	29,716
apex4	11,000	misex3	10,942
bigkey	14,259	pdc	54,363
clma	93,955	s298	7,628
des	20,825	s38417	54,313
diffeq	7,076	s38584.1	50,502
dsip	16,193	seq	14,215
elliptic	25,135	spla	36,597
ex1010	46,962	tseng	4,984

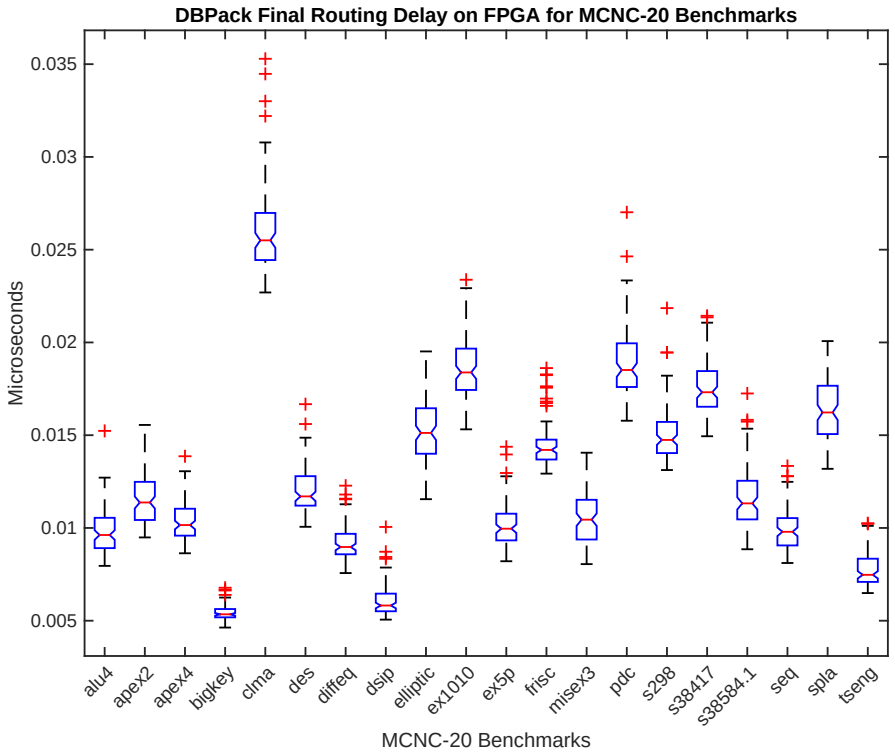


Figure A.28: Boxplot of DBPack on FPGA circuit-critical-path delays for MCNC-20 benchmarks

Table A.26: Medians of Figure A.28 – boxplot of DBPack on FPGA circuit-critical-path delays for MCNC-20 benchmarks

Benchmark	Median ($*10^{-02}\mu S$)	Benchmark	Median ($*10^{-02}\mu S$)
alu4	0.96	ex5p	1.00
apex2	1.14	frisc	1.42
apex4	1.02	misex3	1.04
bigkey	0.53	pdc	1.85
clma	2.55	s298	1.47
des	1.17	s38417	1.73
diffeq	0.90	s38584.1	1.13
dsip	0.58	seq	0.98
elliptic	1.51	spla	1.62
ex1010	1.84	tseng	0.75

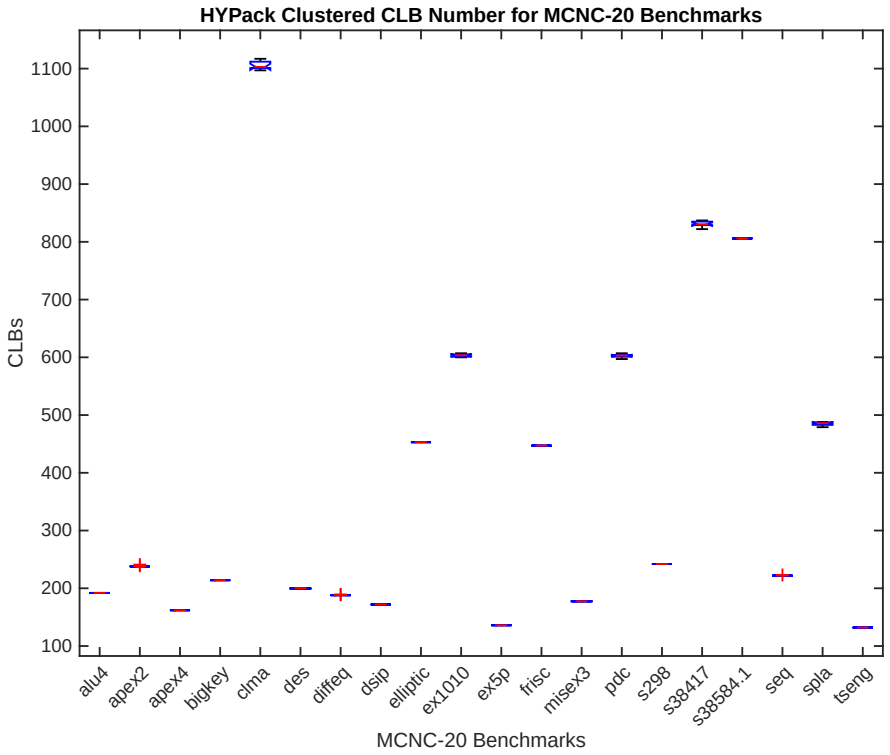


Figure A.29: Boxplot of HYPack clustered CLB number for MCNC-20 benchmarks

Table A.27: Medians of Figure A.29 – boxplot of HYPack clustered CLB number for MCNC-20 benchmarks

Benchmark	Median	Benchmark	Median
alu4	192	ex5p	136
apex2	238	frisc	447
apex4	162	misex3	178
bigkey	214	pdc	603
clma	1,103	s298	242
des	200	s38417	831
diffeq	188	s38584.1	806
dsip	172	seq	222
elliptic	453	spla	486
ex1010	604	tseng	132

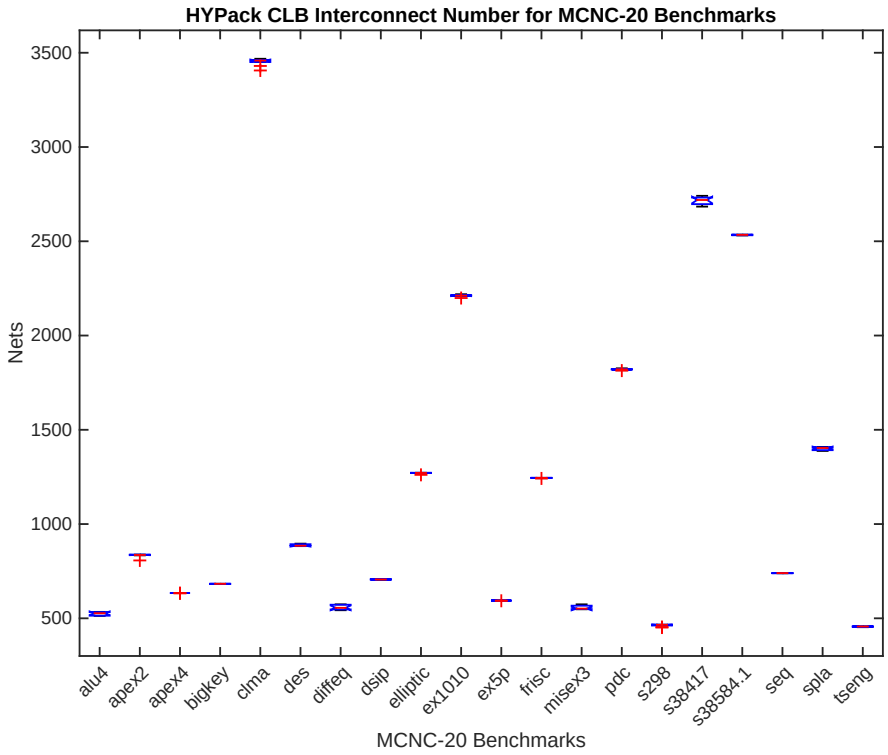


Figure A.30: Boxplot of HYPack clustered CLB interconnect number for MCNC-20 benchmarks

Table A.28: Medians of Figure A.30 – boxplot of HYPack clustered CLB interconnect number for MCNC-20 benchmarks

Benchmark	Median	Benchmark	Median
alu4	527	ex5p	594
apex2	836	frisc	1,245
apex4	634	misex3	552
bigkey	683	pdc	1,821
clma	3,459	s298	466
des	885	s38417	2,719
diffeq	556	s38584.1	2,534
dsip	706	seq	740
elliptic	1,271	spla	1,404
ex1010	2,212	tseng	457

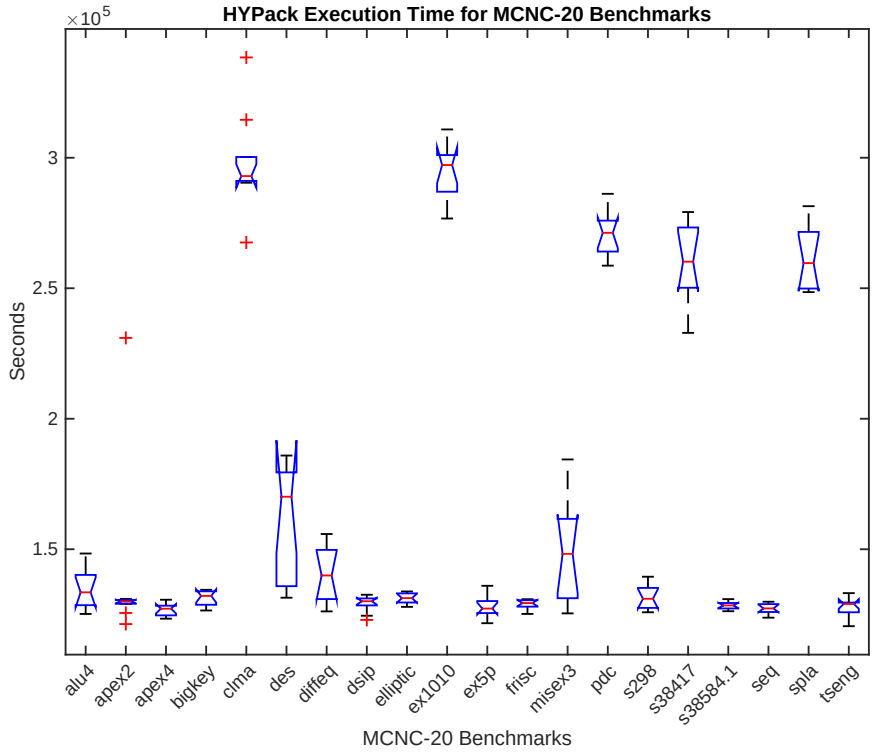


Figure A.31: Boxplot of HYPack execution time for MCNC-20 benchmarks

Table A.29: Medians of Figure A.31 – boxplot of HYPack execution time for MCNC-20 benchmarks

Benchmark	Median	Benchmark	Median
alu4	527	ex5p	594
apex2	836	frisc	1,245
apex4	634	misex3	552
bigkey	683	pdc	1,821
clma	3,459	s298	466
des	885	s38417	2,719
diffeq	556	s38584.1	2,534
dsip	706	seq	740
elliptic	1,271	spla	1,404
ex1010	2,212	tseng	457

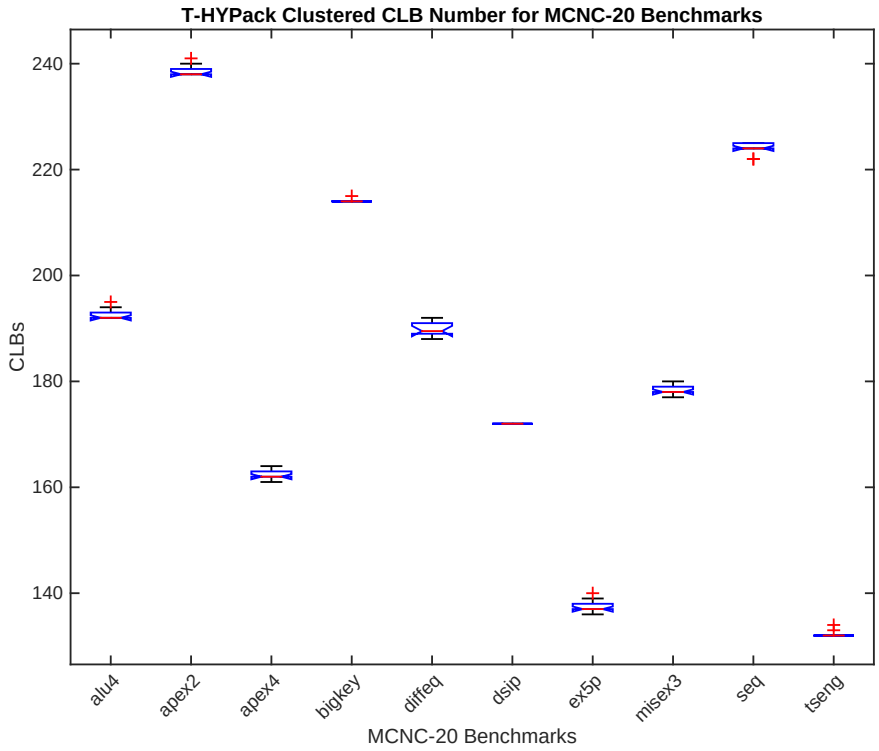


Figure A.32: Boxplot of T-HYPack clustered CLB number for MCNC-20 benchmarks

Table A.30: Medians of Figure A.32 – boxplot of T-HYPack clustered CLB number for MCNC-20 benchmarks

Benchmark	Median	Benchmark	Median
alu4	192	dsip	172
apex2	238	ex5p	137
apex4	162	misex3	178
bigkey	214	seq	224
diffeq	190	tseng	132

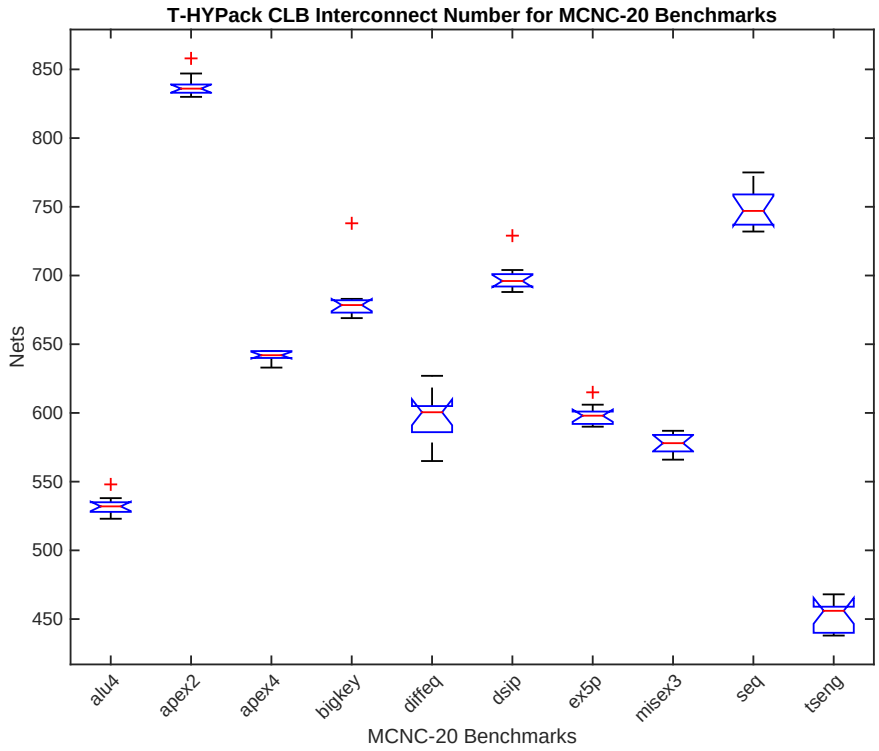


Figure A.33: Boxplot of T-HYPack clustered CLB interconnect number for MCNC-20 benchmarks

Table A.31: Medians of Figure A.33 – boxplot of T-HYPack clustered CLB interconnect number for MCNC-20 benchmarks

Benchmark	Median	Benchmark	Median
alu4	532	dsip	696
apex2	836	ex5p	598
apex4	642	misex3	578
bigkey	679	seq	747
diffeq	601	tseng	456

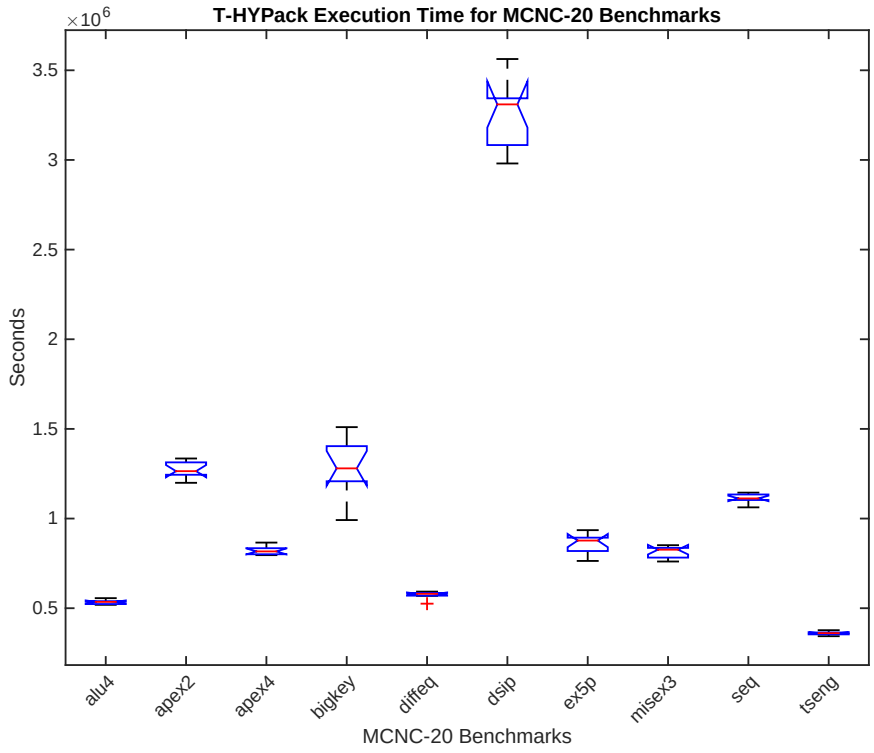


Figure A.34: Boxplot of T-HYPack execution time for MCNC-20 benchmarks

Table A.32: Medians of Figure A.34 – boxplot of T-HYPack execution time for MCNC-20 benchmarks

Benchmark	Median	Benchmark	Median
alu4	535,447.00	dsip	3,309,867.50
apex2	1,264,309.50	ex5p	877,343.50
apex4	816,778.00	misex3	826,523.50
bigkey	1,279,867.00	seq	1,112,237.50
diffeq	580,460.00	tseng	360,889.00

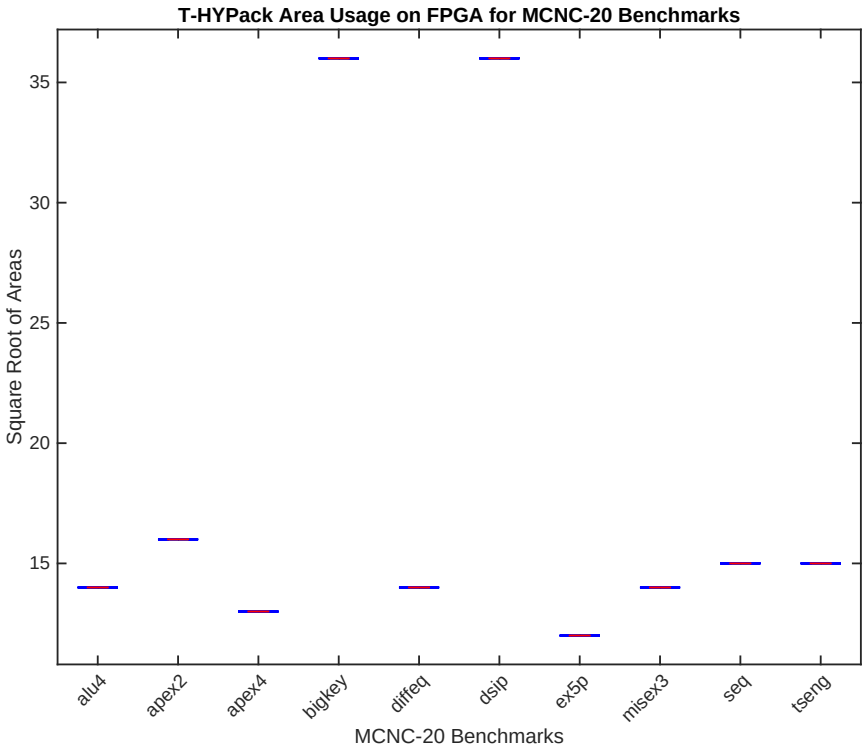


Figure A.35: Boxplot of T-HYPack on FPGA area usages for MCNC-20 benchmarks

Table A.33: Medians of Figure A.35 – boxplot of T-HYPack on FPGA area usages for MCNC-20 benchmarks

Benchmark	Median	Benchmark	Median
alu4	14	dsip	36
apex2	16	ex5p	12
apex4	13	misex3	14
bigkey	36	seq	15
diffeq	14	tseng	15

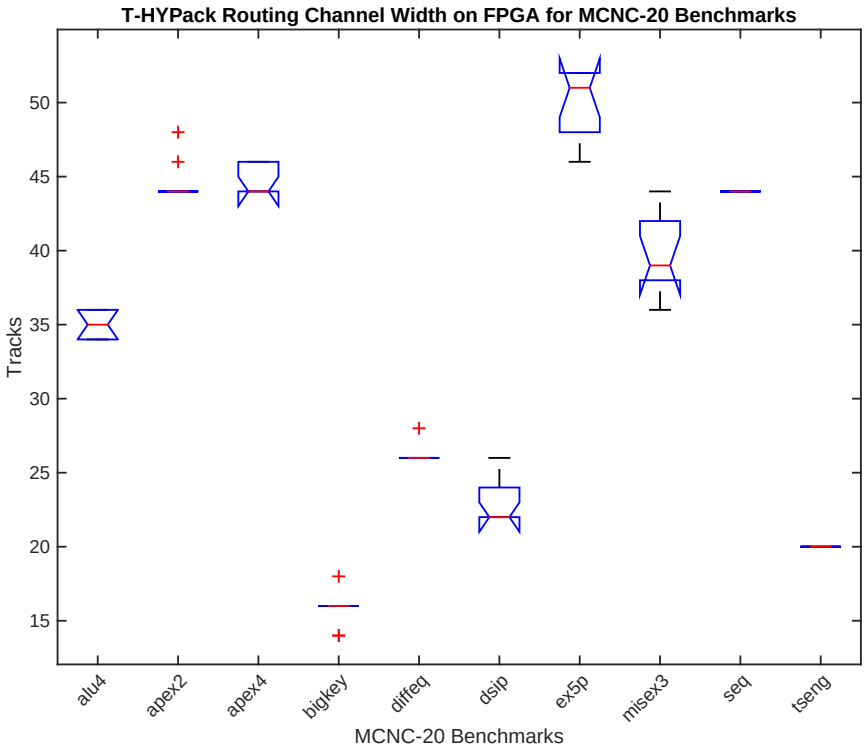


Figure A.36: Boxplot of T-HYPack on FPGA channel widths for MCNC-20 benchmarks

Table A.34: Medians of Figure A.36 – boxplot of T-HYPack on FPGA channel widths for MCNC-20 benchmarks

Benchmark	Median	Benchmark	Median
alu4	35	dsip	22
apex2	44	ex5p	51
apex4	44	misex3	39
bigkey	16	seq	44
diffeq	26	tseng	20

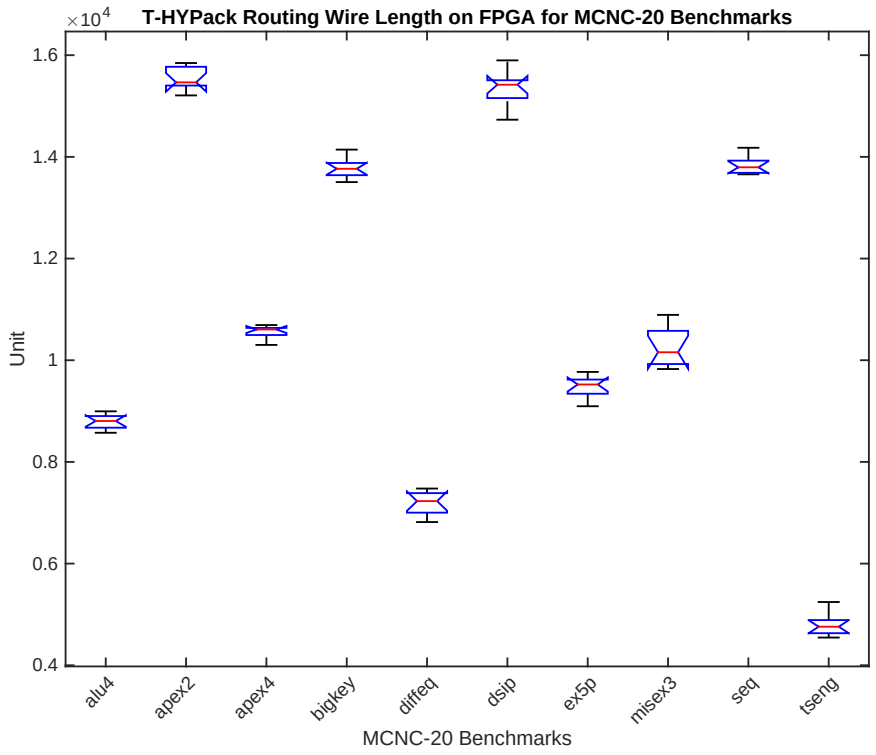


Figure A.37: Boxplot of T-HYPack on FPGA wire lengths for MCNC-20 benchmarks

Table A.35: Medians of Figure A.37 – boxplot of T-HYPack on FPGA wire lengths for MCNC-20 benchmarks

Benchmark	Median	Benchmark	Median
alu4	8,805	dsip	15,420
apex2	15,466	ex5p	9,523
apex4	10,605	misex3	10,157
bigkey	13,766	seq	13,795
diffeq	7,229	tseng	4,759

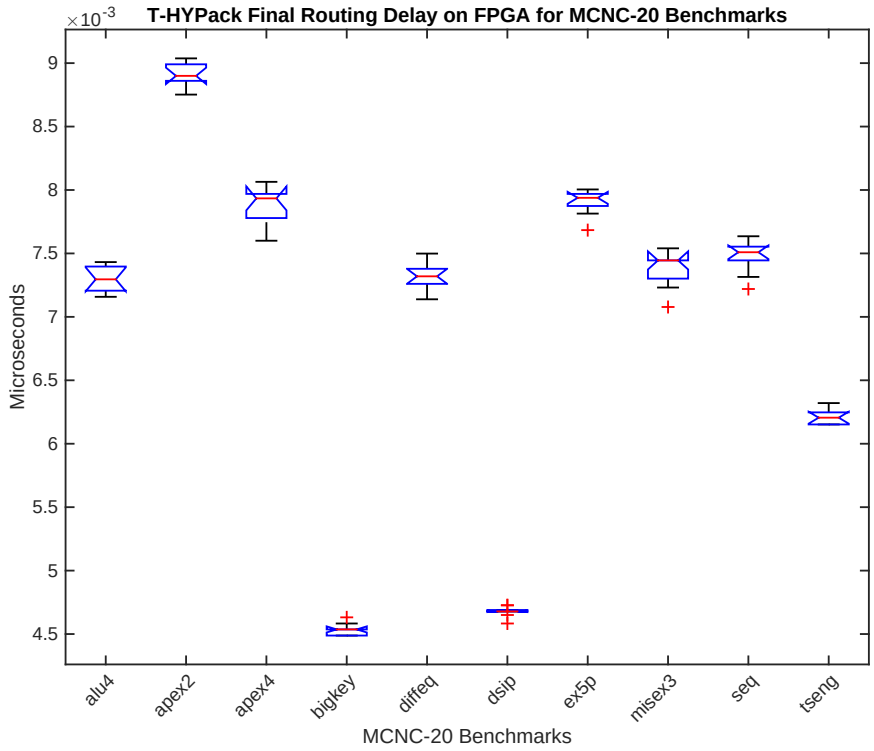


Figure A.38: Boxplot of T-HYPack on FPGA circuit-critical-path delays for MCNC-20 benchmarks

Table A.36: Medians of Figure A.38 – boxplot of T-HYPack on FPGA circuit-critical-path delays for MCNC-20 benchmarks

Benchmark	Median (nS)	Benchmark	Median (nS)
alu4	7.30	dsip	4.68
apex2	8.90	ex5p	7.94
apex4	7.93	misex3	7.45
bigkey	4.54	seq	7.51
diffeq	7.32	tseng	6.21

List of Abbreviations

ABC A System for Sequential Synthesis and Verification.

ADC Analog-to-Digital Converter.

AI Artificial Intelligence.

ALU Arithmetic Logic Unit.

ASIC Application-Specific Integrated Circuit.

BLE Basic Logic Element.

BLIF Berkeley Logic Interchange Format.

CAD Computer-Aided Design.

CH Channel Width.

CLB Configurable Logic Block.

CMOS Complementary Metal-Oxide Semiconductor.

CPU Central Processing Unit.

DAC Digital-to-Analog Converter.

DNA DesoxyriboNucleic Acid.

DSP Digital Signal Processor.

EC Evolutionary Computation.

EDA Electronic Design Automation.

ES Evolution Strategy.

FPAA Field Programmable Analogue Array.

FPGA Field Programmable Gate Array.

FPTA Field Programmable Transistor Array.

GA Genetic Algorithm.

GP Genetic Programming.

IC Integrated Circuit.

IDE Integrated Development Environment.

IIB Input Interconnect Block.

IO Input and Output.

LUT Look Up Table.

MAC Multiply-ACcumulate.

MCNC Microelectronics Center of North Carolina.

MIPS MIPS instruction set microprocessor – originally short for Microprocessor without Interlocked Pipeline Stages.

MO MultiObjective.

MOEA MultiObjective Evolutionary Algorithm.

MOGA MultiObjective Genetic Algorithm.

MUX MUltipleXer.

NP Non-deterministic Polynomial-time.

NPGA Niched-Pareto Genetic Algorithm.

NSGA Non-dominated Sorting Genetic Algorithm.

PAnDA Programmable Analogue and Digital Array.

PCB Printed Circuit Board.

PLD Programmable Logic Device.

PROM Programmable Read-Only Memory.

RAM Random-Access Memory.

RISC RISC instruction set microprocessor – Reduced Instruction Set Computing.

RNA RiboNucleic Acid.

SGA Simple Genetic Algorithm.

SGE Sun Grid Engine.

SIS A System for Sequential Circuit Synthesis.

SoC System on a Chip.

SPICE Simulation Program with Integrated Circuit Emphasis.

SRAM Static Random-Access Memory.

TSMC Taiwan Semiconductor Manufacturing Company.

VHDL VHSIC Hardware Description Language.

VHSIC Very High Speed Integrated Circuit.

VLSI Very-Large-Scale Integration.

VPR Versatile Placement and Routing.

References

(1985). Speial Session: Recent Algorithms for Gate-Level ATPG with Fault Simulation and their Performance Assessment. In *Int. Symp. Circuits and Systems*, pages 663–698. IEEE.

Actel Corp. (2009). ProASIC Flash Family FPGAs - Microsemi. `http://www.microsemi.com/document-portal/doc_download/131796-proasicplusgendes`. [Online; accessed 22-June-2015].

Aggarwal, A., Lewis, D. M., et al. (1994). Routing Architectures for Hierarchical Field Programmable Gate Arrays. In *Computer Design: VLSI in Computers and Processors, 1994. ICCD'94. Proceedings., IEEE International Conference on*, pages 475–478. IEEE.

Ahmed, E. and Rose, J. (2000). The Effect of LUT and Cluster Size on DeepSubmicron FPGA Performance and Density. In *Proc. IEEE Field Programmable Gate Arrays (FPGA)*, pages 3–12.

Alexander, M. J., Cohoon, J. P., Ganley, J. L., and Robins, G. (1994). An Architecture-independent Approach to FPGA Routing Based on Multi-weighted Graphs. In *Proceedings of the conference on European design automation*, pages 259–264. IEEE Computer Society Press.

Alexander, M. J. and Robins, G. (1996). New Performance-driven FPGA Routing Algorithms. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 15(12):1505–1517.

Alpert, C. J., Chan, T. F., Kahng, A. B., Markov, I. L., and Mulet, P. (1998a). Faster Minimization of linear Wirelength for Global Placement. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 17(1):3–13.

Alpert, C. J., Huang, J.-H., and Kahng, A. B. (1998b). Multilevel Circuit Partitioning. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 17(8):655–667.

Altera Corp. (2001). Altera FLEX 6000 Programmable Logic Device Family. `https://www.altera.com/literature/ds/archives/ds_ap2.pdf`. [Online; accessed 06-July-2015].

Altera Corp. (2002). APEX II Programmable Logic Device Family, DS-APEXII-3.0. `https://www.altera.com/literature/ds/archives/ds_ap2.pdf`. [Online; accessed 06-July-2015].

Altera Corp. (2003a). Altera FLEX 10K Embedded Programmable Logic Device Family. `https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/ds/dsf10k.pdf`. [Online; accessed 07-July-2015].

Altera Corp. (2003b). Altera FLEX 8000 Programmable Logic Device Family. `https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/ds/dsf8k.pdf`. [Online; accessed 07-July-2015].

Altera Corp. (2003c). APEX 20K Programmable Logic Device Family Data Sheet, DS-APEX20K-5.1. `http://www.altera.com/literature/ds/apex.pdf`. [Online; accessed 06-July-2015].

Altera Corp. (2003d). FLEX 10K Embedded Programmable Logic Device Family, DS-F10K-4.2. `http://www.altera.com/literature/ds/dsf10k.pdf`. [Online; accessed 06-July-2015].

Altera Corp. (2011a). Stratix II Device Handbook, Volume 1. `https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/stx2/stratix2_handbook.pdf`. [Online; accessed 06-July-2015].

Altera Corp. (2011b). Stratix III Device Handbook, Volume 1. `https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/stx3/stratix3_handbook.pdf`. [Online; accessed 06-July-2015].

Altera Corp. (2012). Cyclone III Device Handbook, Volume 1. `https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/cyc2/cyc2_cii5v1.pdf`. [Online; accessed 26-August-2015].

Altera Corp. (2015). Stratix V Device Handbook, Volume 1: Device Interfaces and Integration. https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/stratix-v/stx5_core.pdf. [Online; accessed 26-August-2015].

Anadigm Inc. (2003). AN121E04 AN221E04 Field Programmable Analog Arrays - User Manual. http://www.anadigm.com/_doc/um021200-u007.pdf. [Online; accessed 26-August-2015].

Bäck, T., Fogel, D. B., and Michalewicz, Z. (2000). *Evolutionary Computation 1*, volume 1. Insititute of Physics.

Bäck, T., Hammel, U., and Schwefel, H.-P. (1997). Evolutionary Computation: Comments on the History and Current State. *Evolutionary computation, IEEE Transactions on*, 1(1):3–17.

Bei, X., Chen, N., and Zhang, S. (2013). On the Complexity of Trial and Error. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 31–40. ACM.

Berkeley (1992). Berkeley Logic Interchange Format (BLIF). Technical report, University of California Berkeley.

Betz, V. (1998). *Architecture and CAD for Speed and Area Optimization of FPGAs*. PhD thesis, University of Toronto.

Betz, V. and Rose, J. (1997a). Cluster-Based Logic Blocks for FPGAs: Area-Efficiency vs. Input Sharing and Size. In *CICC*, pages 551–554.

Betz, V. and Rose, J. (1997b). VPR: A New Packing Placement and Routing Tool for FPGA Research. In *Int Workshop on Int. Workshop on Field-Programmable Logic and Application*, pages 213–222.

Betz, V. and Rose, J. (1998). How Much Logic Should Go in an FPGA Logic Block? *IEEE Des. Test*, 15(1):10–15.

Betz, V., Rose, J., and Marquardt, A. R. (1999). *Architecture and CAD for Deep-Submicron FPGAs*. Kluwer Academic Publishers, Norwell, MA, USA.

Bovet, D. P. and Crescenzi, P. (2006). *Introduction to the Theory of Complexity*, chapter 5: The class NP. Sapienza.

Bowman, K. A., Duvall, S. G., and Meindl, J. D. (2002). Impact of Die-to-die and Within-die Parameter Fluctuations on the Maximum Clock Frequency Distribution for Gigascale Integration. *Solid-State Circuits, IEEE Journal of*, 37(2):183–190.

Bozorgzadeh, E., Memik, S. O., Yang, X., and Sarrafzadeh, M. (2004). Routability-driven Packing: Metrics and Algorithms for Cluster-based FPGAs. *Journal of Circuits, Systems, and Computers*, pages 77–100.

Brayton, R. K., Hachtel, G. D., and Sangiovanni-Vincentelli, A. L. (1990). Multilevel Logic Synthesis. *Proceedings of the IEEE*, 78(2):264–300.

Brglez, F., Bryan, D., and Koźmiński, K. (1989). Combinational Profiles of Sequential Benchmark Circuits. In *Circuits and Systems, 1989., IEEE International Symposium on*, pages 1929–1934. IEEE.

Brown, S., Francis, R., Rose, J., and Vranesic, Z. (1992a). *Field-Programmable Gate Arrays*. VLSI, computer architecture and digital signal processing. Springer US.

Brown, S. and Rose, J. (1996). Architecture of FPGAs and CPLDs: A tutorial. *IEEE Design and Test of Computers*, 13(2):42–57.

Brown, S., Rose, J., and Vranesic, Z. G. (1992b). A Detailed Router for Field-Programmable Gate Arrays. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 11(5):620–628.

Carter, W., Duong, K., Freeman, R. H., Hsieh, H., Ja, J. Y., Mahoney, J. E., Ngo, L. T., and Sze, S. L. (1986). A User Programmable Reconfiguration Gate Array. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, pages 233–235. IEEE.

Chang, Y.-W., Thakur, S., Zhu, K., and Wong, D. (1994). A New Global Routing Algorithm for FPGAs. In *Proceedings of the 1994 IEEE/ACM international conference on Computer-aided design*, pages 356–361. IEEE Computer Society Press.

Chen, D. T., Vorwerk, K., and Kennings, A. (2007). Improving Timing-driven FPGA Packing with Physical Information. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 117–123. IEEE.

Chen, G. and Cong, J. (2004). Simultaneous Timing Driven Clustering and Placement for FPGAs. In *Field Programmable Logic and Application*, pages 158–167. Springer.

Chen, K.-C., Cong, J., Ding, Y., Kahng, A. B., and Trajmar, P. (1992). DAG-Map: Graph-based FPGA Technology Mapping for Delay Optimization. *Design & Test of Computers, IEEE*, 9(3):7–20.

Cong, J. and Ding, Y. (1994a). FlowMap: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-table Based FPGA Designs. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 13(1):1–12.

Cong, J. and Ding, Y. (1994b). On Area/Depth Trade-off in LUT-based FPGA Technology Mapping. *IEEE Trans. VLSI Syst.*, 2(2):137–148.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms, Third Edition, Chapter 16: Greedy Algorithm*. The MIT Press, 3rd edition.

Coyne, J. (2009). *Why Evolution Is True*. OUP Oxford.

Darwin, C. R. (1859). *On the Origin of Species*. John Murray, London.

Deb, K., Agrawal, S., Pratap, A., and Meyarivan, T. (2000a). A Fast Elitist Non-dominated Sorting Genetic Algorithm for Multi-objective Optimization: NSGA-II. *Lecture notes in computer science*, 1917:849–858.

Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T. (2002). A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. *Evolutionary Computation, IEEE Transactions*, 6:182–197.

Deb, K., Pratap, A., and Meyarivan, T. (2001). Constrained Test Problems for Multi-Objective Evolutionary Optimization. In *Evolutionary Multi-Criterion Optimization*, pages 284–298. Springer.

Deb, K., Pratap, A., and Moitra, S. (2000b). Mechanical Component Design for Multiple Objectives Using Elitist Non-dominated Sorting GA. In *Proceedings of the Parallel Problem solving from Nature VI Conference*, pages 859–868.

Dijkstra, E. W. (1959). A Note on Two Problems in Connexion with Graphs. *Numerische mathematik*, 1(1):269–271.

Donath, W. E. (1979). Placement and Average Interconnection Lengths of Computer Logic. *Circuits and Systems, IEEE Transactions on*, 26(4):272–277.

Dunlop, A. E. and Kernighan, B. W. (1985). A Procedure for Placement of Standard Cell VLSI Circuits. *IEEE Transactions on Computer-Aided Design*, 4(1):92–98.

Ebeling, C., McMurchie, L., Hauck, S., Burns, S., et al. (1995). Placement and Routing Tools for the Triptych FPGA. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 3(4):473–482.

E.Bozorgzadeh, Ogrenci-Memik, S., and Sarrafzadeh, M. (2001). RPack: Routability-driven Packing for Cluster-based FPGAs. In *Design Automation Conference, 2001. Proceedings of the ASP-DAC 2001. Asia and South Pacific*, pages 629–634. IEEE.

Edgar, L. J. (1930). Method and Apparatus for Controlling Electric Currents. US Patent 1,745,175.

Eiben, A. and Smith, J. (2007). *Introduction to Evolutionary Computing*. Natural Computing Series. Springer Berlin Heidelberg.

Elmore, W. (1948). The Transient Response of Damped Linear Networks with Particular Regard to Wideband Amplifiers. *Journal of applied physics*, 19(1):55–63.

Estrin, G. (1960). Organization of Computer Systems: The Fixed Plus Variable Structure Computer. In *Papers Presented at the May 3-5, 1960, Western Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM '60 (Western), pages 33–40, New York, NY, USA. ACM.

Estrin, G. and Viswanathan, C. R. (1962). Organization of a "Fixed-Plus-Variable" Structure Computer for Computation of Eigenvalues and Eigenvectors of Real Symmetric Matrices. *J. ACM*, 9(1):41–60.

Falkenauer, E. (1994). A New Representation and Operators for Genetic Algorithms Applied to Grouping Problems. *Evolutionary Computation*, 2(2):123–144.

Farooq, U., Parvez, H., Mehrez, H., and Marrakchi, Z. (2011). Exploration of Heterogeneous FPGA Architectures. *International Journal of Reconfigurable Computing*, 2011:2.

Feng, W. (2012). K-way Partitioning Based Packing for FPGA Logic Blocks without Input Bandwidth Constraint. In *Field-Programmable Technology (FPT), 2012 International Conference*, pages 8–15. IEEE.

Feng, W., Greene, J., Vorwerk, K., Pevzner, V., and Kundu, A. (2014a). Rent's Rule Based FPGA Packing for Routability Optimization. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, FPGA '14, pages 31–34, New York, NY, USA. ACM.

Feng, W., Greene, J., Vorwerk, K., Pevzner, V., and Kundu, A. (2014b). Rent's Rule Based FPGA Packing for Routability Optimization. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, FPGA '14, pages 31–34, New York, NY, USA. ACM.

Feng, W. and Kaptanoglu, S. (2008). Designing Efficient Input Interconnect Blocks for LUT Clusters Using Counting and Entropy. *ACM Trans. Reconfigurable Technol. Syst.*, 1(1):6:1–6:28.

Fiduccia, C. M. and Mattheyses, R. M. (1982). A Linear-time Heuristic for Improving Network Partitions. In *Proceedings of the 19th Design Automation Conference*, DAC '82, pages 175–181, Piscataway, NJ, USA. IEEE Press.

Fleming, J. A. (1905). Fleming Valve. US Patent 803,684.

Fogel, D. B. (1998). *Evolutionary Computation: the Fossil Record*. Wiley-IEEE Press.

Fogel, L., Owens, A., and Walsh, M. (1966). *Artificial Intelligence Through Simulated Evolution*. John Wiley & Sons.

Fonseca, C. M. and Fleming, P. J. (1993). Genetic Algorithms for Multiobjective Optimization: Formulation, Discussion and Generalization. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 416–423. S. Forrest, Ed.

Francis, R., Rose, J., and Vranesic, Z. (1991a). Chortle-crf: Fast Technology Mapping for Lookup Table-Based FPGAs. In *Proc, 28 th ACM/IEEE Design Automation Conference*, pages 248–251.

Francis, R. J., Rose, J., and Vranesic, Z. (1991b). Technology Mapping of Lookup Table-Based FPGAs for Performance. In *Computer-Aided Design, 1991. ICCAD-91. Digest of Technical Papers., 1991 IEEE International Conference on*, pages 568–571. IEEE.

Frankle, J. (1992). Iterative and Adaptive Slack Allocation for Performance-driven Layout and FPGA Routing. In *Proceedings of the 29th ACM/IEEE Design Automation Conference*, pages 536–542. IEEE Computer Society Press.

Fukunaga, A. S. and Korf, R. E. (2007). Bin completion algorithms for multicontainer packing, knapsack, and covering problems. *Journal of Artificial Intelligence Research*, pages 393–429.

Garey, M. R., Graham, R. L., and Ullman, J. D. (1973). An Analysis of Some Packing Algorithms. In *Combinatorial Algorithms*, pages 39–47, New York. Algorithmics Press.

GENCODYS (2010). Genetic and Epigenetic Networks in Cognitive Dysfunction – Background information on genetics. `http://www.gencodys.eu/Patient%20entry%20.php`. [Online; accessed 31-August-2015].

Gokhale, M. B. and Graham, P. S. (2005). *Reconfigurable Computing: Accelerating Computation with Field-Programmable Gate Arrays*, pages 11–36. Springer.

Goldberg, D. E. and Holland, J. H. (1988). Genetic Algorithms and Machine Learning. *Machine learning*, 3(2):95–99.

Greene, J., Roychowdhury, V., Kaptanoglu, S., and Gamal, A. E. (1991). Segmented Channel Routing. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pages 567–572. ACM.

Greenwood, G. W. and Tyrrell, A. M. (2007). *Introduction to Evolvable Hardware: A Practical Guide for Designing Self-Adaptive Systems (IEEE Press Series on Computational Intelligence)*. Wiley-IEEE Press.

Greiner, D., Galván, B., Periaux, J., Gauger, N., Giannakoglou, K., and Winter, G. (2003). *Evolutionary Methods for Design, Optimization and Control with Applications to Industrial and Societal Problems (EUROGEN 2003), Url = https://books.google.co.uk/books?id=XTZpBQAAQBAJ*. International Center for Numerical Methods in Engineering (CIMNE), Barcelona, Spain.

Greiner, D., Galván, B., Periaux, J., Gauger, N., Giannakoglou, K., and Winter, G. (2013). *Advances in Evolutionary and Deterministic Methods for Design, Optimization and Control in Engineering and Sciences (EUROGEN2013)*. Computational Methods in Applied Sciences. Springer International Publishing.

Gross, J. and Yellen, J. (2005). *Graph Theory and Its Applications, Second Edition*, chapter 1: Introduction to graph models. Textbooks in Mathematics. CRC Press.

Hitchcock, R., Smith, G., and Cheng, D. (1983). Timing Analysis of Computer-Hardware. *IBM Journal of Research and Developrnent*, pages 100–105.

Holland, J. H. (1973). Genetic Algorithms and the Optimal Allocation of Trials. *SIAM Journal on Computing*, 2(2):88–105.

Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. U Michigan Press.

Holland, J. H. (1992). *Adaptation in Natural and Artificial Systems*. MIT Press, Cambridge, MA, USA.

Horn, J., Nafploitis, N., and Goldberg, D. E. (1994). A Niched Pareto Genetic Algorithm for Multiobjective Otimization. In Michalewicz, Z., editor, *Proceedings of the First IEEE Conference on Evolutionary Computation,*, pages 82–87, Piscataway NJ. IEEE Press.

Huang, D. J.-H. and Kahng, A. B. (1997). Partitioning-based Standard-cell Global Placement with An Exact Objective. In *Proceedings of the 1997 international symposium on Physical design*, pages 18–25. ACM.

IEEE (1988). *Standard VHDL Language Reference Manual Std 1076-1987*. The Institute of Electrical and Electronics Engineers, New York, NY, USA.

IEEE (1995). *IEEE Standard Hardware Description Language Based on the Verilog(R) Hardware Description Language Std 1364-1995*. The Institute of Electrical and Electronics Engineers, New York, NY, USA.

Jacobi, W. (1952). Halbleiterverstärker. DE patent 833366 priority filing on April 14, 1949.

Jiménez, F. and Verdegay, J. L. (1998). Constrained Multiobjective Optimization by Evolutionary Algorithms. In *Proceedings of the International ICSC Symposium on Engineering of Intelligent Systems (EIS'98)*, pages 226–271.

Karypis, G. and Kumar, V. (1999). Multilevel K-way Hypergraph Partitioning. In *DAC '99 Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, pages 343–348. ACM.

Kernighan, B. and Lin, S. (1970). An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell Systems Technical Journal*, 49(2).

Khellah, M., Brown, S., and Vranesic, Z. (1993). Modelling Routing Delays in SRAM-based FPGAs. In *Canadian Conference on VLSI*, page 6B. Citeseer.

Kirkpatrick, S., Gelatt, C. D., Vecchi, M. P., et al. (1983). Optimization by Simulated Annealing. *Science*, 220(4598):671–680.

Kleinhans, J. M., Sigl, G., Johannes, F. M., and Antreich, K. J. (1991). GORDIAN: VLSI Placement by Quadratic Programming and Slicing Optimization. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 10(3):356–365.

Koza, J. R. (1990). Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems. Technical report, Stanford, CA, USA.

Koza, J. R. (1999). *Genetic Programming III: Darwinian Invention and Problem Solving*, volume 3. Morgan Kaufmann.

Kuehlmann, A. (2012). *The Best of ICCAD: 20 Years of Excellence in Computer-Aided Design*, chapter : ICCAD and Xilinx. Springer US.

Kuon, I. and Rose, J. (2006). Measuring the Gap Between FPGAs and ASICs. In *Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays*, FPGA '06, pages 21–30, New York, NY, USA. ACM.

Kuon, I., Tessier, R., and Rose, J. (2007). FPGA Architecture: Survey and Challenges. *Foundations and Trends in Electronic Design Automation*, 2(2):135–253.

Lancaster, D. (1974). *TTL Cookbook*. H. W. Sams.

Landman, B. S. and Russo, R. L. (1971). On a Pin Versus Block Relationship For Partitions of Logic Graphs. *Computers, IEEE Transactions*, C-20.

Langeheine, J. (2005). *Intrinsic Hardware Evolution on the Transistor Level*. PhD thesis.

Langeheine, J., Becker, J., Fölling, S., Meier, K., and Schemmel, J. (2001). A cmos fpta chip for intrinsic hardware evolution of analog electronic circuits. In *Evolvable Hardware, 2001. Proceedings. The Third NASA/DoD Workshop on*, pages 172–175. IEEE.

Lattice Corp. (2000). ispPAC10 In-System Programmable Analog Circuit. `http://pdf.datasheetcatalog.com/datasheets/480/254290_DS.pdf`. [Online; accessed 26-August-2015].

Lattice Corp. (2001a). ispPAC20 In-System Programmable Analog Circuit. http://www.farnell.com/datasheets/4632.pdf. [Online; accessed 26-August-2015].

Lattice Corp. (2001b). ispPAC30 In-System Programmable Analog Circuit. http://www.farnell.com/datasheets/8421.pdf. [Online; accessed 26-August-2015].

Lattice Corp. (2001c). ispPAC80 In-System Programmable Analog Circuit. http://www.latticesemi.com/~/media/LatticeSemi/Documents/DataSheets/ispPAC/ispPAC80DataSheet.PDF. [Online; accessed 26-August-2015].

Lattice Semi. Corp. (2007). LatticeXP Family Data Sheet. http://www.latticesemi.com/~/media/LatticeSemi/Documents/DataSheets/LatticeXP2/LatticeXPFamilyDataSheet.PDF. [Online; accessed 07-July-2015].

Lee, C. Y. (1961). An Algorithm for Path Connections and Its Applications. *Electronic Computers, IRE Transactions on*, (3):346–365.

Lee, Y.-s. and Wu, A. C. (1997). A Performance and Routability-driven Router for FPGAs Considering Path Delays. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 16(2):179–185.

Lemieux, G. G. and Brown, S. D. (1993). A Detailed Routing Algorithm for Allocating Wire Segments in Field-Programmable Gate Arrays. In *ACM-SIGDA Physical Design Workshop*.

Lemieux, G. G., Brown, S. D., and Vranesic, D. (1997). On Two-step Routing for FPGAs. In *Proceedings of the 1997 international symposium on Physical design*, pages 60–66. ACM.

Leventis, P., Chan, M., Chan, M., Lewis, D., Nouban, B., Powell, G., Vest, B., Wong, M., Xia, R., and Costello, J. (2003). Cyclone: A Low-cost, High-performance FPGA. In *In Proceedings of the IEEE 2003 CICC*, pages 49–52.

Lewis, D. (2003). The stratix^{TM} logic and routing architecture. In *Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field Programmable Gate Arrays*, pages 12–20. ACM.

Liu, H. and Akoglu, A. (2009). T-NDPack: Timing-driven Non-uniform Depopulation Based Clustering. In *Programmable Logic, 2009. SPL. 5th Southern Conference on*, pages 9–14. IEEE.

Luu, J., Kuon, I., Jamieson, P., Campbell, T., Ye, A., Fang, W. M., Kent, K., and Rose, J. (2011). VPR 5.0: FPGA CAD and Architecture Exploration Tools with Single-driver Routing, Heterogeneity and Process Scaling. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 4(4):32.

Marquardt, A. R. (1999). Cluster-Based Architecture, Timing-Driven Packing and Timing-Driven Placement for FPGAs. Master's thesis, University of Toronto.

Marquardt, A. S., Betz, V., and Rose, J. (1999). Using Cluster-based Logic Blocks and Timing-driven Packing to Improve FPGA Speed and Density. In *FPGA '99 Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, pages 37–46. ACM.

Marrakchi, Z., Mrabet, H., and Mehrez, H. (2005). Hierarchical FPGA Clustering Based on A Multilevel Partitioning Approach to Improve Routability and Reduce Power Dissapation. page 25. IEEE.

Maulik, U., Bandyopadhyay, S., and Mukhopadhyay, A. (2011). *Multiobjective Genetic Algorithms for Clustering, Applications in Data Mining and Bioinformatics*, chapter Genetic Algorithms and Multiobjective Optimization. Number 30-31. Springer.

May, R. M. et al. (1988). How Many Species Are There On Earth? *Science (Washington)*, 241(4872):1441–1449.

Mead, C. and Conway, L. (1978). *Introduction to VLSI systems*. Addison-Wesley Reading, MA.

Michalewicz, Z. (2013). *Genetic Algorithms + Data Structures = Evolution Programs*. Springer Science & Business Media.

Mishchenko, A., Chatterjee, S., and Brayton, R. K. (2007). Improvements to Technology Mapping for LUT-based FPGAs. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(2):240–253.

Moore, G. E. (1965). Cramming More Components onto Integrated Circuits. *Electronics*.

Moore, G. E. (2006). Moore's Law at 40. *Understanding Moore's law: four decades of innovation*.

Nag, S. K., Rutenbar, R., et al. (1998). Performance-driven Simultaneous Placement and Routing for FPGA's. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions*, 17(6):499–518.

Nagel, L. W. and Pederson, D. (1973). SPICE (Simulation Program with Integrated Circuit Emphasis). Technical Report UCB/ERL M382, EECS Department, University of California, Berkeley.

Oracle Corp. (2010). Oracle Grid Engine: An Overview. `http://www.oracle.com/technetwork/oem/host-server-mgmt/twp-gridengine-overview-167117.pdf`. [Online; accessed 16-August-2015].

Pareto, V. (1971 (1906)). *Manual of Political Economy (manuale di economia politica)*. Kelley, New York. Translated by Ann S. Schwier and Alfred N. Page.

Pavlov, A. and Sachdev, M. (2008). *CMOS SRAM Circuit Design and Parametric Test in Nano-Scaled Technologies: Process-Aware SRAM Design and Test*. Frontiers in Electronic Testing. Springer Netherlands.

Prado, D. F. G. (2006). Tutorial on FPGA Routing. *Electrónica-UNMSM*, (17):23–33.

QuickLogic Corp. (1998). QuickLogic 1997-98 Data Book. `http://cset.sp.utoledo.edu/cset4650oc/FPGA_Datasheets/quicklogic_pasic1_Family`. [Online; accessed 07-July-2015].

QuickLogic Corp. (2000). pASIC FPGA Families. `http://www.cryptomuseum.com/crypto/philips/vcard/files/QL_family.pdf`. [Online; accessed 07-July-2015].

QuickLogic Corp. (2005). pASIC 3 FPGA Family Data Sheet. `http://www.quicklogic.com/assets/pdf/data-sheets/pASIC3-Family-Data-Sheet.pdf`. [Online; accessed 07-July-2015].

Rajavel, S. T. and Akoglu, A. (2011). MO-Pack: Many-objective Clustering for FPGA CAD. In *Proceedings of the 48th Design Automation Conference*, pages 818–823. ACM.

Ray, T., Tai, K., and Seow, K. C. (2001). Multiobjective Design Optimization by An Evolutionary Algorithm. *Engineering Optimization*, 33(4):399–424.

Rechenberg, I. (1973). *Evolutionstrategie: Optimierung Technisher Systeme nach Prinzipien Des Biologischen Evolution*. Frommann-Holzboog, Stuttgart.

Riess, B. M. and Ettelt, G. G. (1995). Speed: Fast and Efficient Timing Driven Placement. In *Circuits and Systems, 1995. ISCAS'95., 1995 IEEE International Symposium on*, volume 1, pages 377–380. IEEE.

Rikizo, U. and Suzuki, H. (1974). *Explaining Biology II.* Bun-eido, Tokyo.

Rose, J. (1990). Parallel Global Routing for Standard Cells. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 9(10):1085–1095.

Rose, J. and Brown, S. (1991). Flexibility of Interconnection Structures for Field-Programmable Gate Arrays. *Solid-State Circuits, IEEE Journal*, 26(3):277–282.

Rubinstein, J., Penfield Jr, P., Horowitz, M., et al. (1983). Signal Delay in RC Tree Networks. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 2(3):202–211.

Sangiovanni-Vincentelli, A., Gamal, A. E., and Rose, J. (1993). Synthesis Method for Field Programmable Gate Arrays. *Proceedings of the IEEE*, 81(7):1057–1083.

Sechen, C. and Lee, K.-W. (1987). An Improved Simulated Annealing Algorithm for Row-based Placement. In *Proc. IEEE International Conference on Computer Aided Design*, pages 478–481.

Sechen, C. and Sangiovanni-Vincentelli, A. (1985). The TimberWolf Placement and Routing Package. *IEEE Journal of Solid-State Circuits*, 20(2):510–522.

Sentovich, E., Singh, K., Lavagno, L., Moon, C., Murgai, R., Saldanha, A., Savoj, H., Stephan, P., Brayton, R. K., and Sangiovanni-Vincentelli, A. L. (1992). SIS: A System for Sequential Circuit Synthesis. Technical Report UCB/ERL M92/41, EECS Department, University of California, Berkeley.

Sigl, G., Doll, K., and Johannes, F. M. (1991). Analytical Placement: A Linear or A Quadratic Objective Function? In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, pages 427–432. ACM.

Singh, A. and Marek-Sadowska, M. (2002). Efficient Circuit Clustering for Area and Power Reduction in FPGAs. In *FPGA '02 Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, pages 59–66. ACM.

Srinivas, M. and Patnaik, L. M. (1994). Genetic algorithms: A Survey. *Computer*, 27(6):17–26.

Srinivas, N. and Deb, K. (1995). Multiobjective Function Optimization Using Nondominated Sorting Genetic Algorithms. In *Evol. Comput.*, volume 2, pages 221–248.

Srinivasan, A. (1991). An Algorithm for Performance-driven Initial Placement of Small-cell ICs. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, pages 636–639. ACM.

Starr, C. (2007). *Biology: Concepts and Applications without Physiology*. Brooks/Cole biology series. Cengage Learning.

Stoica, A., Keymeulen, D., Zebulum, R., Thakoor, A., Daud, T., Tawel, R., and Duong, V. (2000). Evolution of Analog Circuits on Field Programmable Transistor Arrays. In *Evolvable Hardware, 2000. Proceedings. The Second NASA/DoD Workshop on*, pages 99–108. IEEE.

Sun, H., Yang, C., Lin, C., Pan, J., Snásel, V., and Abraham, A. (2014). *Genetic and Evolutionary Computing: Proceeding of the Eighth International Conference on Genetic and Evolutionary Computing (GECCO 2014)*. Advances in Intelligent Systems and Computing. Springer International Publishing, Nanchang, China.

Sun, W.-J. and Sechen, C. (1995). Efficient and Effective Placement for Very Large Circuits. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 14(3):349–359.

Swartz, W. and Sechen, C. (1995). Timing Driven Placement for Large Standard Cell Circuits. In *Design Automation, 1995. DAC'95. 32nd Conference on*, pages 211–215. IEEE.

Tessier, R. (1998). Negotiated A* routing for FPGAs. In *Proceedings of the 5th Canadian Workshop on Field Programmable Devices*, volume 6. Citeseer.

Tom, M., Leong, D., and Lemieux, G. (2006). Un/DoPack: Re-clustering of Large System-on-Chip Designs with Interconnect Variation for Low-cost FPGAs. In *Computer-Aided Design, 2006. ICCAD'06. IEEE/ACM International Conference on*, pages 680–687. IEEE.

Toole, G. and Toole, S. (1999). *Understanding Biology for Advanced Level*. New Understanding Series. Stanley Thornes.

Trefzer, M. A. and Tyrrell, A. M. (2015). *Evolvable Hardware From Practice to Application*. Springer, 1 edition.

Tsu, W., Macy, K., Joshi, A., Huang, R., Walker, N., Tung, T., Rowhani, O., George, V., Wawrzynek, J., and DeHon, A. (1999). HSRA: High-speed, Hierarchical Synchronous Reconfigurable Array. In *FPGA '99: Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, pages 125–134, New York, NY, USA. ACM.

Verel, S. (2015). Fitness Landscapes: the Metaphor and Beyond. `http://www-lisic.univ-littoral.fr/~verel/talks/tuto-fl-lion15.pdf`. [Online; accessed 11-Aug-2015].

Vygen, J. (2002). Basic Placement Algorithms. `http://www.or.uni-bonn.de/~vygen/files/Tutorial`. [Online; accessed 28-August-2015].

Weste, N. and Harris, D. (2010). *CMOS VLSI Design: A Circuits and Systems Perspective*, chapter 1, Introduction. Addison-Wesley Publishing Company, USA, 4th edition.

Wilton, S. J. E. (1997). *Architectures and Algorithms for Field-Programmable Gate Arrays with Embedded Memory*. PhD thesis, University of Toronto.

Wolpert, D. H. and Macready, W. G. (1997). No Free Lunch Theorems for Optimization. *Trans. Evol. Comp*, 1(1):67–82.

Wu, Y.-L. and Marek-Sadowska, M. (1995). Orthogonal Greedy Coupling: A New Optimization Approach to 2-D FPGA Routing. In *Proceedings of the 32Nd Annual ACM/IEEE Design Automation Conference*, DAC '95, pages 568–573, New York, NY, USA. ACM.

Wu, Y.-L. and Sadowska, M. M. (1994). An Efficient Router for 2-D Field Programmable Gate Array. In *European Design and Test Conference, 1994. EDAC, The European Conference on Design Automation. ETC European Test Conference. EUROASIC, The European Event in ASIC Design, Proceedings.*, pages 412–416. IEEE.

Xilinx Inc. (1985). XC2000 Logic Cell Array Families. `http://www.itisravenna.gov.it/sheet/XC2000FM.PDF`. [Online; accessed 06-July-2015].

Xilinx Inc. (1993). XC4000, XC4000A, XC4000H Logic Cell Array Families. `http://media.digikey.com/pdf/Data%20Sheets/Xilinx%20PDFs/XC4000,A,H.pdf`. [Online; accessed 07-July-2015].

Xilinx Inc. (1998). XC5200 Field Programmable Gate Arrays. http://www.xilinx.com/support/documentation/data_sheets/5200.pdf. [Online; accessed 07-July-2015].

Xilinx Inc. (2008). XtremeDSP for Virtex-4 FPGAs. http://www.xilinx.com/support/documentation/user_guides/ug073.pdf. [Online; accessed 27-August-2015].

Xilinx Inc. (2010). Virtex-4 Family Overview. http://www.xilinx.com/support/documentation/data_sheets/ds112.pdf. [Online; accessed 07-July-2015].

Xilinx Inc. (2011). Spartan-3 Generation FPGA User Guide. http://www.xilinx.com/support/documentation/user_guides/ug331.pdf. [Online; accessed 26-August-2015].

Xilinx Inc. (2012a). Virtex-5 FPGA User Guide, UG190 V5.4. http://www.xilinx.com/support/documentation/user_guides/ug190.pdf. [Online; accessed 06-July-2015].

Xilinx Inc. (2012b). Virtex-6 FPGA Configurable Logic Block User Guide, UG364 V1.2. http://www.xilinx.com/support/documentation/user_guides/ug364.pdf. [Online; accessed 26-August-2015].

Xilinx Inc. (2013). Virtex 2.5 Field Programmable Gate Arrays. http://www.xilinx.com/support/documentation/data_sheets/ds003.pdf. [Online; accessed 07-July-2015].

Xilinx Inc. (2014). 7 Series FPGA Configurable Logic Block User Guide, UG474 V1.7. http://www.xilinx.com/support/documentation/user_guides/ug364.pdf. [Online; accessed 26-August-2015].

Xilinx Inc. (2015a). MicroBlaze Soft Processor Core. http://www.xilinx.com/tools/microblaze.htm. [Online; accessed 22-June-2015].

Xilinx Inc. (2015b). Zynq®-7000 All Programmable SoCs. http://www.xilinx.com/publications/prod_mktg/zynq7000/Zynq-7000-combined-product-table.pdf. [Online; accessed 26-August-2015].

Yang, S. (1991). Logic Synthesis and Optimization Benchmarks, Version 3.0. Technical report, Microelectronics Center of North Carolina.

Zetex Corp. (1999). Totally Reconfigurable Analog Circuit – TRAC, TRAC020LH. `http://pdf.datasheetcatalog.net/datasheets/228/306843_DS.pdf`. [Online; accessed 26-August-2015].

Zhong, J., Hu, X., Gu, M., and Zhang, J. (2005). Comparison of Performance between Different Selection Strategies on Simple Genetic Algorithms. In *Computational Intelligence for Modelling, Control and Automation, 2005 and International Conference on Intelligent Agents, Web Technologies and Internet Commerce, International Conference*, volume 2, pages 1115–1121. IEEE.

Zitzler, E., Deb, K., and Thiele, L. (Summar 2000). Comparison of Multiobjective Evolutionary Algorithms: Empirical Results. *Evolutionary Computation*, 8(2):173–195.