

# A Genetic Parallel Programming based Logic Circuit Synthesizer

LAU, Wai Shing

A Thesis Submitted in Partial Fulfilment  
of the Requirements for the Degree of  
Master of Philosophy  
in  
Computer Science and Engineering

©The Chinese University of Hong Kong  
November 2006

The Chinese University of Hong Kong holds the copyright of this thesis. Any person(s) intending to use a part or whole of the materials in the thesis in a proposed publication must seek copyright release from the Dean of the Graduate School.



Thesis/Assessment Committee

Professor Ng Kam Wing (Chair)  
Professor Leung Kwong Sak (Thesis Supervisor)  
Professor Lee Kin Hong (Thesis Supervisor)  
Professor Wu Yu Liang, David (Committee Member)

Abstract of thesis entitled:

A Genetic Parallel Programming based Logic Circuit Synthesizer

Submitted by LAU Wai Shing

for the degree of Master of Philosophy

at The Chinese University of Hong Kong in November 2006

Genetic Parallel Programming (GPP) is a novel Genetic Programming paradigm. This thesis presents a GPP based Logic Circuit Synthesizer (GPPLCS) which is a combinational logic circuit learning system. GPPLCS can synthesize (evolve) optimal logic circuits on Field Programmable Gate Arrays (FPGAs) given the truth table of a circuit as an input. It employs a Multi Logic Unit Processor (MLP) which is a multiple instruction-stream multiple data-stream (MIMD), general-purpose register machine. Based on the parallel architecture of MLP, GPPLCS evolves genetic programs in parallel form (MLP programs).

The GPPLCS has been improved in two different ways. First of all, we make use of hardware accelerator in the GPPLCS. A Multi MLP based GPPLCS (MMGPPLCS) is proposed so that the whole evolution can be sped up. MMGPPLCS is designed to speed up the processes of both evolution and evaluation of genetic parallel programs that represent combinational logic circuits. Moreover, a hardware based MLP has been implemented in FPGAs. Experimental result shows that the speedups vary from 10 to 36 depending on applications. The second improvement is by making use of local search operators, FlowMap or DAOMap. By integrating GPPLCS and FlowMap, a Hy-

bridized GPPLCS (HGPPLCS) is developed. The HGPPLCS first evolves circuits in 2-input LookUp Table (LUT) circuit and then relies on FlowMap to give a 4-input LUT mapping solution. Experimental results show that both the LUT counts and the propagation LUT delays of the circuits collected are better than the original GPPLCS. In addition, by including DAOMap as a local search operator, a novel memetic algorithm has been developed and used in a Memetic GPPLCS (MGPPLCS). GPP is first used for evolving a population of LUT-based circuits. DAOMap is for optimization purpose while the GPP searches for the possible global optima locations (vicinity). DAOMap acts as a greedy local search operator to return an optimum circuit for each individual in the GPP population. GPP keeps on evolving and the process continues until some certain stopping criteria are met. Experimental results show that circuits found using this approach contain smaller number of LUTs and LUT levels compared with existing approaches with a smaller number of tournaments.

## 論文撮要

本論文旨在設計、實踐及改善一個全新遺傳平行程式編寫邏輯電路合成器 (GPPLCS)。它建基於一種新的遺傳平行程式編寫(GPP)。根據電路的真值表(truth table)，GPPLCS可以合成(演變)優化的邏輯電路給予可編程序閘矩陣晶片(FPGAs)運作。它採用了一個多算術邏輯單元處理器(MLP)，MLP 是一個多指令多數數據(MIMD)、一般性用途的處理器，基於MLP 的平行結構，GPP 以平行形程式(MLP 程式)來展開進化過程。

我們以兩種不同的改進方式改善 GPP 邏輯電路合成器。首先使用的是利用硬體加速，我們建議一個含有多個 MLP 的 GPP 邏輯電路合成器(MMGPPLCS)，MMGPPLCS 旨在加快基因演變和遺傳平行程式的評估。此外，放置到 FPGA 的 MLP 上運行能加快遺傳平行程式的評估速度。實驗結果表明有 10 至 36 倍的加速。另一種方式是在基因演算法配搭上一個局部搜尋(Local Search)演算法，如 FlowMap 或 DAOMap。混合了 FlowMap 的 GPPLCS (HGPPLCS) 是用 GPPLCS 在發展雙輸入 LookUp 表 (2-input LUT) 邏輯電路後，然後依靠 FlowMap 產生 4 輸入 LookUp 表(4-input LUT) 的邏輯電路。實驗結果表明，無論 LUT 的數目和邏輯電路的層數都比原來 GPPLCS 收集到的電路少。此外，把 DAOMap 作為一個局部搜尋的工具放進 GPPLCS 便成為了一個全新的 Memetic GPPLCS (MGPPLCS)。GPP 首先產生一組電路，然後 DAOMap 優化它們，GPP 會不斷搜索最佳地點的近鄰位。GPP 不斷演變，直到某些標準達到。試驗結果表明 MGPPLCS 採用這種方法可在較短的時間找到一些較少 LUT 的數目和邏輯電路的層數的電路。

# Acknowledgement

I would like to express my appreciation to my supervisors Professor K.S. Leung and Professor K.H. Lee for their invaluable advice and guidance during my research study. Besides, I would like to thank Professor K.W. Ng for his suggestions in my research.

During my Master of Philosophy study, I have also benefited from a lot of people. My peers in our Evolutionary Computation study group, Li Gang, Ar Ho, Ar Man, Dr. Liang Yong and Dr. Ivan Cheang, have given me a lot of invaluable comments and suggestions in my past 2 years.

Last but not least, I would like to express my gratitude to my family for their support and love throughout my life.

This work is dedicated to my parents.



# Contents

<b>Abstract</b>	<b>i</b>
<b>Acknowledgement</b>	<b>iv</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Field Programmable Gate Arrays . . . . .	2
1.2 FPGA technology mapping problem . . . . .	3
1.3 Motivations . . . . .	5
1.4 Contributions . . . . .	6
1.5 Thesis Organization . . . . .	9
<b>2 Background Study</b>	<b>11</b>
2.1 Deterministic approach to technology mapping problem . . . . .	11
2.1.1 FlowMap . . . . .	12
2.1.2 DAOMap . . . . .	14
2.2 Stochastic approach . . . . .	15
2.2.1 Bio-Inspired Methods for Multi-Level Com- binational Logic Circuit Design . . . . .	15
2.2.2 A Survey of Combinational Logic Circuit Representations in stochastic algorithms . . . . .	17
2.3 Genetic Parallel Programming . . . . .	20
2.3.1 Accelerating Phenomenon . . . . .	22
2.4 Chapter Summary . . . . .	23

<b>3</b>	<b>A GPP based Logic Circuit Synthesizer</b>	<b>24</b>
3.1	Overall system architecture . . . . .	25
3.2	Multi-Logic-Unit Processor . . . . .	26
3.3	The Genotype of a MLP program . . . . .	28
3.4	The Phenotype of a MLP program . . . . .	31
3.5	The Evolution Engine . . . . .	33
3.5.1	The Dual-Phase Approach . . . . .	33
3.5.2	Genetic operators . . . . .	35
3.6	Chapter Summary . . . . .	38
<b>4</b>	<b>MLP in hardware</b>	<b>39</b>
4.1	Motivation . . . . .	39
4.2	Hardware Design and Implementation . . . . .	40
4.3	Experimental Settings . . . . .	43
4.4	Experimental Results and Evaluations . . . . .	46
4.5	Chapter Summary . . . . .	50
<b>5</b>	<b>Feasibility Study of Multi MLPs</b>	<b>51</b>
5.1	Motivation . . . . .	52
5.2	Overall Architecture . . . . .	53
5.3	Experimental settings . . . . .	55
5.4	Experimental results and evaluations . . . . .	59
5.5	Chapter Summary . . . . .	59
<b>6</b>	<b>A Hybridized GPPLCS</b>	<b>61</b>
6.1	Motivation . . . . .	62
6.2	Overall system architecture . . . . .	62
6.3	Experimental settings . . . . .	64
6.4	Experimental results and evaluations . . . . .	66
6.5	Chapter Summary . . . . .	70
<b>7</b>	<b>A Memetic GPPLCS</b>	<b>71</b>
7.1	Motivation . . . . .	72
7.2	Overall system architecture . . . . .	72

7.3	Experimental settings . . . . .	76
7.4	Experimental results and evaluations . . . . .	77
7.5	Chapter Summary . . . . .	80
<b>8</b>	<b>Conclusion</b>	<b>82</b>
8.1	Future work . . . . .	83
	<b>Bibliography</b>	<b>85</b>

# List of Figures

1.1	General Model of an FPGA which consists of Configurable Logic Blocks (CLBs), Input Output Blocks (IOBs) and routing resources . . . . .	2
1.2	2-Slice Virtex-E CLB . . . . .	3
1.3	Schematic of a SRAM-based 3-LUT . . . . .	4
1.4	FPGA mapping example . . . . .	4
1.5	The system block diagram of the GPPLCS . . . . .	7
2.1	Label Calculation in FlowMap . . . . .	13
2.2	Label Calculation in FlowMap (Cont') . . . . .	14
2.3	The structure of Programmable Logic Devices . . . . .	18
2.4	The phenotype used in Cartesian GP . . . . .	19
2.5	Louis's Two-Dimensional Gate Array . . . . .	19
2.6	The phenotype proposed by Torresen . . . . .	20
2.7	The phenotype of $F^2$ PGA . . . . .	20
2.8	The framework of a GPP system [12] . . . . .	21
3.1	The system block diagram of GPPLCS . . . . .	25
3.2	The 2-LUT MLP used by the GPPLCS . . . . .	26
3.3	The 4-LUT MLP used by the GPPLCS . . . . .	27
3.4	The genotype of a $L_{MAX}$ -PI (PI[0]-PI[ $L_{MAX}$ -1]), 16-SI(SI[*],0)-SI[*],15)) MLP program . . . . .	28
3.5	Representations of SIs in evolving 2-LUT and 4-LUT circuits . . . . .	29
3.6	Functions $b0$ - $bF$ used in 2-LUT circuits SI . . . . .	30

3.7	The corresponding content of 4-LUT of the "bF6E0 r31 r27 r08 r29 r00" sub-instruction . . . . .	31
3.8	Optimized MLP program for 1-bit full adder in 2-LUT format . . . . .	32
3.9	A 1-bit full adder in 2-LUT format . . . . .	32
3.10	Optimized MLP program for 2-bit full adder in 4-LUT format . . . . .	33
3.11	A 2-bit full-adder in 4-LUT format . . . . .	33
3.12	PI level crossover on two parents . . . . .	36
3.13	An SI swapping in a single MLP program . . . . .	37
4.1	The architecture of the MLP core . . . . .	41
4.2	A Processing Element . . . . .	42
4.3	The speedup ratio versus tournaments for MUX problem . . . . .	47
4.4	The speedup ratio versus tournaments for ADD problem . . . . .	47
4.5	The speedup ratio versus tournaments for CMP problem . . . . .	48
4.6	The speedup ratio versus tournaments for PRI problem . . . . .	48
4.7	The speedup ratio versus tournaments for MAJ problem . . . . .	48
4.8	The speedup ratio versus tournaments for BCD problem . . . . .	49
5.1	The system block diagram of MMGPPLCS . . . . .	53
5.2	FIFO design . . . . .	54
5.3	Algorithm of MMGPPLCS in simulation . . . . .	57
6.1	HGPPLCS . . . . .	63
6.2	FlowMap refines the fitness of individuals in GP-PLCS . . . . .	64

6.3	Average number of 4-LUT count and LUT level collected from HGPPLCS and GPPLCS on the six problems in 50 runs . . . . .	69
6.4	Best number of 4-LUT and LUT level collected from HGPPLCS and GPPLCS on the six problems in 50 runs . . . . .	69
6.5	The best 3-bit comparator evolved by the HGPPLCS . . . . .	69
7.1	The system block diagram of MGPPLCS . . . . .	73
7.2	DAOMap refines the fitness of individuals in GPPLCS . . . . .	74
7.3	Algorithm of MGPPLCS . . . . .	75
7.4	6-bit multiplexer evolved by the MGPPLCS . . . . .	80

# List of Tables

3.1	Control-codes in 2-LUT circuits SI . . . . .	28
3.2	Control-codes in 4-LUT circuits SI . . . . .	30
4.1	Pilchard board features . . . . .	42
4.2	Six combinational logic circuit problems used in GPPLCS with the hardware assisted MLP. The $N_{in}$ and $N_{out}$ denote the numbers of inputs and outputs respectively. The $N_{row}$ ( $=2^{N_{in}}$ )denotes the number of rows in the truth tables . The $N_{case}$ ( $=N_{row} \times N_{out}$ )denotes the total number of training cases . . . . .	44
4.3	Experimental settings used in GPPLCS with the hardware assisted MLP . . . . .	45
4.4	Summary of experimental results in GPPLCS with hardware assisted MLP . . . . .	47
5.1	Six combinational logic circuit problems used in the simulation. The $N_{in}$ and $N_{out}$ denote the numbers of inputs and outputs respectively. The $N_{row}$ ( $=2^{N_{in}}$ )denotes the number of rows in the truth tables . The $N_{case}$ ( $=N_{row} \times N_{out}$ )denotes the total number of training cases . . . . .	56
5.2	Experimental settings used in MMGPPLCS and GPPLCS . . . . .	58

5.3	Number of tournaments ( $\times 10^6$ ) needed by MMGP-PLCS and GPPLCS in design phase on six problems (Average value) . . . . .	59
6.1	Six combinational logic circuit problems used in HGPPLCS. The $N_{in}$ and $N_{out}$ denote the numbers of inputs and outputs respectively. The $N_{row}$ ( $=2^{N_{in}}$ )denotes the number of rows in the truth tables . The $N_{case}$ ( $=N_{row} \times N_{out}$ )denotes the total number of training cases . . . . .	65
6.2	Experimental settings used in HGPPLCS . . . . .	67
6.3	Best circuits collected from HGPPLCS, GPPLCS and FlowMap algorithm on six problems . . . . .	68
6.4	Successful rate of evolving circuit problems in HGP-PLCS and GPPLCS . . . . .	68
7.1	Six combinational logic circuit problems used in MGPPLCS. The $N_{in}$ and $N_{out}$ denote the numbers of inputs and outputs respectively. The $N_{row}$ ( $=2^{N_{in}}$ )denotes the number of rows in the truth tables . The $N_{case}$ ( $=N_{row} \times N_{out}$ )denotes the total number of training cases . . . . .	76
7.2	Experimental settings used in MGPPLCS . . . . .	78
7.3	Best circuits collected from MGPPLCS, GPPLCS, DAOMap and FlowMap algorithm on six problems	79
7.4	Circuits collected from MGPPLCS, GPPLCS, DAOMap and FlowMap on six problems (Average value) . .	79



# List of Abbreviations

- ALU: Arithmetic Logic Units
- ACO: Ant Colony Algorithms
- CAD: Computer Aided Design
- CGP: Cartesian Genetic Programming
- CIGA: Case Injected Genetic Algorithms
- CLBs: Configurable Logic Blocks
- CU: Control Unit
- DSW: Dynamic Sample Weighting
- EE: Evolution Engine
- EHW: Evolvable Hardware
- ES: Evolutionary Strategy
- FF: Flip Flop
- $F^2$ PGA: Functional-based Field Programmable Gate Array
- FPGA: Field Programmable Gate Array
- GAs: Genetic Algorithms
- GASA: Genetic Algorithms with Simulated Annealing
- GPs: Genetic Programmings

- GPP: Genetic Parallel Programming
- GPPLCS: Genetic Parallel Programming based Logic Circuit Synthesizer
- HGPPLCS: Hybridized Genetic Parallel Programming based Logic Circuit Synthesizer
- ICs: Integrated Circuits
- IOBs: Input Output Blocks
- IORs: Internal Operand Registers
- $k$ -LoUs:  $k$ -input logic units
- LUT: LookUp Table
- $k$ -LUT:  $k$ -input LookUp Table
- MIMD: Multiple Instruction-streams Multiple Data-streams
- MLP: Multi Logic Unit Processor
- MGPPLCS: Memetic Genetic Parallel Programming based Logic Circuit Synthesizer
- MMGPPLCS: Multi MLP Genetic Parallel Programming based Logic Circuit Synthesizer
- OLMC: Output Logic Macro Cell
- PFU: Programmable floating-point processing units
- PE: Processing Element
- PI: Primary Input
- PIs: Parallel Instructions
- PLD: Programmable Logic Device

- PO: Primary Output
- PSO: Particle Swarm Optimization
- SGA: Simple Genetic Algorithms
- SIs: Sub Instructions
- SIR: Sub Instructions Registers
- VGA: Variable-length Genetic Algorithms
- VHDL: Very High Speed Integrated Circuit Hardware Description Language

# List of Symbols

- $d$ : the propagation delay
- $d_{max}$ : the maximum value allowed for propagation delay
- $f_{raw}$ : raw fitness
- $f_{dp}$ : fitness in the design phase
- $f_{op}$ : fitness in the optimization phase
- $g$ : the number of LookUp Table count (the number of normal sub-instructions(SI))
- $g_{max}$ : the maximum value allowed for number of LookUp Table count
- $L$ : length of Parallel Instructions (PIs)
- $L_{max}$ : Maximum length of Parallel Instructions
- $t_{max}$ : Maximum tournaments allowed
- $P_{xover}$ : PI crossover Probability
- $P_{btmut}$ : Bit Mutation Probability
- $P_{siswp}$ : SI swapping Probability
- $P_{sidel}$ : SI deletion Probability

# Chapter 1

## Introduction

Field Programmable Gate Arrays (FPGAs) have become very popular for prototyping new designs of digital logic circuits. This is because the FPGA implementation of a design is relatively easy, thus allowing logic verification to be performed early in the design process and reducing the turnaround time [62]. This has further ramifications on the manufacturing costs. In implementing a design in FPGAs, the optimized logic description obtained during logic synthesis must be mapped onto the modules and routing resources available on a particular FPGA. The objective is to find the best mapping, in terms of number of modules required, onto the FPGA. Other factors, such as performance, may also be considered. In this thesis, a synthesizer using genetic parallel programming (GPP) for FPGA technology mapping problem - a Genetic Parallel Programming based Logic Circuit Synthesizer (GPPLCS) is presented.

This chapter is organized as follows. An overview of the FPGA is given in Section 1.1. In Section 1.2, FPGA technology mapping problem is described. The motivations and our contributions can be found in Sections 1.3 and 1.4 respectively. Finally, the thesis organization is given in Section 1.5.

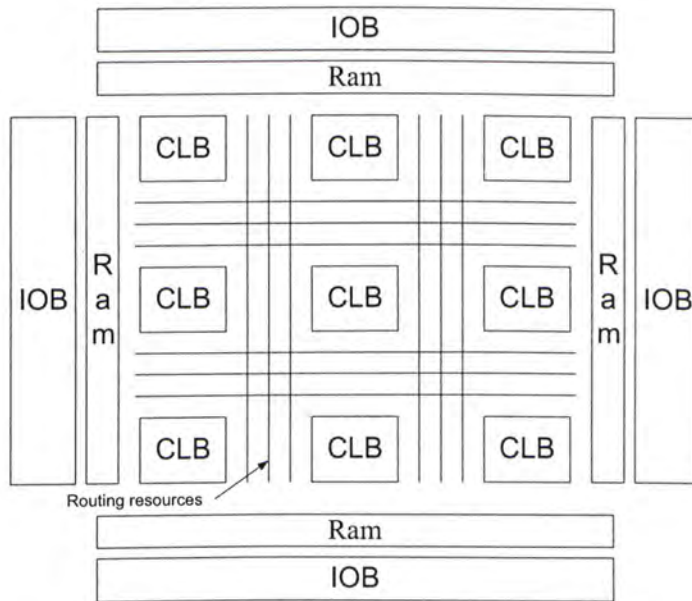


Figure 1.1: General Model of an FPGA which consists of Configurable Logic Blocks (CLBs), Input Output Blocks (IOBs) and routing resources

## 1.1 Field Programmable Gate Arrays

Field Programmable Gate Arrays (FPGAs) are a class of programmable hardware devices which consist of an array of Input Output Blocks (IOBs), Configurable Logic Blocks (CLBs) and routing resources. A simplified general model of an FPGA is shown in Figure 1.1. IOBs are responsible for connection between the CLBs logic and the outside world. A CLB is a basic unit of a logic function implementation in FPGAs. Routing resources interconnect the CLBs and form connections between the CLBs and the IOBs. Some FPGAs may also contain on-chip RAM. Figure 1.2 shows a 2-Slice Virtex-E CLB [2] which contains two logic cells. Each Logic Cell consists of a function generator in the form of a LookUp Table (LUT), a storage element or Flip Flop (FF), internal Carry and Control Logic and registers.

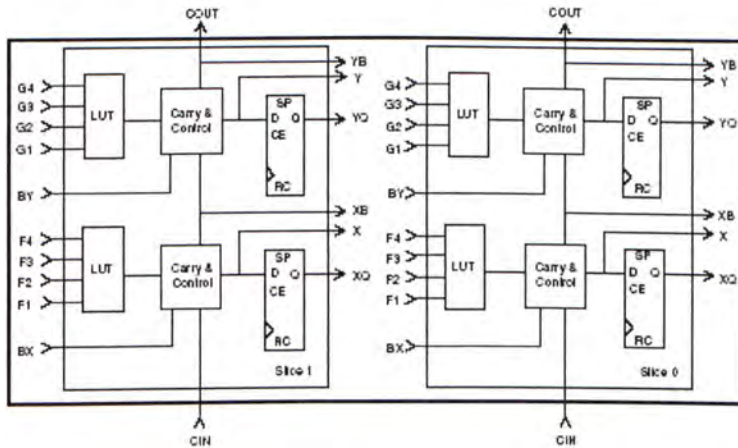


Figure 1.2: 2-Slice Virtex-E CLB

LUT-based FPGAs are a new generation of integrated circuit with an array of programmable logic blocks placed in an infrastructure of interconnections. Usually, fixed size LUTs are used among the whole FPGA chip and the size of every LUT is denoted by the number of inputs ( $k$ ), which is commonly chosen to be 4 or 5. A  $k$ -input LUT ( $k$ -LUT) can be used to implement any Boolean function of up to  $k$  variables. Every LUT is implemented by  $2^k$  memory cells with  $k$ -bit address decoder. Any inputs to a Boolean function will be taken as an address to read the corresponding bit pre-loaded inside the memory cell. Therefore, a  $k$ -LUT can be used to implement any  $k$ -variable Boolean functions. Figure 1.3 shows a possible structure of a 3-LUT.

## 1.2 FPGA technology mapping problem

A typical design flow for FPGAs consists of a number of steps. We first synthesize the logic circuit from specification and then follow by logic optimization. Then, it is followed by technology mapping and finally placement and routing. The aim of

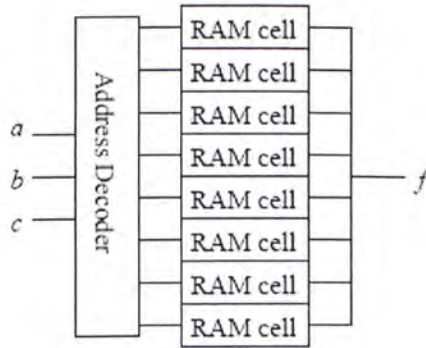


Figure 1.3: Schematic of a SRAM-based 3-LUT

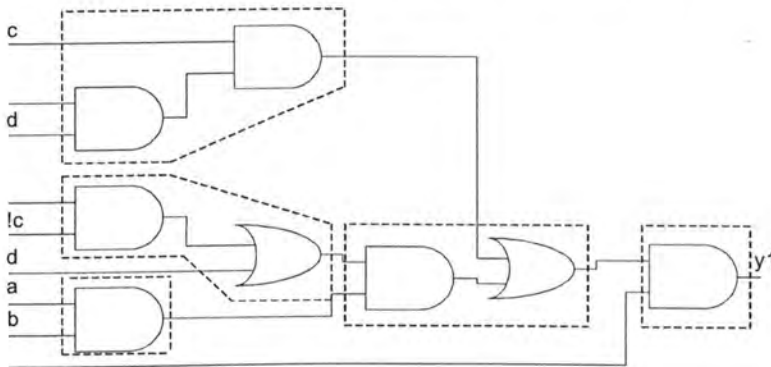


Figure 1.4: FPGA mapping example

FPGA technology mapping is to get a functionally equivalent LUT network based on a given Boolean circuit while placement and routing is to realize an implementation of the mapped LUT network. As a result, the objective of technology mapping is either to use a minimal chip area (i.e. area minimization) or to have a minimum circuit delay (i.e. depth minimization). The area is commonly indicated by the number of LUTs while the circuit delay is measured by the number of level of LUTs.

Our definition of the FPGA technology mapping problem is slightly different from the one used by the Computer Aided Design group. In their problem definition, the input to the FPGA



technology mapping problem is a Boolean Network which is modeled from a circuit. That means technology mapping applies on an existing circuit. We believe that any existing circuits would hinder our GPPLCS from reaching a global optimum. Thus, we used a different definition. Our input to GPPLCS is a truth table of a circuit. As the truth table specifies the functionality of a circuit only, circuits can be evolved freely in the GPPLCS. Thus, GPPLCS can be prevented from being trapped in a local optima.

The output of our GPPLCS would be a network which is composed of LUTs which performs the same function as stated in the input truth table. The number of inputs to LUT is bounded by a variable  $k$ . If the network is  $k$ -bounded, all inputs of LUTs will be less than or equal to  $k$ . Clearly,  $k$ -bounded network can be implemented by an FPGA using  $k$ -LUTs as logic block. Figure 1.4 shows an example on this problem. This example can be implemented by 5 LUTs.

The FPGA technology mapping problem is formulated as follows:

- INPUT: A truth table of a circuit
- OUTPUT: A  $k$ -bounded network
- Objectives:
  1. Minimize the number of LUTs used to map the circuit.
  2. Minimize the delay of the circuit mapping result.

### 1.3 Motivations

A Genetic Parallel Programming based Logic Circuit Synthesizer (GPPLCS) is proposed in this thesis. It is motivated by the following two observations:

1. Traditionally, technology mapping problems are solved by deterministic algorithms like FlowMap [21] and DAOMap [13]. Although mapping solutions can be obtained in a short period of time, the qualities of the solutions are not the best. The application of stochastic algorithms like Genetic Parallel Programming (GPP), which are particularly good at finding the global optimum to optimization problems, should be explored. Moreover, since GPP is a population-based search approach and has a strong optimization capability, it can find more of the best solutions among the possible solutions. That means more than one mapping solutions can be found by GPP.
2. Although GPP is good at locating the global optimum in optimization problems, GPP usually takes a long time for the computation. Some improvements are necessary to tackle this problem.

A GPPLCS is therefore proposed and implemented to tackle the first problem. Some further improvements are made to the GPPLCS. By having an hardware implementation of the GPPLCS in FPGAs is one of a feasible ways to solve the efficiency problem. The other way is to include a non-genetic deterministic local search operator in the GPPLCS. These improvements are shown to be effective in significantly shortening the computation time.

## 1.4 Contributions

Firstly, the major contribution of our work is the design and implementation of a GPPLCS. The GPPLCS is used to design optimized combinational logic circuits with LUTs, which are the basic logic representation components in FPGAs. Designing an optimized lookup-table network is a non-trivial task. Based on a

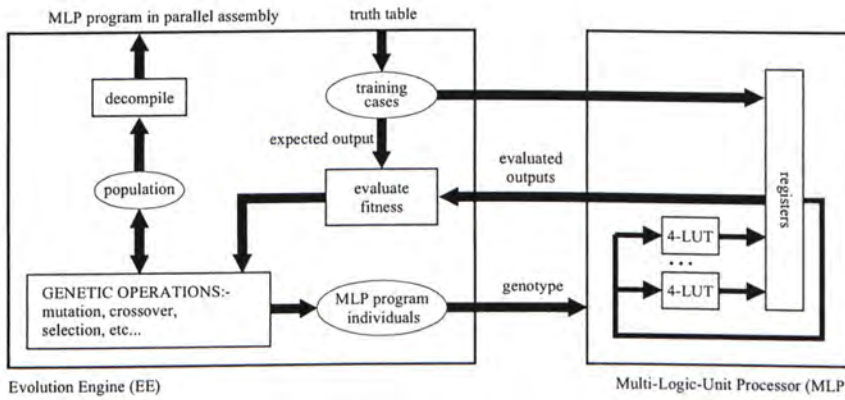


Figure 1.5: The system block diagram of the GPPLCS

taylor-made combinational logic evaluation engine, Multi Logic Unit Processor (MLP) and an Evolution Engine (EE) (see Figure 1.5), the GPPLCS successfully evolved high qualities multi-level combinational logic circuits. The results are superior to other existing Genetic Programmings (GPs) and Genetic Algorithms (GAs) systems.

Secondly, we have successfully built a hardware evaluation engine on FPGAs. Based on the architecture of the MLP, a hardware based MLP on FPGAs has been designed and implemented so that the evolution speed can be boosted. A GPPLCS with software version of the EE and the hardware based MLP were built to verify the effectiveness.

Thirdly, further improvements have been achieved on the GPPLCS with the hardware assisted MLP. First of all, we have investigated the possibility of full scale hardware implementation of GPPLCS. As the execution time of the MLP and the EE are different, a special model of cooperation between the MLP and the EE are necessary in a hardware implementation of the GPPLCS in an FPGA. By including multi MLP with a single EE in a GPPLCS, it can reduce the waiting time of EE during an evaluation of evolved combinational logic circuit in the MLP.

The simulation shows that the model works fine in evolving logic circuits and is suitable for the implementation of the GPPLCS in FPGAs.

Fourthly, we have included a local search operator in our GPPLCS. Based on existing deterministic algorithms for technology mapping problems such as FlowMap and DAOMap [13, 21], a Hybridized GPPLCS (HGPPLCS) and a Memetic GPPLCS (MGPPLCS) have been designed and implemented. The hybridized GPPLCS make use of the population-based Genetic Parallel Programming (GPP) and FlowMap to evolve 4-LUT circuits. Since GPP is population-based, it has a number of individuals (circuits) that have the same function (i.e. many-to-one genotype<sup>1</sup>-phenotype<sup>2</sup> mapping). Thus, GPP can provide a number of different circuits as inputs to the FlowMap algorithm. In this way, FlowMap can return different mapping solutions so that a better solution can be obtained.

Lastly, algorithms hybridize a non-genetic deterministic local search to refine the qualities of solutions with a genetic algorithm are called memetic algorithms [53]. This inspires an idea of using a local search operator in GPPLCS. By refining the individuals, local optima can be found more efficiently. During the process of evolution, DAOMap keeps refining individuals so that more and more optima can be explored. This new GPPLCS with a local search operator - DAOMap becomes our memetic GPPLCS. Experimental result shows that the memetic GPPLCS evolve better circuits using smaller number of tournaments.

Generally speaking, the memetic GPPLCS is the most efficient and effective method to generate circuits. It requires fewer evaluations to identify higher quality solutions than GPP. Both

---

<sup>1</sup>This is the representation which consists of encoded codes (chromosomes) for the phenotype

<sup>2</sup>The phenotype is the representation (as opposed to the genotype) which exhibits features that can be evaluated. The phenotype is the visible, behavioral expression of the genotype

the lookup table counts and the propagation delays of the circuits collected are better than those obtained by conventional design or evolved by GPP alone.

## 1.5 Thesis Organization

The rest of the thesis is organized as follows:

Chapter 2 first presents the research background of this thesis. Then, it gives a thorough review on both deterministic and stochastic algorithms to technology mapping problem. Afterwards, a brief introduction of Genetic Parallel Programming (GPP) will be given.

Chapter 3 presents a Genetic Parallel Programming based Logic Circuit Synthesizer (GPPLCS). GPPLCS is a GPP system which comprises two core components, a Multi-Logic-Unit Processor (MLP) and an Evolution Engine (EE). The MLP is an evaluation engine to execute parallel genetic programs for fitness evaluation. The EE is a population-based evolutionary process which manipulates the population and performs genetic operators.

Chapter 4 shows a design and implementation of a Multi Logic Unit Processor (MLP). The MLP is a hardware implementable evaluation engine to execute parallel genetic programs for fitness evaluation. With a cooperation of the software version EE and the hardware based MLP, combinational circuits are evolved at a faster rate. Experimental results in terms of actual speedup ratio on several combinational logic circuits are presented.

In Chapter 5, we describe a new model of cooperation between the MLP and the EE. This new model is designed for hardware implementation in FPGAs. The main contribution is to shorten the waiting time of EE during an evaluation of logic circuit programs in the MLP based on a pipeline concept. Sim-

ulation results on several combinational circuits compared with the current GPPLCS are presented.

Chapter 6 presents a hybridized GPPLCS. A system which integrates the GPPLCS and the FlowMap algorithm is presented. Experiments on several combinational logic circuits are presented.

Chapter 7 gives a presentation of a memetic GPPLCS. By including a non-genetic local search operator - DAOMap in GPPLCS, better circuits can be evolved with a smaller number of tournaments. Experimental result on several combinational logic circuits are given.

Finally, Chapter 8 concludes this thesis with a summary of the issues addressed in this thesis and their contributions. It also suggests several directions for future research in our GPPLCS.

---

□ End of chapter.

## Chapter 2

# Background Study

In Computer Aided Design (CAD) field, technology mapping problem is mainly tackled by deterministic algorithms. They are mainly network-flow-based algorithms which produce mapping solutions with optimal depth. Although there are no stochastic algorithms designed to tackle the technology mapping problem, some stochastic algorithms are designed for multi-level combinational logic circuit design.

This chapter is organized as follows. A literature review on two deterministic network-flow-based algorithms (FlowMap and DAOMap) is given in Section 2.1. Section 2.2 is a literature review on stochastic algorithms for multi-level combinational logic circuit design. Finally, a brief introduction of Genetic Parallel Programming is presented in Section 2.3.

### **2.1 Deterministic approach to technology mapping problem**

In this section, we introduce two network-flow-based algorithms for the technology mapping problem. These algorithms guarantee to produce mapping solutions with optimal depth. Therefore in the later design process, the wiring delays of the circuit are also optimized.

### 2.1.1 FlowMap

A circuit is modeled as a Boolean Network. There is a set of nodes PI representing the primary inputs (PIs) and another set of nodes PO representing the primary outputs (POs). All other nodes in the network are called internal nodes and these nodes are associated with specific functions. The function type of the internal nodes can be simple (AND, OR, NOT, XOR) or complex. Every wire in the circuit is represented by an edge between two nodes. All incoming edges to a node are called fanin of this node and all outgoing edges are called fanout; Nodes in PI has only fanouts while nodes in PO has only fanins. If the in-degrees of all nodes are less than or equal to  $k$ , the network is  $k$ -bounded. Clearly  $k$  bounded network can be implemented by an FPGA using  $k$ -input LookUp Tables ( $k$ -LUTs) as logic block.

FlowMap [21] is the first depth-optimal technology mapping algorithm developed. The algorithm will first apply Decompose Multi-Input Gate (DMIG) [14] to decompose the network into a network composed of small gates which have a smaller number of inputs (say 2). Experimental results show that small gates can be packed and grouped more efficiently than large input gates. The depth of the mapped network is the smallest when the original network was first decomposed into 2-input gates.

After gate decomposition, the algorithm enters the *labeling phase*. The algorithm calculate a label  $l(t)$  for every node  $t$  in topological order. The label  $l(t)$  gives the minimum depth of any mapping solution of the subnetwork rooted at node  $t$ , denoted by  $N_t$ . Moreover,  $l(t)$  is either equal to the maximum label  $p$  of the nodes in fanin of  $t$  or one more than the maximum label. FlowMap first collapses all nodes with label  $p$  in  $N_t$  to get a new network  $N'_t$ , then it continues to compute the maximum volume min-cut of  $N'_t$  using the classic network flow technique. If the cut size is less than or equal to  $k$ , the label  $l(t)$  is assigned to be  $p$ , otherwise  $l(t) = p+1$ , indicating a new LUT is used to



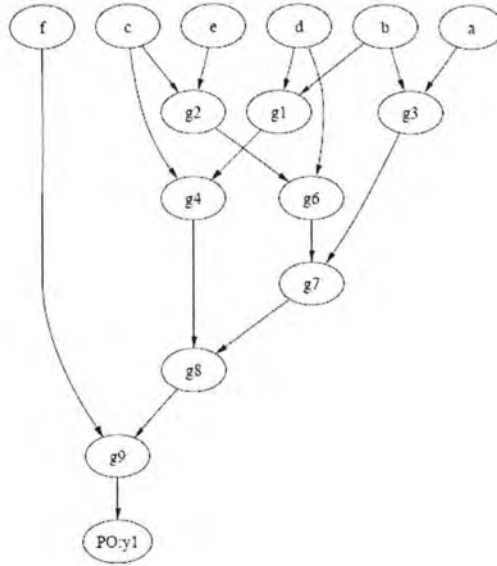


Figure 2.1: Label Calculation in FlowMap

map  $N_t$ .

After label calculation, FlowMap starts  $k$ -LUT generation with a list of PO nodes. It iteratively takes a non-PI nodes on the list and generate a LUT to implement the function for all the nodes with the same label. The fanins to this newly generated LUT is then put on the list.

To illustrate the label calculation we show the network for the circuit in Figure 2.1. There are 6 PIs (from  $a$  to  $f$ ) and 1 PO ( $y1$ ). For simplicity, we take  $k = 3$  (i.e. 3-input LUT). Suppose we need to compute the label for node  $g8$  with  $p = 2$  (i.e label of  $g7$  is 2,  $l(g7) = 2$ ) during the label phase. Thus we collapse the node  $g7$  with  $g8$  together and consider this collapsed node as the node *sink*. After addition of a dummy source node (*src*) connecting to all 5 PI nodes, we find a minimum cut on the network by network flow technique. Figure 2.2 shows the collapsed network and the graph for flow calculation. The min-cut simply separates the sink node with all the other nodes,

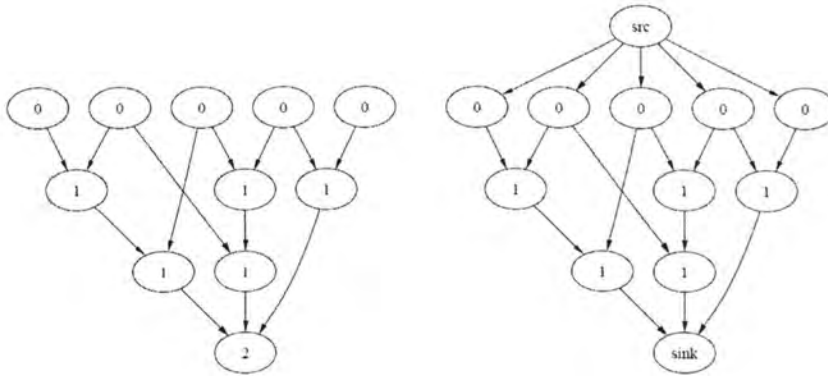


Figure 2.2: Label Calculation in FlowMap (Cont')

which implies that nodes  $g7$  and  $g8$  can be grouped together and implemented by a 3-LUT. Since the cut size equals to 3, the label of node  $g8$  is 2, same as that of  $g7$ .

FlowMap has a polynomial time complexity of  $O(kmn)$  where  $n$  and  $m$  are the number of nodes and the number of edges in  $N$ . Therefore the algorithm is extremely fast even for large circuits with thousands of gates.

### 2.1.2 DAOMap

DAOMap [13] which stands for Depth-optimal Area Optimization of FPGA designs is an extension of FlowMap. The difference lies in the way of modeling and controlling node duplications so as to reduce area through the entire mapping process. First, a cut-enumeration-based method that consists of cut generation and cut selection is adopted. Cut generation traverses the network from PIs to POs, and combines subcuts on the fanin nodes of the target node to generate all the cuts on the target node (each cut represents one possible LUT implementation rooted on the target node). After all the cuts are generated, the network from POs to PIs is traversed and cuts to produce the LUT mapping result is selected.

In order to reduce area through the entire mapping process, three novel approaches to effectively model and control node duplications and reduce area through the entire mapping process are done in DAOMap. First, the potential duplications during the cut generation procedure are considered so that the mapping solutions encoded in the cuts can consider duplication costs. This will help the cut selection procedure to make the right decisions to cover the circuit with less node duplications from a global optimization point of view. Second after the timing constraint is determined (the longest optimal mapping delay of the network), the noncritical paths will be relaxed by searching the solution space which will consider both local and global optimality information to minimize the mapping area. Third, an iterative cut selection procedure that further explores and perturbs the solution space is carried out to improve the solution quality.

## 2.2 Stochastic approach

Although there are no stochastic algorithms designed for tackling technology mapping problems, there are some related work on multi-level combinational logic circuit design by bio-inspired methods. In addition, there are many different existing phenotype representations for combinational logic circuits. They are described in the following subsections.

### 2.2.1 Bio-Inspired Methods for Multi-Level Combinational Logic Circuit Design

In this subsection, we summarize the current researches on bio-inspired methods for multi-level combinational logic circuit design.

- Simple Genetic Algorithms (SGA): It encodes a combina-

tional logic circuit by using a fixed-length genotype [15, 16, 17, 32, 49, 50, 55, 58, 59]. Standard genetic operators such as one-point crossover and bit mutation are used.

- Variable-length Genetic Algorithms (VGA). It is an extension of SGA [33, 34, 35]. A genotype only encodes the effective part of the architecture bits of a combinational logic circuit. Comparing with SGA, the lengths of VGA genotypes are smaller. Thus, it is possible to grow larger circuits in a shorter evolution time with VGA. Special genetic operators such as cut, splice [25] are used.
- Standard GP. It uses a tree structure to represent an individual combinational logic circuit [4, 40]. Standard GP operators such as node mutation, sub-tree mutation and sub-branch crossover are used. The main drawback of this method is that only single-output combinational logic circuits can be evolved. It is because there is only one root node in each program tree.
- Evolutionary Strategy (ES) are used to evolve combinational logic circuits [37, 52]. It includes five steps: 1) randomly initializes a population of  $\gamma$  genotypes; 2) evaluates all genotypes; 3) copies the fittest genotype into a new population; 4) fills the remaining  $\gamma - 1$  places in the new population by the mutated versions of the fittest genotype; and 5) replaces the old population by the new one. The algorithm repeats steps 2 to 5 until the termination criterion is achieved.
- Ant Colony Algorithms (ACO). ACO is used to evolve logic circuits [3, 20]. It is a multi-agent system in which interactions between low-level agents (ants) results in a meta-heuristic behavior of the whole ant colony [24].

- Particle Swarm Optimization (PSO). PSO is to evolve combinational logic circuits [19]. It simulates the movements of a flock of birds which seek for food (a global aim). It is a distributed algorithm that performs a multi-dimensional search [38].
- Genetic Algorithms with Simulated Annealing (GASA). It is a hybridization of a GA with Simulated Annealing (SA) [18, 39]. In this algorithm, the GA locates good regions of the search space whereas the SA exploits these good regions in order to find the optima.
- Case Injected Genetic Algorithms (CIGA). It combines a GA system with a Case-Based Reasoning (CBR) module [45, 46]. In the CBR, a case-base is built during GA search. Whenever the best individual is found, it will be stored in the case-base. The case-base can be reused to solve a new problem by injecting similar cases to the initial population of a new GA search.

### 2.2.2 A Survey of Combinational Logic Circuit Representations in stochastic algorithms

Most of the existing phenotype representations for combinational logic circuits adopt two-dimensional geometric structures. This subsection presents five typical geometries proposed and used by different groups of researchers. They are:

- Programmable Logic Device (PLD) Structure. PLD structure is used to evolve logic circuits [29]. PLD is a class of reprogrammable logic devices, e.g. GAL16V8. Each PLD consists of a fused array and an Output Logic Macro Cells (OLMC) (see Figure 2.3). A fused array can be programmed to represent minterms of a Boolean function. Multiple minterms are connected to an OLMC in which

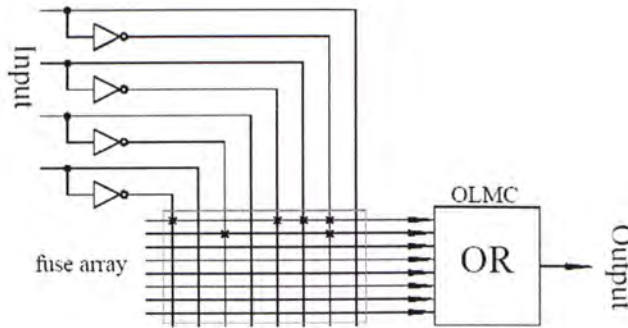


Figure 2.3: The structure of Programmable Logic Devices

a multi-input OR gate is configured. This phenotype is designed to match the architecture bits of PLDs in a sum-of-products form.

- Cartesian GP (CGP) [51]. As shown in Figure 2.4, the phenotype is a two-dimensional array of cells. Each cell contains a logic gate with some inputs and outputs. All external inputs and gate outputs can be reused by their higher level (right-hand side) cells. The final outputs can be connected to any external inputs and/or cell outputs in any levels. A levels-back parameter is used to limit the maximum number of levels that a cell output can be reused by its higher level cells.
- Louis's Two-Dimensional Gate Array. It is a two-dimensional gate array proposed by Louis [17, 45] (see Figure 2.5). The phenotype is a two-dimensional array of two-input logic gates. Except the first level gates (the left-most column in the figure), a gate  $G[i,j]$  gets its upper input from  $G[i,j.1]$  and lower input from either  $G[i.1,j.1]$  or  $G[i+1,j.1]$ . The outputs of the circuit are always connected to the outputs of the highest level gates (the right-most column in the figure). This representation reduces the genotype length by

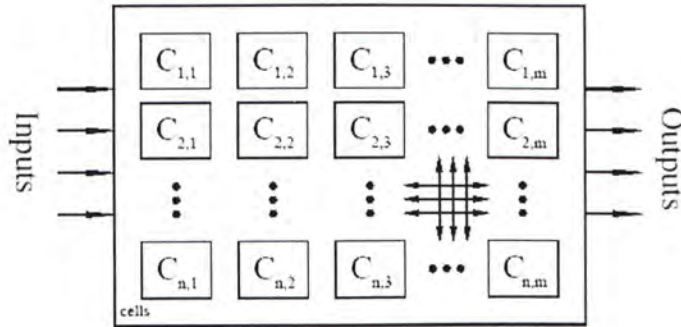


Figure 2.4: The phenotype used in Cartesian GP

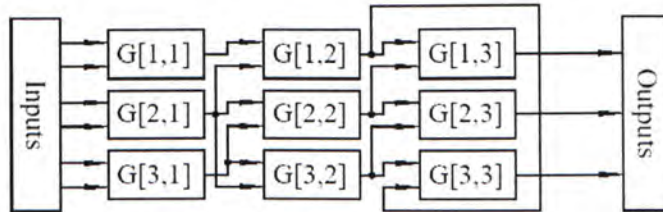


Figure 2.5: Louis's Two-Dimensional Gate Array

restricting the connectivity of a circuit.

- **Torresen's Two-Dimensional Gate Array.** Another two-dimensional gate array is proposed by Torresen [58] (see Figure 2.6). It relaxes the restrictions imposed on Louis's phenotype. A gate's input can be connected to any gate output in its previous layer.
- **The Function-Based FPGA ( $F^2$ PGA).** It is a function-level Evolvable Hardware (EHW) proposed by Murakawa [54] (see Figure 2.7). It is used to evolve hardware solutions for calculation intensive applications such as digital signal processing and data compression [54]. In an  $F^2$ PGA, there are multiple layers of programmable floating-point processing units (PFUs) that can perform different high-

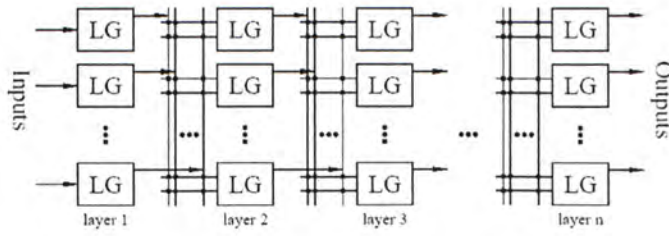


Figure 2.6: The phenotype proposed by Torresen

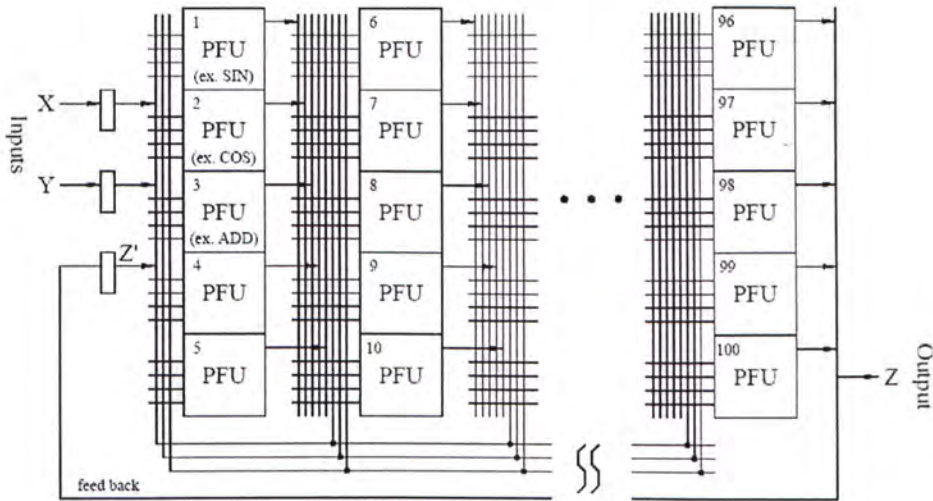


Figure 2.7: The phenotype of  $F^2$ PGA

level mathematic functions (e.g. sine, cosine, etc.). The architecture of  $F^2$ PGA is similar to the Torresen's one. The main difference is that  $F^2$ PGA shares all external inputs to all PFUs in all layers.

### 2.3 Genetic Parallel Programming

In this section, a brief introduction about Genetic Parallel Programming will be given.

Genetic Programming (GP) [31] is a robust method in Evo-



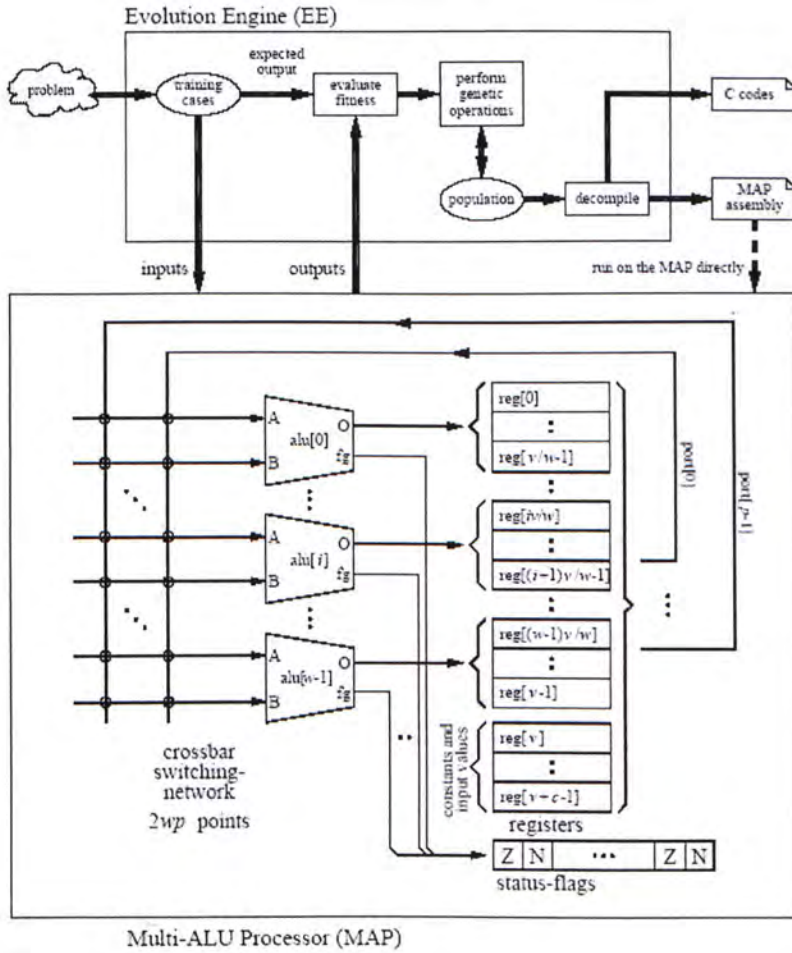


Figure 2.8: The framework of a GPP system [12]

lutionary Computation. There are many streams in GP like graph-based GP, stack-based GP, Cartesian GP, linear-tree and linear-graph GP and grammar-based GP. The two main streams in GP are standard GP [40] and linear-structured GP (linear GP) [6]. In standard GP, a genetic program is represented in a tree structure. In linear GP, a genetic program is represented in a linear list of machine code instructions or high-level language statements. A linear genetic program can be run on a target machine directly without performing any translation process.

The Genetic Parallel Programming (GPP) paradigm proposed by Cheang et. al. [43] is developed on the basis of linear GP. GPP is a novel linear GP paradigm that evolves parallel programs of a Multiple Instruction-streams Multiple Data-streams (MIMD) architecture with multiple Arithmetic-Logic-Units (ALU). A genetic parallel program consists of a sequence of parallel-instructions. A parallel-instruction comprises multiple sub-instructions that can perform multiple operations simultaneously in an execution step. GPP has been used to evolve compact parallel programs for different problems, such as numeric function regression [43] and data classification problems [9]. Figure 2.8 shows the framework of a GPP system. It consists of two components, a Multi Logic Unit Processor (MLP) and an Evolution Engine (EE). The MLP is an execution engine for genetic program fitness evaluation. The EE manipulates the population of genetic programs, performs genetic operators such as mutation and crossover and decompiles the solution program to symbolic assembly and high-level language codes. The details of the MLP and the EE are presented in the subsequent section.

### 2.3.1 Accelerating Phenomenon

Experimental results show that GPP can evolve wide programs (more sub-instructions within a parallel-instruction) more ef-

ficiently than narrow programs (less sub-instructions within a parallel-instruction). It is called the GPP accelerating phenomenon [44]. This phenomenon is particularly important and necessary. Having more sub-instructions within a parallel-instruction means that circuits can be evolved by GPP with a smaller depth level and smaller number of lookup tables. As a result, a Genetic Parallel Programming based Logic Circuit synthesizer (GPPLCS) can be developed based on GPP.

## 2.4 Chapter Summary

This chapter has given a literature review on two deterministic network-flow-based algorithms, i.e. FlowMap and DAOMap which are popular among Computer Aided Design community. Moreover, a literature review on stochastic algorithms for multi-level combinational logic circuit design as well as five different phenotype representation of combinational logic circuit design. Finally, a brief introduction of Genetic Parallel Programming have been presented.

---

□ End of chapter.

## Chapter 3

# A GPP based Logic Circuit Synthesizer

In this chapter, a Genetic Parallel Programming based Logic Circuit Synthesizer System (GPPLCS) is presented [10, 11, 12]. There are two main cores, the Evolution Engine (EE) and the Multi Logic Unit Processor (MLP). The EE manipulates the population of genetic programs, performs genetic operators such as mutation and crossover. The MLP is an execution engine for genetic program fitness evaluation. Variable-length parallel program structure (MLP program) is used to represent combinational logic circuits in order to preserve introns in the early stage. Circuits are evolved by a dual-phase approach. The first phase is called design phase. GPPLCS aims at finding a 100% functional program. Only functional correctness of the genetic programs are taken into consideration in this stage. Other qualitative factors like LookUp Table (LUT) count, propagation delay and program size are not considered. Once a first correct genetic program is found by the GPPLCS, we proceed to the second phase, optimization phase. Another set of genetic operators together with an optimization-oriented fitness function are used to improve the qualities of the correct program.

This chapter is organized as follows. The overall architecture of GPPLCS is described in Section 3.1. A detailed description

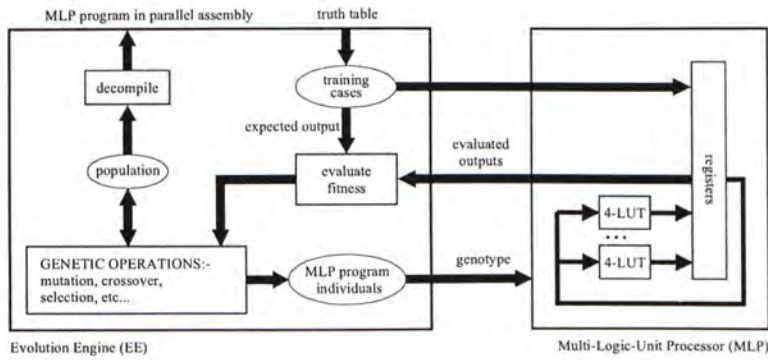


Figure 3.1: The system block diagram of GPPLCS

of the MLP is presented in Section 3.2. Then, both the genotype and phenotype of MLP program are discussed in Section 3.3 and 3.4. It is followed by a detailed description of the EE in Section 3.5. Finally, a chapter summary is given in Section 3.6.

### 3.1 Overall system architecture

Genetic Parallel Programming (GPP) is a linear GP paradigm that evolves parallel programs based on the MLP. Thus, parallel programs evolved are called MLP programs. GPPLCS is developed based on the GPP. It is a logic circuit synthesizer designed for tackling technology mapping problem by a stochastic approach. It first takes a truth table of a circuit (training cases) as an input. The output is a mapping solution to the circuit in the LUT format. Although numbers of inputs to the LUT can be varied, they are chosen to be either 2 or 4. All combinational digital circuits presented are evolved by a two-stage (i.e. design and optimization stages) approach. Different sets of genetic operators including crossover, bit mutation and sub-instruction swapping are used in different stages. In the design stage, the GPPLCS system aims at finding a 100% functional program (correct program). The raw fitness is given by the ratio

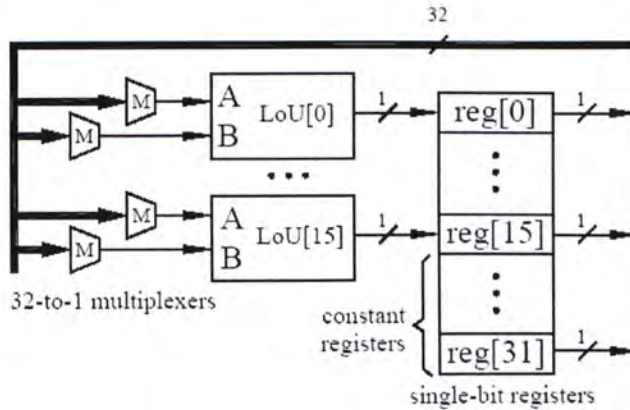


Figure 3.2: The 2-LUT MLP used by the GPPLCS

of unsolved training cases. In the optimization stage, the raw fitness then puts emphasis on the LUT count, the propagation delay and the program length. In other words, the major objective of the optimization stage is reducing the LUT count and then the propagation delay.

GPPLCS consists of two components, the EE and the MLP. The EE manipulates the genetic parallel programs and performs genetic operations. The MLP evaluates the genetic parallel programs to determine their fitness. Figure 3.1 shows the system block diagram of GPPLCS. The details of the EE and the MLP are presented in the subsequent sections.

### 3.2 Multi-Logic-Unit Processor

The MLP used in the GPPLCS is a general-purpose, tightly coupled processor. It is used for executing Boolean circuits evolved in GPPLCS (i.e evaluation of genetic program in GPPLCS). Since GPPLCS can evolve circuits in either 2-input LUT (2-LUT) format or 4-input LUT (4-LUT) format, the architecture of MLP is problem specific. The difference lies on the  $k$ -input logic units ( $k$ -LoUs).

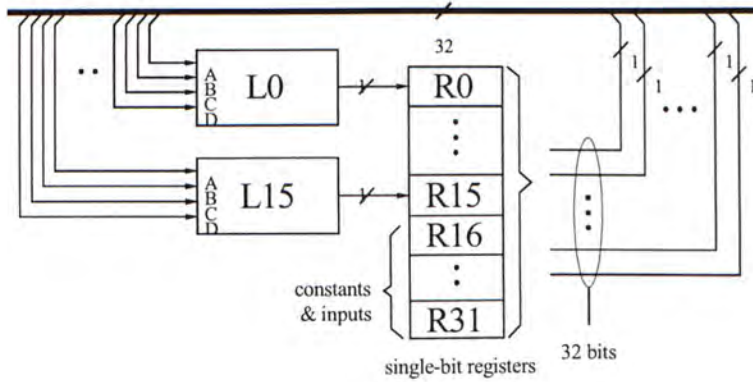


Figure 3.3: The 4-LUT MLP used by the GPPLCS

The MLP designed for evaluating circuits in 2-LUT format (2-LUT MLP) is shown in Figure 3.2. It consists of 16-LoUs (L0-L15), 16 variable registers (reg[0]-reg[15]) and 16 constant registers (reg[16]-reg[31]).

In the MLP, variable registers store intermediate values and program outputs; and constant registers store program inputs and constants. Each variable register can only be modified by a dedicated LoU (as shown in Figure 3.2, LoU[i] writes to reg[i] only). Constant registers are preloaded by EE before execution of an MLP program. In each processor clock cycle, multiple LoUs take input values from registers and perform Boolean operations concurrently. Then, all LoUs write single-bit results to their corresponding output variable registers. For example, the 2-LUT MLP shown in Figure 3.2 can perform up to 16 different operations concurrently, and 16 intermediate results can be carried forward to the subsequent parallel-instructions through the variable registers.

Figure 3.3 shows the MLP designed for evaluating circuits in 4-LUT format (4-LUT MLP). Similarly, the MLP consists of 32 registers. R0-R15 are variable registers that store intermediate values and program outputs while R16-R31 are read-only regis-

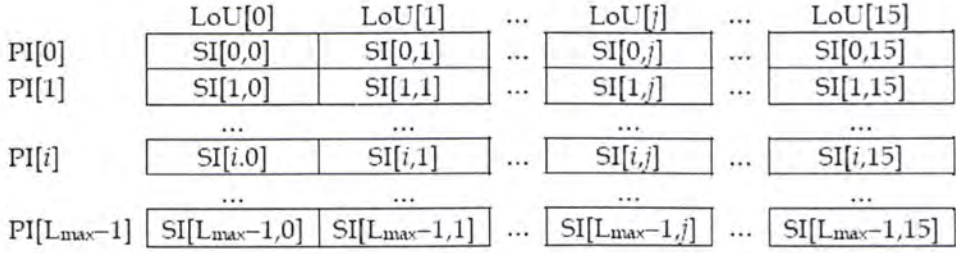


Figure 3.4: The genotype of a  $L_{MAX}$ -PI (PI[0]-PI[ $L_{MAX}$ -1]), 16-SI(SI[\*],0)-SI[\*],15) MLP program

Table 3.1: Control-codes in 2-LUT circuits SI

fields	number of bits	encoding
function opcode	5	00000 - 01111 = $b0 - bF$ (see Figure 3.6) 10000 - 11111 = <i>no operation, nop</i>
operand A	5	00000 - 11111 = input[0] - input [31]
operand B	5	00000 - 11111 = input[0] - input [31]
Total	15	

ters that store program inputs and logic constants. A variable register can only be modified by a dedicated 4-LoUs (e.g. L0 can write to R0 only). 16 4-LoUs (L0-L15) perform logic operations. EE will preload the program inputs and the constants into the read-only registers before a parallel program is executed.

### 3.3 The Genotype of a MLP program

The individual representation of GPPLCS includes a sequence ( $L_{MAX}$ ) of parallel instructions (PIs). In each PI, there are 16 sub-instructions (SIs). Figure 3.4 shows the genotype of an MLP program. The choice of  $L_{MAX}$  depends on the problem difficulty. Normally, it is set to 25. Figure 3.5 shows the representation of SIs.



SI used in evolving 2-LUT circuits	5-bit opcode	5-bit operand	5-bit operand		
SI used in evolving 4-LUT circuits	17-bit opcode	5-bit operand	5-bit operand	5-bit operand	5-bit operand

Figure 3.5: Representations of SIs in evolving 2-LUT and 4-LUT circuits

Theoretically, GPPLCS can evolve circuits with any number of inputs of LUTs. The difference only lies on the encoding. Since GPPLCS currently evolves circuits in either 2-LUT or 4-LUT format, encoding methods of SIs used are slightly different as each SI is used to resemble a LUT. For 2-LUT circuits, each SI consists of a 5-bit opcode (encoding at most 32 functions) and two 5-bit operands (encoding 32 choices of different inputs) (see Table 3.1). Since there are 16 SIs in a PI, a total of 240 bits  $((5+5+5) \times 16)$  are used to encode a parallel-instruction. If  $L_{MAX}$  is chosen to be 25 (25 PIs), the genotype may contain up to 6,000  $(240 \times 25)$  bits.

For 4-LUT circuits, each SI consists of a 17-bit opcode and four 5-bit operands (see Table 3.2). The Boolean function of each SI is denoted by a four-digit hexadecimal number which represents the 16-bit memory contents of the 4-LUT. For example, the SI with opcode bF6E0 means loading "0000 0111 0110 1111" to the corresponding 4-LUT which can be treated as a 16 to 1 multiplexer. The content of the corresponding 4-LUT is shown in Fig. 3.7. Similar to the 2-LUT circuits, if the maximum program length is 25 parallel-instructions, the genotype may contain up to 14,800  $((17+5+5+5+5) \times 16 \times 25)$  bits.

GPPLCS can further be extended to evolve 6-LUT circuits. Each SI will consist of 65-bit opcode and six 5-bit operands.

















function names	inputs		addresses				Boolean expressions	2-LUT symbols
	A	B	1	1	0	0		
b0			0	0	0	0	0	
b1			0	0	0	1	$\overline{A + B}$	
b2			0	0	1	0	$\overline{A}B$	
b3			0	0	1	1	$\overline{A}$	
b4			0	1	0	0	$A\overline{B}$	
b5			0	1	0	1	$\overline{B}$	
b6			0	1	1	0	$A \oplus B$	
b7			0	1	1	1	$\overline{A}B$	
b8			1	0	0	0	$AB$	
b9			1	0	0	1	$\overline{A} \oplus B$	
bA			1	0	1	0	$B$	
bB			1	0	1	1	$\overline{A} + B$	
bC			1	1	0	0	$A$	
bD			1	1	0	1	$A + \overline{B}$	
bE			1	1	1	0	$A + B$	
bF			1	1	1	1	1	

Figure 3.6: Functions  $b0 - bF$  used in 2-LUT circuits SI

Table 3.2: Control-codes in 4-LUT circuits SI

fields	number of bits	encoding
function opcode	17	00...0 - 01...1 = $b0000 - bFFFF$ 10...0 - 11...1 = <i>no operation, nop</i>
operand A	5	00000 - 11111 = input[0] - input [31]
operand B	5	00000 - 11111 = input[0] - input [31]
operand C	5	00000 - 11111 = input[0] - input [31]
operand D	5	00000 - 11111 = input[0] - input [31]
Total	37	

Input(4 Registers' value)	Output
0000	0
0001	0
0010	0
0011	0
0100	0
0101	1
0110	1
0111	1
1000	0
1001	1
1010	1
1011	0
1100	1
1101	1
1110	1
1111	1

Figure 3.7: The corresponding content of 4-LUT of the "bF6E0 r31 r27 r08 r29 r00" sub-instruction

### 3.4 The Phenotype of a MLP program

MLP programs are presented in parallel assembly form. Figure 3.8 shows an optimized MLP program for 1-bit full adder in 2-LUT format evolved by GPPLCS. It consists of two sections, the #data and #program sections. The #data section defines constant, input and output Boolean variables. Before starting an execution, an MLP always initializes all variable registers (reg[0]-reg[15]) to logic 0. The constants: line in the #data section initializes constant registers reg[16]-reg[21] to logic 0 and reg[22]-reg[28] to logic 1. The inputs: line defines input variables (Cin, A and B) and assigns them to constant registers (reg[29], reg[30] and reg[31]). The outputs: line defines output variables (Cout and S) and assigns them to variable registers (reg[0] and reg[1]). The #program section contains parallel-instructions that perform Boolean operations.

For example, the numbered lines in the #program section

```

#data
constants: (r16-r21)=0, (r22-r28)=1
inputs:    (r29, r30, r31) <= (Cin, A, B)
outputs:   (r00, r01) => (Cout, S)
#program
00:  b9 r29 r30 r04
01:  b8 r04 r30 r00, b2 r04 r31 r14
02:  b6 r14 r00 r00, b9 r31 r04 r01

```

Figure 3.8: Optimized MLP program for 1-bit full adder in 2-LUT format

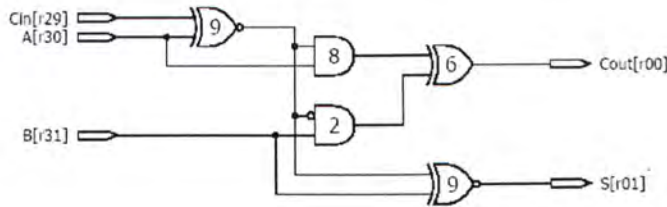


Figure 3.9: A 1-bit full adder in 2-LUT format

in Figure 3.8 list out three parallel-instructions. For easy interpretation, all nop sub-instructions in the original program are hidden. Each sub-instruction consists of three parts: 1) a function name ( $b0$ - $bF$  or nop); 2) registers for input operands; and 3) an output register. For example, the  $b6$   $r14$   $r00$   $r00$  sub-instruction in parallel-instruction 02: performs  $b6$  (XOR) on  $reg[14]$  and  $reg[0]$  and then writes the result back to  $reg[0]$ . Figure 3.9 shows the corresponding combinational logic circuit of the MLP program shown in Figure 3.8.

The situation is similar in evolving circuits in 4-LUT format. Figure 3.10 shows a 2-bit full-adder in 4-LUT format evolved by the GPPLCS. Figure 3.11 shows the 2-bit full adder. Noticeably, three out of the four 4-LUTs can be replaced by 3-LUTs because they have one input set to a constant logic 0.

```

#data
constants: (r16-r21)=0, (r22-r26)=1
inputs:    (r27, r28, r29, r30, r31) <= (Cin, A1, A0, B1, B0)
outputs:   (r00, r01, r02) => (Cout, S1, S0)
#program
00:  bF6E0 r31 r27 r08 r29 r00
01:  b3AA4 r00 r28 r06 r30 r00, bCB9E r00 r28 r30 r21 r01,
     b849E r31 r27 r31 r29 r02

```

Figure 3.10: Optimized MLP program for 2-bit full adder in 4-LUT format

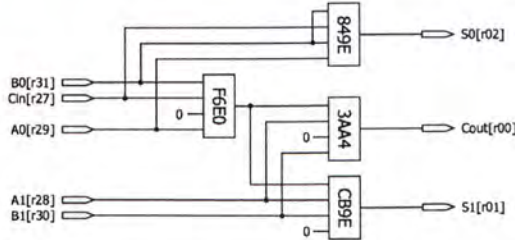


Figure 3.11: A 2-bit full-adder in 4-LUT format

## 3.5 The Evolution Engine

The Evolution Engine (EE) is responsible for manipulating the population, performing genetic operations, loading genetic programs to a MLP for fitness evaluations, calculating/reporting statistics and decompiling the evolved solution program to a symbolic parallel assembly program (MLP program).

### 3.5.1 The Dual-Phase Approach

In order to evolve a solution with GPP, enough spare space (for both parallel-instructions and sub-instructions) are necessary to be given in each genetic program for introns to be built up. Introns are non-effective instructions which do not contribute to the final output of a genetic program. Research results show that the existence of introns in genetic programs in the early and middle stage of a run can benefit evolution [5]. The existence of introns in the early and middle stages of a GP evolution is necessary. Introns are necessary to be in the genetic programs

until we find the first correct program. However, the first correct program is usually not an optimized solution in terms of quality measurements such as LUT count and the propagation delay. To tackle this problem, GPPLCS uses a dual-phase (design and optimization phases) approach with a dual-phase fitness function. The dual-phase fitness function intends to improve the functionality of genetic programs before the first correct genetic program is found. Whenever a correct genetic program is found, it changes its fitness calculation criteria to incorporate optimization-oriented measurements. Besides the dual-phase fitness function, GPPLCS uses different set of genetic operators in the two phases. Details can be found in subsequent section.

In the design phase, GPPLCS aims at finding a 100% functional program (correct program). Its raw fitness is given by

$$f_{dp} = \frac{U}{T}$$

where  $U$  is the number of unmatched training case and  $T$  is the total number of training cases.

The design phase raw fitness  $f_{dp}$  is used to evaluate the functional fitness of a genetic program. If there is a partial correct genetic program, its  $f_{dp}$  is greater than zero.  $f_{dp}$  equals to zero only when all training cases are matched. After finding the first correct genetic program, the evolution will proceed to the optimization phase to optimize correct genetic programs based on some optimization-oriented criteria. In the optimization phase, the raw fitness is given by

$$f_{op} = \frac{g}{g_{max}} + \frac{d}{d_{max}} \times \frac{1}{g_{max}} + \frac{L}{L_{max}} \times \frac{1}{d_{max}g_{max}}$$

The optimization phase raw fitness  $f_{op}$  of a correct genetic program is calculated from three qualitative indicators: 1) the LUT count  $g$  (the number of normal sub-instructions); 2) the propagation delay  $d$ ; and 3) the program length  $L$  (the number of

parallel-instructions). Since a genetic program consists of *nop* and introns,  $L$  represents the number of LUT levels in the logical circuit diagram but not the actual LUT delay in hardware. It is because *nop* and introns are not placed in real hardware so that their LUT delays are not counted. The  $g_{max}$ ,  $d_{max}$  and  $L_{max}$  are the maximum values allowed for the LUT count, the propagation delay and the program length respectively. The main objective of the optimization phase is to reduce the LUT count and then the propagation delay. The last multiplication term in  $f_{op}$  guides the evolution to shorten the lengths of correct genetic programs. Normally, a shorter program has greater chance to have smaller  $g$  and  $d$  values.

By combining the two phases raw fitness functions ( $f_{dp}$  and  $f_{op}$ ), the dual-phase fitness function of the whole evolution process is obtained. In the design phase ( $f_{dp} > 0$ ),  $f_{raw}$  is given by

$$f_{raw} = 1.0 + f_{dp}$$

In the optimization phase ( $f_{dp} = 0$ ),  $f_{raw}$  is given by

$$f_{raw} = f_{op}$$

The constant 1.0 is used to distinguish the two phases. With this fitness function, a partially correct genetic program has an  $f_{raw}$  greater than 1.0 whereas a correct genetic program has an  $f_{raw}$  less than 1.0. In the design phase, whenever GPPLCS finds the first genetic program with an  $f_{raw}$  equal to 1.0, it proceeds to the optimization phase.

### 3.5.2 Genetic operators

In this subsection, genetic operators used in GPPLCS are described.

- Genetic Programs Initialization: GPPLCS uses a binary string (genotype) to encode a MLP program (phenotype).

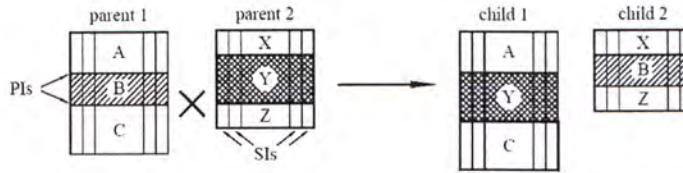


Figure 3.12: PI level crossover on two parents

Before an evolution process, EE initializes all genetic programs in a population randomly. The number of PI ( $L$ : length) of a genetic program is chosen randomly between one to a predefined value ( $L_{max}$ : the maximum program length). Each bit in a genotype has equal chance to be 0 or 1.

- **Tournament Selection:** GPPLCS uses tournament selection to produce its offspring. In each tournament, a fixed number (tournament size) of genetic programs are randomly selected from the population to form a tournament set. According to their fitness, the two best genetic programs in the tournament set are selected as parents to produce two offspring. The tournament size controls the selection pressure and affects the convergence rate.
- **PI level crossover:** It is a two-point crossover to exchange two segments of PI from two parent MLP programs (see Figure 3.12). All sub-instructions in a parallel-instruction will always be kept as a whole. The probability to take this operator is  $P_{cover}$ .
- **Bit Mutation:** It mutates individual bits in the genotype of an MLP program based on a probability  $P_{btmut}$ .
- **SI swapping:** It swaps two sub-instructions inside an MLP program based on a probability  $P_{siswp}$  (see Figure 3.13). It can pack more normal sub-instructions in less number



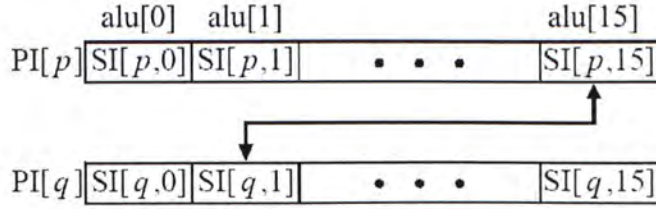


Figure 3.13: An SI swapping in a single MLP program

of parallel-instructions so as to increase the parallelism of MLP program. SI swapping is only used in the optimization phase since it intends to improve the performance of a correct genetic program.

- **SI-Deletion:** It simply replaces a normal sub-instruction with a nop sub-instruction based on a probability  $P_{sidel}$ . It can delete inactive sub-instructions (introns) from a correct genetic program and therefore is only used in the optimization phase.
- **Diversity Maintenance:** In order to maintain the diversity of population, EE adopts an individual replacement technique similar to the pre-selection [47]. In each tournament, two children are bred and evaluated. Then, the better one is selected and compared with its parents. If its fitness is different from both of its parents, it will replace the worst individual in the tournament set. This approach avoids similar individuals filling up the population and hence increases the diversity of search.
- **Dynamic Sample Weighting (DSW):** For some problems, e.g. Boolean functions, the distribution of training samples in the sample space is biased. These biased samples usually cause premature convergence in Genetic Algorithms (GAs) and Genetic Programmings (GPs). DSW [8] is used to balance the contributions of training samples so that the di-

versity of genetic programs can be increased. This operator is only used in the design phase.

### 3.6 Chapter Summary

This chapter has presented GPPLCS. Two core components of GPPLCS (MLP and EE) are described. The MLP is tightly-coupled processor which is used to execute and evaluate genetic programs produced by EE. The genotype of a MLP program is a sequence of control-codes which can be executed on the corresponding MLP directly. The phenotype of a MLP program is a parallel assembly program. EE is an evolutionary process which performs genetic operators, loads genetic programs to the MLP, calculates/reports statistics and decompiles the solution parallel program to a symbolic parallel assembly program.

Furthermore, GPPLCS uses a dual-phase evolutionary approach which divides the evolution into two sequential phases. Firstly, the leaning phase evolves correct genetic programs. Then, the optimization phase improves the qualities of correct genetic programs. A dual-phase fitness function is used to guide the evolution.

---

□ End of chapter.

## Chapter 4

# MLP in hardware

This chapter presents a hardware-assisted Multi-Logic-Unit Processor (MLP). It is a hardware processor built on a Field Programmable Gate Array (FPGA). The purpose is to speed up the evaluation of genetic parallel programs (MLP programs) that represent combinational logic circuits. Six combinational logic circuit problems are presented to show the performance of the hardware-assisted Genetic Parallel Programming based Logic Circuit Synthesizer (GPPLCS). Experimental results show that the hardware MLP speeds up the evolutions over 10 times. For difficult problems such as the 7-bit majority selector, the speedup ratio can be up to 36.

This chapter is organized as follows. Our motivation is described in Section 4.1. Then, the hardware design and implementation of MLP is presented in Section 4.2. It is followed by experiments. Section 4.3 is on the experimental settings. The experimental results and evaluations are given in Section 4.4. Finally, Section 4.5 is a chapter summary.

### 4.1 Motivation

In the last decade, advances in FPGA [2] have made efficient Evolvable Hardware (EHW) [63] possible. EHW uses Evolu-

tionary Algorithms to evolve hardware architecture extrinsically or intrinsically. One of the major usages of EHW is to design combinational logic circuits [19, 36, 52]. However, the importance of scalability of EHW has been recognized by several researchers [27, 30]. It is a tough problem faced not only by EHW researchers, but by other researchers in the fields of evolutionary computation, artificial neural networks, and artificial intelligence in general.

Using hardware to increase the speed of evolution is one of possible ways to combat the high computational cost. FPGA has been adopted to speed up Genetic Algorithms (GAs) and Genetic Programming systems [28, 41, 48, 56]. The basic idea is to put the whole or a part of a GA or GP system in hardware so as to solve problems in a shorter time than a pure software system.

A hardware assisted MLP is designed and implemented to speed up evaluation of genetic parallel programs in GPPLCS. The overall system of hardware assisted GPPLCS is exactly the same as the pure software GPPLCS in Chapter 3. The difference only lies on the MLP. Experiments on six combinational logic circuit problems (i.e. a 6-bit multiplexer, a 2-bit full-adder, a 3-bit comparator, a 6-bit priority selector, a 7-bit majority selector and a 2-digit binary coded decimal to binary decoder) were conducted to show the effectiveness of GPPLCS with the hardware MLP. Experimental results show that the hardware MLP speeds up the evolution by at least 10 times even for the easier problems which are less computation intensive.

## 4.2 Hardware Design and Implementation

This section presents the hardware design and implementation details of MLP. Fig. 4.1 shows the architecture of the core part of MLP. The 16 sub-instruction registers (SIR0-SIR15) store the

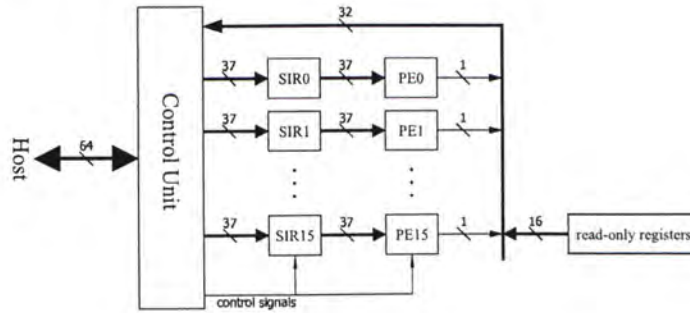


Figure 4.1: The architecture of the MLP core

individual sub-instructions in the current parallel-instructions. The 16 processing elements (PE0-PE15) run sub-instructions and store results to their corresponding variable registers. The Control Unit (CU) decodes parallel-instructions and gives control signals to all MLP components. Due to the limited size of the interface bus between the CU and the host (64-bit only), more than one bus cycle are needed to transfer the evaluation results of all rows in a truth table to the host.

In most cases, GPPLCS only uses the first eight variable registers (R0-R7) to store program outputs. Thus, MLP only needs to transfer the first eight variable registers to the host. In order to maximize the usage of the 64-bit interface bus, MLP is designed to buffer eight sets of program outputs (of eight training cases). In this way, the evaluation results of the entire truth table are passed to the host in burst mode. For example, if there are  $N$  rows in a truth table, it takes  $N/8$  clock cycles to transfer all program outputs to the host.

Fig. 4.2 shows a PE (PE $_i$ ) which receives a sub-instruction from SIR $_i$ . It stores the result in the variable register R $_i$ . The core of the PE is a 4-LUT. It takes two processor clock cycles for the PE to execute one sub-instruction. In the first cycle, four input registers are selected by four multiplexers (M1-M4), and their values are then latched into an Internal Operand Register

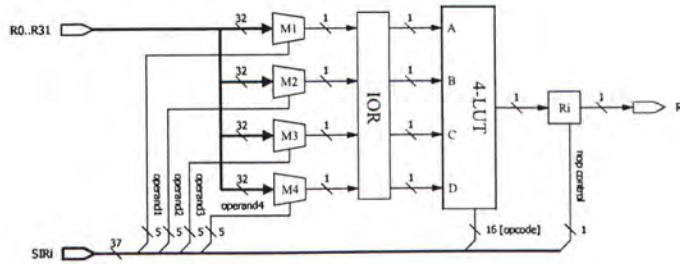


Figure 4.2: A Processing Element

Table 4.1: Pilchard board features

Field	Details
Host interface:	DIMM interface (a 64-bit data bus and a 14-bit address bus)
Operating frequency:	100 MHz
FPGA device:	XCV1000E-HQ240-6
OS supported:	GNU/Linux

(IOR). In the second cycle, the 4-LUT uses the four latched operands to look up one bit and stores the result into  $R_i$ . The IOR is used to pipeline the operations, i.e. selecting operands and looking up results, and to balance the long delay time on the route from the registers' outputs to the multiplexers' inputs.

The MLP is implemented on a Pilchard board [42, 60] which is a high performance reconfigurable computing development environment employing an FPGA. The Pilchard board is plugged into a 133 MHz synchronous dynamic RAM Dual In-line Memory Modules (DIMMs) slot of a PC. The Pilchard board can achieve a very high data transfer rate by making use of the DIMM RAM interface of the PC. Its efficient interface and low cost make it suitable for implementing the MLP. Here are some major features of the Pilchard board:

The FPGA used in the Pilchard board belongs to the Virtex-E series. The MLP uses only 2,515 slices. It is about 20% of the

12,288 slices available in the FPGA. Moreover, only one (out of 96) BlockRAM is used by the MLP. The critical path delay of the MLP is 9.965ns. Hence, it can operate at 100 MHz.

The MLP is coded in Very High Speed Integrated Circuit Hardware Description Language (VHDL) [57] which is a standard language for describing the structure and function of integrated circuits (ICs).

### 4.3 Experimental Settings

To investigate the performance of the GPPLCS, we have used the system to evolve networks for six combinational logic circuit problems in 4-input LUT format (see Table 4.2). Although the GPPLCS evolves circuits in dual-phase approach, all experiments in this chapter are conducted with design phase only. It is because large proportion of execution time used in evolving circuits by the GPPLCS lies on the design phase. Moreover, only one independent run is necessary to show the effectiveness of the hardware assisted GPPLCS.

Note that the 6-bit priority selector is to show the position of value '1' which first appears starting from the least significant bit in the 6-bit input. If none of the bits is set to value '1', an extra output bit which shows the case of all zero value is responsible for this special case. Since we have got six input bits (Input5 - Input0), we need extra three bits to indicate the position. Therefore, there are 4-bit outputs.

The 7-bit majority selector is to determine the majority value of the 7 bits inputs. If more than 4 bits have value '1', the output value will be '1'. Otherwise, the output bit will have value '0'.

In addition, the 2 digit Binary Coded Decimal (BCD) to Binary decoder is to decode the 2 BCD into binary value. BCD is the most common way of encoding decimal digits in computing and in electronic systems. In BCD, a digit is usually represented

Table 4.2: Six combinational logic circuit problems used in GPPLCS with the hardware assisted MLP. The  $N_{in}$  and  $N_{out}$  denote the numbers of inputs and outputs respectively. The  $N_{row}$  ( $=2^{N_{in}}$ ) denotes the number of rows in the truth tables. The  $N_{case}$  ( $=N_{row} \times N_{out}$ ) denotes the total number of training cases.

Name	Description	$N_{in}$	$N_{out}$	$N_{row}$	$N_{case}$
MUX	6-bit multiplexer	6	1	64	64
ADD	2-bit full-adder	5	3	32	96
CMP	3-bit comparator	6	3	64	192
PRI	6-bit priority selector	6	4	64	256
MAJ	7-bit majority selector	7	1	128	128
BCD	2-digit Binary Coded Decimal to Binary de- coder	8	7	256	1792

by four (binary) bits, of which the leftmost (written conventionally) has value 8, and the remaining three have values 4, 2, and 1. Only the combinations of these bits which, when summed, have values in the range 0-9 are valid. The decoder has got 2 BCD. Thus, there will be 8-bit input which is correspond to value 0 - 99. The output value range 0 - 99 then needs 7 bits to represent its output values.

All experimental settings are listed out in Table 4.3 below. In order to have a fair comparison in the performance between hardware-assisted GPPLCS and the pure software counterpart, evolutions of combinational logic circuits for the six combinational logic circuit problems were run on the same host (i.e. the PC where a Pilchard board locates). The host in which the Pilchard board locates is a Pentium III 800 MHz PC with ASUS CUSL2-C motherboard. The Pilchard board relies on the PC to communicate. User can transfer data to the Pilchard board via the DIMM slot in the host PC. The PC host is chosen because



Table 4.3: Experimental settings used in GPPLCS with the hardware assisted MLP

<b>Design phase only</b>	
maximum program length ( $L_{max}$ )	25 parallel instructions (PIs)
initialization	bit random, average 12.5 ( $L_{max}/2$ ) PIs
selection method	tournament (size=10)
4-LUT function set	b0000, ..., bFFFF, nop
inputs	$R_{32-N_{in}} \dots R_{31}$
outputs	outputs: $R_0 \dots R_{N_{out}-1}$
constants	logic 0, logic 1
population size	2000
termination( $t_{max}$ )	40,000,000 tournaments
PI crossover Prob. ( $P_{xover}$ )	0.1
bit mutation Prob. ( $P_{btmut}$ )	0.002
Sub instruction (SI). swap- ping Prob. ( $P_{siswp}$ )	0.0
SI. deletion Prob. ( $P_{sidel}$ )	0.0
Dynamic Sample Weight- ing (DSW) (weights update freq.)	10,000 tournaments
preselection	yes
raw fitness	the ratio of unsolved training cases (= $1.0 + f_{dp}$ )
success predicate	all training cases solved (= 1.0 (i.e. $f_{dp}=0.0$ ))

of the low level control required to manage the Pilchard board.

We tested the problems with both the hardware-assisted GPPLCS and the pure software GPPLCS. The time for each tournament was recorded for comparison.

## 4.4 Experimental Results and Evaluations

Promising results are obtained for all the six combinational logic circuit problems. Table 4.4 summarizes the total elapsed times for the GPPLCS to evolve complete correct solutions with a pure software MLP and a hardware MLP. The  $t_H$  and  $t_S$  columns list out the execution times of the hardware-assisted GPPLCS and the pure software GPPLCS respectively.

It can be seen that the speedup of hardware over software is significant. For the ADD, MUX and PRI problems, the speedups are more than 10 times. For the CMP and BCD problems, the speedups are more than 20 times. For the most difficult problem in our circuits evolved - MAJ, the speedup can be up to 36. The CMP problem takes nearly 10 hours to complete with the pure software GPPLCS, but it only takes less than half an hour with the hardware-assisted GPPLCS. Thus, problems of different levels of difficulties gain different speedups. This is easily recognized because the more difficult the problems, the more tournaments (computational effort) are taken to complete. Fig. 6 shows the speedup curves for the six tested problems. In these figures, the X-axis is the number of tournaments taken while the Y-axis is the speedup ratio ( $t_S/t_H$ ).

Figures 4.3 and 4.4 show that the speedup ratios for the MUX and ADD problems increase steadily to around 10. These two problems are relatively simple. Thus, the required computational efforts to evolve solutions for them are not so large. Conversely, in Figures 4.5, 4.6 and 4.8, the speedup ratios are less than five initially when the evolution takes only a few thousand

Table 4.4: Summary of experimental results in GPLCS with hardware assisted MLP

Problems	$t_H$ (in sec)	$t_S$ (in sec)	speedup ratio ( $t_S/t_H$ )
MUX	68	689	10.13
ADD	346	3497	10.11
CMP	1,575	31,983	20.30
PRI	720	13,471	18.71
MAJ	24,680	895,581	36.29
BCD	11,608	280,269	24.14

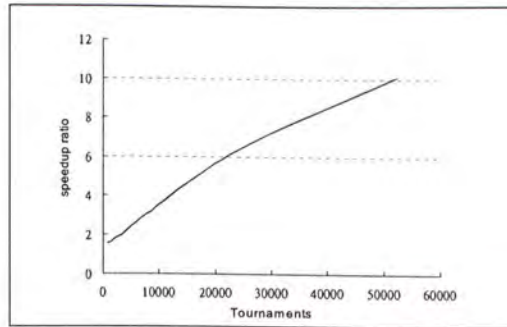


Figure 4.3: The speedup ratio versus tournaments for MUX problem

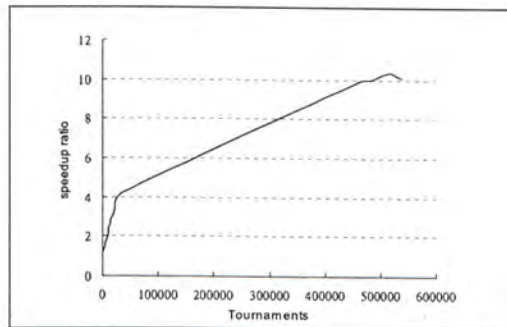


Figure 4.4: The speedup ratio versus tournaments for ADD problem

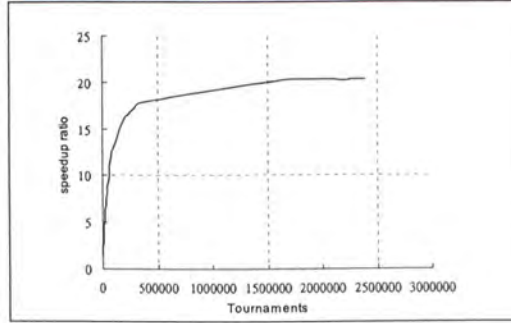


Figure 4.5: The speedup ratio versus tournaments for CMP problem

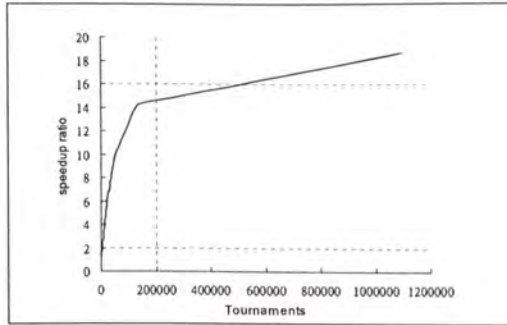


Figure 4.6: The speedup ratio versus tournaments for PRI problem

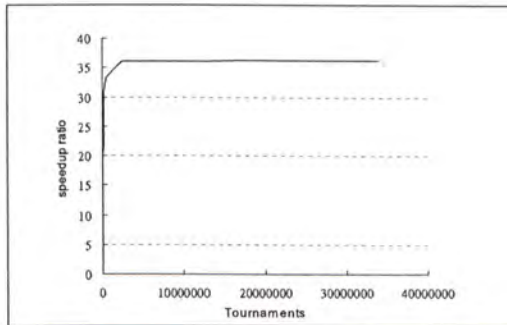


Figure 4.7: The speedup ratio versus tournaments for MAJ problem

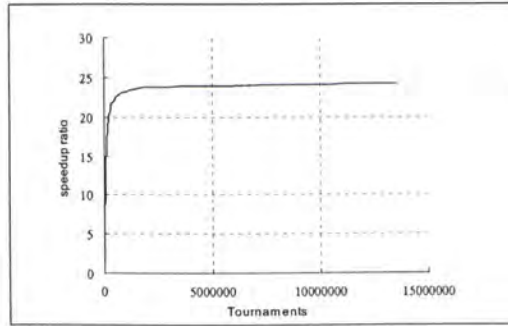


Figure 4.8: The speedup ratio versus tournaments for BCD problem

tournaments. As the evolution completes more tournaments, the speedup ratio increases rapidly to 24 times. For the most difficult problem - MAJ in our problem sets, the speedup can be up to 36 due to large computational efforts required. The result is shown in Figure 4.7.

It is found that the speedup ratio increases with the number of tournaments taken in the evolution. It is obvious since execution time of each hardware evaluation is faster than that of each software evaluation by a certain theoretical limit. However, the speedup is not so high due to the overhead in the communication bus between the software EE and the hardware MLP. Thus, there is a small speedup ratio when the number of tournaments executed is small as the overhead occupies a larger proportion of execution time during the evolution than than fitness evaluation. However, it is expected that the speedup ratio is higher in those problems which have a larger number of tournaments taken as fitness evaluation occupies the largest proportion of execution time. For example, in the MUX problem, only 10-time speedup is obtained due to the small number of tournaments taken (52,286). However, 20-time speedup is found in the CMP problem which takes 2,398,865 tournaments. 36-time speedup is also found in the MAJ problem which takes 34,006,503 tournaments.

## 4.5 Chapter Summary

In this chapter, we have presented the design and implementation of a hardware-assisted GPP Logic Circuit Synthesizer (GPPLCS) prototype which uses a 4-LUT Multi-Logic-Unit Processor (MLP). The MLP uses a generic register machine architecture which can represent any combinational logic circuits. Moreover, the architecture of the MLP is so simple that multiple MLPs can be placed in an FPGA.

The hardware-assisted GPPLCS shows promising results in the speedup. With the help of hardware, GPPLCS achieves a 36-time speedup at most in our tested problems. Furthermore, the speedup ratio increases with the number of tournament taken in solving the problems. It is particularly suitable for solving difficult problems.

---

□ End of chapter.

## Chapter 5

# Feasibility Study of Multi MLPs

Although the circuits evolved by Genetic Parallel Programming based Logic Circuit Synthesizer (GPPLCS) are of good qualities, it is computation intensive. As a result, implementation of GPPLCS in Field Programmable Gate Arrays (FPGAs) is proposed. The idea is to speed up the fitness evaluations. However, the current model is not suitable for the implementation. Two main components in GPPLCS, Evolution Engine (EE) and Multi Logic Unit Processor (MLP), are discovered either one is idle during the evolution. Thus, a Multi MLP Genetic Parallel Programming base Logic Circuit Synthesizer (MMGPPLCS) is proposed and presented for implementation in FPGAs in this chapter. Simulations are done to evaluate the effectiveness of our proposed architecture.

This chapter is organized as follows. Section 5.1 gives our motivation. Then, our proposed architecture of MMGPPLCS is presented in Section 5.2. It is followed by experimental settings in Section 5.3. Section 5.4 is the experimental result and evaluations. Finally, a chapter summary is found in Section 5.5.

## 5.1 Motivation

As introduced in the previous chapter, GPPLCS is a dual phase fitness suitable for evolving LookUp Table (LUT) based circuits. In the design phase, GPPLCS aims at finding a 100 % correct genetic program. Once it is found, GPPLCS proceeds to the optimization phase. Other factors such as lookup table (LUT) count and LUT level count are taken into consideration in the optimization phase. It is discovered that design phase occupies a large proportion of computation time during the whole evolution process. Thus, we would like to seek help from implementation of GPPLCS in FPGAs to speed up the whole evolution process especially in design phase.

In GPPLCS, there are two steps which are always repeated. They are the fitness evaluation and breeding stages. During breeding stage, the current population is used to form a new population by selecting the better programs and using the breeding operators such as crossover and mutation to propagate and modify the programs. It is held in the EE. The programs are then evaluated to measure how fit they are. The two stages are repeated until either a pre-determined number of generations have been processed or an individual meets a pre-determined level of fitness. This is done in the MLP. It is discovered either EE or MLP is idle at any time. Thus, direct implementation of this model in FPGAs does not maximize the benefits of the parallelism in FPGAs.

We propose an MMGPPLCS for implementation in FPGAs which is based on the pipeline concept in hardware design. Implementing algorithmic parallelism, or pipelining, is a frequently used technique in hardware design that reduces the number of clock cycles needed to perform complex operations. The idea is to execute the fitness evaluation (held in the MLP) in parallel with the breeding stages of GPPLCS (done in the EE). In this



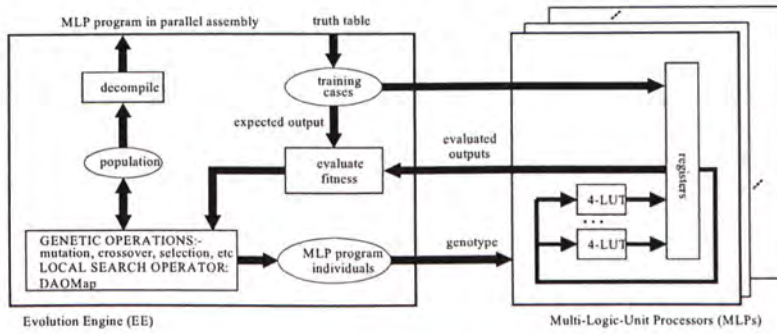


Figure 5.1: The system block diagram of MMGPPLCS

way, both the MLP and the EE can be kept operating at full speed.

## 5.2 Overall Architecture

This section presents the design of the new architecture of GP-PLCS for implementation in FPGAs. Figure 5.1 shows the block diagram of MMGPPLCS. It has one EE and several (up to  $n$ ) MLPs (MLP1, MLP2, ..., MLP $n$ ). The existence of several MLPs is to execute the fitness evaluation in parallel with the breeding operations.

The breeding operations and fitness evaluation are the iterative processes and their execution time are different. The time used in fitness evaluation is much longer than the one used in the breeding operation. Moreover, with the advance in FPGAs, it is possible to allow more MLPs within an FPGA. Thus, we propose to implement one EE and  $n$  MLPs in MMGPPLCS. EE can keep generating new children and then pass to MLPs for evaluation. Previous experiments done on circuits evolved by GPP show that the execution time of breeding operation in EE is 10 times faster than that of fitness evaluation in MLP including overhead. Thus, in our design, we employ 10 MLPs so that every children generated in EE can be evaluated in the

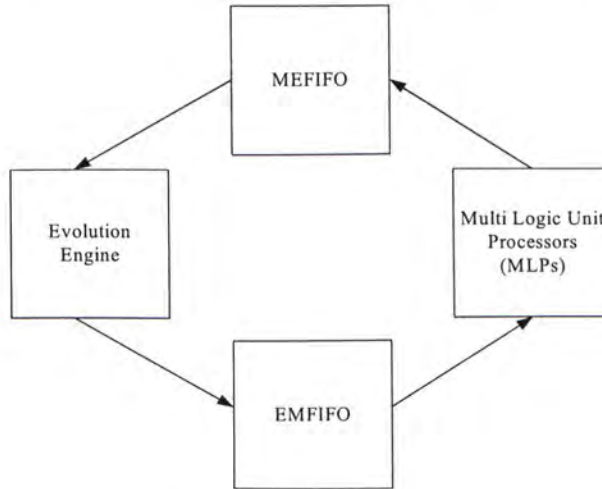


Figure 5.2: FIFO design

MLP with no delay. 1:10 pipeline design can maximize the advantage of implementation in FPGAs. The algorithm can be found in Figure 5.3.

In hardware design of the MMGPPLCS, we insert two FIFOs between EE and MLP, the EMFIFO and the MEFIFO. The purpose is to keep both EE and MLP running. For EE, they can keep evolving children from the population. Children is then placed in EMFIFO for fitness evaluation. Once one of the evaluation engines is ready, it can push one child from the EMFIFO for evaluation and place the fitness evaluation result in the MEFIFO. The process continues until a solution is found. Figure 5.2 shows our proposed design. In MMGPPLCS, evolution are no longer based on up-to-date fitness evaluation of the population. Instead, cross breeding is among old and new generations as evaluation is done on different era child. Since it is different from the original flow of GPPLCS, a software simulation is necessary to evaluate the impact on GPPLCS. The simulation result is presented later.

### 5.3 Experimental settings

Simulation of MMGPPLCS was done on six problems. We first assume that ratio of execution time of the EE and that of MLP is 1 over 10. That means there are 10 MLPs and one EE in the MMGPPLCS. In our software simulation, there are 3 phases. First of all, it is the initialization. The first ten breeding operations without any fitness evaluations are done initially. This is to model the situation in the MMGPPLCS. Then, it comes to pipeline phase. A fitness evaluation is done on the first children generated. After the first fitness evaluation is done, the children evaluated are determined whether it is discarded or not. If they are fitter than their parents, they replace their parents. As the breeding operation and fitness evaluation are expected to execute in parallel in this phase, the first fitness evaluation is followed by the eleventh breeding operations in our simulation. Indeed, we resumes original flow in the pipeline phase. That means a breeding operation is followed by a fitness evaluation. However, the fitness evaluation is not on the children which are just generated. Instead, the MLP evaluates the past children. The pipeline phase continues until a number of tournaments have been processed or an individual meets a pre-determined level of fitness (i.e.  $f_{dp} = 0$ ). The evolution is finished in the last phase. See Figure 5.3.

The six problems are 2-bit full adder (ADD2), 6-bit comparator (CMP3), 4-to-1 multiplexer (MUX6), 6-bit priority selector (PSL6), 3-bit multiplier (MUL3) and 6-bit one's counter (OCN6). See Table 5.1.

Note that the 6-bit priority selector is to show the position of value - 1 which first appears starting from the least significant bit in the 6-bit input. If none of the bits is set to value - 1, an extra output bit which shows the case of all zero value. Since we have got six input bits (Input5 - Input0), we need extra three

Table 5.1: Six combinational logic circuit problems used in the simulation. The  $N_{in}$  and  $N_{out}$  denote the numbers of inputs and outputs respectively. The  $N_{row}$  ( $=2^{N_{in}}$ ) denotes the number of rows in the truth tables. The  $N_{case}$  ( $=N_{row} \times N_{out}$ ) denotes the total number of training cases.

Name	Description	$N_{in}$	$N_{out}$	$N_{row}$	$N_{case}$
ADD2	2-bit full-adder	5	3	32	96
CMP3	3-bit comparator	6	3	64	192
MUX6	6-bit multiplexer	6	1	64	64
PSL6	6-bit priority selector	6	4	64	256
MUL3	3-bit multiplier	6	6	64	384
OCN6	6-bit one's counter	6	3	64	192

bits to indicate the position. Therefore, there are 4-bit outputs.

In addition, the 6-bit one's counter is to calculate the number of value - 1 in the 6-bit inputs. Therefore, it requires 3-bit to represent the number in the output.

All experimental settings are listed in Table 5.2 below. Having investigated the difficulties of the six benchmark problems shown in Table 5.1, we set the maximum program length to 25 PIs. This provides enough sub-instructions (for both effective operations and introns) to evolve correct programs. Hence, at most 400 (25 by 16) operations can be used to build a solution. As introduced before, the design phase occupies the largest proportion of execution time during evolution. Thus, experiments conducted in the design phase only are sufficient to show the effectiveness of the MMGPPLCS.

We have also tried the six problems on the GPPLCS. The GPPLCS adopts the same experimental settings as MMGPPLCS which are shown in Table 5.2. To ensure a fair comparison between MMGPPLCS and GPPLCS, all evolutions of combinational logic circuits for the six combinational logic circuit problems were run on the same PC configuration (Pentium 4 CPU 2.80GHz with 512 MB RAM) with 20 independent runs.

**Algorithm** *MMGPPLCS in simulation*

**Input:** Truth table of circuits

**Output:** Circuits in 4-LUT format

1. Initialize population
2. Evaluate population
3. Perform 10 breeding operations:
4. Tournament selection, Bit Mutation with  $P_{btmut}$  and PI crossover with  $P_{xover}$
5. Evaluate the first children
6. **if**  $f_{children} > f_{parents} \wedge children \neq parents$
7.     **then**
8.         Replace parents with children
9.     **else**
10.         Discard children
11. Perform breeding operations:
12. Tournament selection, Bit Mutation with  $P_{btmut}$  and PI crossover with  $P_{xover}$
13. Evaluate children
14. **if**  $f_{children} > f_{parents} \wedge children \neq parents$
15.     **then**
16.         Replace parents with children
17.     **else**
18.         Discard children
19. **if**  $t < t_{max}$
20.     **then**
21.         **if**  $f_{dp} > 0$
22.             **then**
23.                 GOTO Step 11
24.             **else**
25.                 Terminate
26.     **else**
27.         Terminate
- 28.

Figure 5.3: Algorithm of MMGPPLCS in simulation

Table 5.2: Experimental settings used in MMGPPLCS and GPPLCS

**Design phase only**


---

maximum program length ( $L_{max}$ )	25 parallel instructions (PIs)
initialization	bit random, average 12.5 ( $L_{max}/2$ ) PIs
selection method	tournament (size=10)
4-LUT function set	b0000, ..., bFFFF, nop
inputs	$R_{32-N_{in}} \dots R_{31}$
outputs	outputs: $R_0 \dots R_{N_{out}-1}$
constants	logic 0, logic 1
population size	2000
termination( $t_{max}$ )	40,000,000 tournaments
PI crossover Prob. ( $P_{xover}$ )	0.1
bit mutation Prob. ( $P_{btmut}$ )	0.002
Sub instruction (SI). swap- ping Prob. ( $P_{sisup}$ )	0.0
SI. deletion Prob. ( $P_{sidel}$ )	0.0
Dynamic Sample Weight- ing (DSW) (weights update freq.)	10,000 tournaments
preselection	yes
raw fitness	the ratio of unsolved training cases (= $1.0 + f_{dp}$ )
success predicate	all training cases solved (= 1.0 (i.e. $f_{dp}=0.0$ ))

---

Table 5.3: Number of tournaments ( $\times 10^6$ ) needed by MMGPPLCS and GPPLCS in design phase on six problems (Average value)

Version	ADD2	CMP3	MUX6	PSL6	MUL3	OCN6
MMGPPLCS	0.56	1.79	0.09	0.39	83.32	9.97
GPPLCS	0.50	1.80	0.08	0.47	81.94	9.84

## 5.4 Experimental results and evaluations

The proposed MMGPPLCS neither improves nor worsens the evolution process. Table 4.4 shows the average number of tournaments required in evolving six circuits in both the MMGPPLCS and the GPPLCS. The number of tournaments are expressed in  $10^6$  order of magnitude.

Our objective of the simulation is to prove the pipeline phase works. Although the MMGPPLCS does not decrease the number of tournaments used in the whole evolution process, the MMGPPLCS is a feasible model for implementation in FPGAs. This multi MLPs with one EE can keep both MLP and EE running without being idle. The performance of the MMGPPLCS is similar to that of the GPPLCS. This is critical to the success of the MMGPPLCS. Executing parallel fitness evaluation with breeding operators without increasing number of tournaments can be found during the whole evolution process. As a result, the MMGPPLCS is a suitable for implementation in FPGAs.

## 5.5 Chapter Summary

The proposed MMGPPLCS has been shown to be a feasible model for implementation in FPGAs. Simulation results show that MMGPPLCS does not increase the number of tournaments during evolution of circuits. Since the performance of the MMGPPLCS is nearly the same as that of GPPLCS, the GPPLCS can be benefited from a hardware implementation by adopting a

model like MMGPPLCS to significantly increase the evaluation speed by orders of magnitude as shown in next chapter.

---

□ End of chapter.



## Chapter 6

# A Hybridized GPPLCS

Based on Genetic Parallel Programming (GPP) [43] paradigm and a deterministic local search operator - FlowMap [21], a logic circuit synthesizing system integrating Genetic Parallel Programming based Logic Circuit Synthesizer (GPPLCS) and FlowMap, a Hybridized GPPLCS is developed. To show the effectiveness of the proposed HGPPLCS, six combinational logic circuit problems are used for evaluations. Each problem is run for 50 times. Experimental results show that both the lookup table counts and the propagation delays of the circuits collected are better than those obtained by conventional design or evolved by GPPLCS alone. For example, in a 6-bit one counter experiment, we obtained combinational digital circuits with 8 four-input lookup tables in 2 LUT level on average. It utilizes 2 lookup tables and 3 LUT levels less than circuits evolved by GPPLCS alone.

This chapter is organized as follows. Our motivation can be found in Section 6.1. Section 6.2 presents HGPPLCS. Experimental settings can be found in Section 6.3. Section 6.4 presents results and discussions. Finally, section 6.5 concludes our work.

## 6.1 Motivation

Although the qualities of evolved combinational digital circuits from GPPLCS are better than conventional designs, there is still room for improvement. Algorithms hybridize a non-genetic local search to refine the qualities of solutions with a genetic algorithm are called memetic algorithms [53]. This inspires an idea of using a local search operator in GPPLCS. Since GPP is population-based, it has a number of individuals (circuits) that performs the same function (i.e. many-to-one genotype-phenotype mapping). Thus, GPP can provide a number of different circuits as inputs to the FlowMap algorithm. In this way, FlowMap can return different mapping solutions so that a better solution can be obtained. Since FlowMap obtains a depth optimal mapping solutions when it is applied on 2-input lookup table (LUT) Boolean circuit, GPPLCS must first evolve circuits in 2-input LUT (2-LUT) and then relies on FlowMap to give a 4-LUT mapping solution. This new GPPLCS with a local search operator - FlowMap is the basic of our HGPPLCS.

## 6.2 Overall system architecture

FlowMap [21] is an LUT-based FPGA mapping algorithm for depth minimization guaranteeing depth-optimal mapping for a given input Boolean circuit. Since the working principle of FlowMap algorithm is not our focus, only a very brief description of FlowMap is given in this section. Details can be found in 2.1.1. A key step in FlowMap algorithm is to compute a minimum height  $K$ -feasible cut in a network, which is solved optimally in polynomial time based on network flow computation. FlowMap algorithm also effectively minimizes the number of LUTs by maximizing the volume of each cut and by several post-processing operations. It should be noted that FlowMap

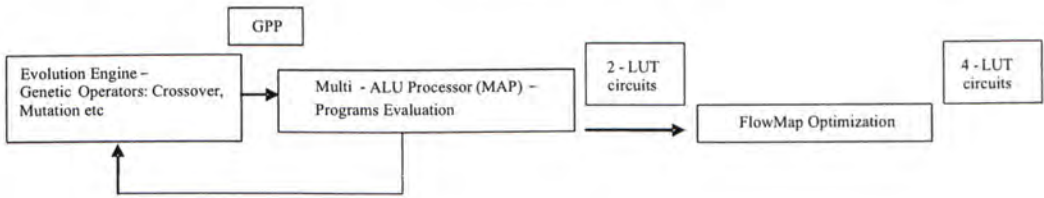


Figure 6.1: HGPPLCS

gets a better mapping solution when a 2-LUT Boolean circuit is given as an input. As a result, it gives an opportunity of adopting FlowMap in GPPLCS.

Since FlowMap will return a depth optimal mapping solution for a 2-LUT Boolean circuit input, and hence is a very suitable tool to help GPPLCS to locate the local optimum. Since GPPLCS can provide a population of 2-LUT Boolean circuits with same functionality, FlowMap can give a best mapping solution among all the mapping solutions.

HGPPLCS first evolves 2-LUT Boolean circuits. Then it chooses the best one among the population of the 2-LUT Boolean circuits as the input for the FlowMap. The FlowMap generate a 4-LUT mapping solution (see Figure 6.1). The synergy effect of GPPLCS and FlowMap in HGPPLCS is well established that evolutionary algorithms are not well suited to fine tuning greedy local search in complex combinatorial spaces and that hybridization with other techniques can greatly improve the efficiency of search [22, 23, 26, 61]. FlowMap can be applied to significantly improve GPPLCS by obtaining the local optimal circuits efficiently and effectively (see Figure 6.2). The population-based GPPLCS provides FlowMap with a group of diversified Boolean circuits with the same functionality which cannot be obtained by any deterministic algorithms. In this way, a global optimal circuit can be evolved with the aid of the efficient local and global search power efficiency from FlowMap and GPP respectively.

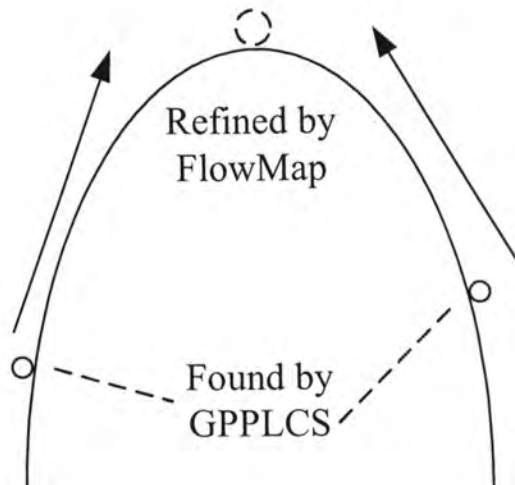


Figure 6.2: FlowMap refines the fitness of individuals in GPPLCS

### 6.3 Experimental settings

HGPPLCS were evaluated on six problems the same used in Chapter 5. They are the 2-bit full adder (ADD2), 6-bit comparator (CMP3), 4-to-1 multiplexer (MUX6), 6-bit priority selector (PSL6), 3-bit multiplier (MUL3) and 6-bit one's counter (OCN6) (see Table 6.1). They are all benchmark Boolean problems that have been tried in other evolvable hardware approaches.

All experimental settings are listed in Table 6.2 below. Having investigated the difficulties of the six benchmark problems shown in Table 6.1, we set the maximum program length to 25 PIs. This provides enough sub-instructions (for both effective operations and introns) to evolve correct programs. Hence, at most 400 (25 X 16) operations can be used to build a solution. It is important to note that, in the optimization stage, we force the system to optimize the size of the correct programs as much as possible. Thus, all runs terminate after 40,000,000 tournaments which we believe it is large enough to evolve the circuits. Preliminary experiments have been done to show circuits can

Table 6.1: Six combinational logic circuit problems used in HGPPLCS. The  $N_{in}$  and  $N_{out}$  denote the numbers of inputs and outputs respectively. The  $N_{row}$  ( $=2^{N_{in}}$ ) denotes the number of rows in the truth tables. The  $N_{case}$  ( $=N_{row} \times N_{out}$ ) denotes the total number of training cases.

Name	Description	$N_{in}$	$N_{out}$	$N_{row}$	$N_{case}$
ADD2	2-bit full-adder	5	3	32	96
CMP3	3-bit comparator	6	3	64	192
MUX6	6-bit multiplexer	6	1	64	64
PSL6	6-bit priority selector	6	4	64	256
MUL3	3-bit multiplier	6	6	64	384
OCN6	6-bit one's counter	6	3	64	192

be evolved at most 40,000,000 tournaments in our benchmark problems.

In order to show the effectiveness of HGPPLCS, we tried the same six problems on GPPLCS and FlowMap. However, we have not compared with any evolvable hardware techniques like Cartesian GP due to the different circuits evolved. They are in boolean gate form (i.e., 2-LUT) while we are focusing on 4-LUT circuits. GPPLCS adopts the same experimental settings as HGPPLCS which are shown in Table 6.2. To ensure a fair comparison between HGPPLCS and GPPLCS, all evolutions of combinational logic circuits for the six combinational logic circuit problems are run on the same PC configuration (Pentium 4 CPU 2.80GHz with 512 MB RAM) with 50 independent runs. In addition, circuits are also evolved by dual phase approach. The only difference is in the types of circuits evolved. GPPLCS evolves the circuits with 4-LUT while HGPPLCS evolves the 2-LUT type. Since the difficulty for evolving 2-LUT and 4-LUT Boolean circuits in each problem are different, numbers of tournaments are not compared in this paper.

Results from the FlowMap algorithm are collected from the experiments which were run on UCLA RASP FPGA/CPLD

Technology Mapping and Synthesis Package [1]. Firstly, we used the ESPRESSO [7] to optimize the truth tables of the six Boolean problems into optimal (or near optimal) sum of product (SOP) forms. Then the resulting SOP expressions were passed to produce 4-LUT networks with FlowMap algorithm.

## 6.4 Experimental results and evaluations

From the 50 runs of the six individual problems, it is shown that HGPPLCS evolved the best circuits among the three methods (HGPPLCS, GPPLCS and FlowMap). Table 6.3 shows the best circuits collected from the three methods and Table 6.4 indicates the successful rate of evolving circuits in HGPPLCS and GPPLCS. Since FlowMap depends heavily on the given input circuits, the mapping solution will not be of a good quality if the input circuits provided are in a bad form (e.g in SOP forms). As FlowMap is a deterministic algorithm, the mapping solutions are always the same regardless of the number of times it is tried. Thus, mapping results by FlowMap are not shown in the charts about comparison between HGPPLCS and GPPLCS. Fig. 6.3 is the average values of the circuits evolved (in terms of 4-LUT count and LUT level) collected in the 50 independent run of HGPPLCS and GPPLCS while Fig. 6.4 is the best circuit evolved in the 50 runs of HGPPLCS and GPPLCS. Obviously, HGPPLCS successfully improves the GPPLCS. On the six problems, both the average number of LUT count and LUT level in the circuits evolved from HGPPLCS are smaller than that from GPPLCS. HGPPLCS outperforms GPPLCS. The circuits evolved by the HGPPLCS are better than that by the GPPLCS. In the 3-bit comparator problem (CMP3), the best circuit evolved from HGPPLCS is 1 4-LUT and 1 LUT level less than the one from GPPLCS. The circuit is shown in Fig. 6.5.

Table 6.2: Experimental settings used in HGPPLCS

	both design and optimization phases	
maximum program length ( $L_{max}$ )	25 parallel instructions (PIs)	
initialization	bit random, average 12.5 ( $L_{max}/2$ ) PIs	
selection method	tournament (size=10)	
4-LUT function set	b0000, ..., bFFFF, nop	
2-LUT function set	b0, ..., bF, nop	
inputs	$R_{32-N_{in}} \dots R_{31}$	
outputs	outputs: $R_0 \dots R_{N_{out}-1}$	
constants	logic 0, logic 1	
population size	2000	
termination( $t_{max}$ )	40,000,000 tournaments	
PI crossover Prob. ( $P_{xover}$ )	0.1	
	design phase	optimization phase
bit mutation Prob. ( $P_{btmut}$ )	0.002	0.0
Sub instruction (SI). swapping Prob. ( $P_{siswap}$ )	0.0	0.5
SI. deletion Prob. ( $P_{sidel}$ )	0.0	0.1
Dynamic Sample Weighting (DSW) (weights update freq.)	10,000 tournaments	-
preselection	yes	-
raw fitness	the ratio of unsolved training cases (= 1.0 + $f_{dp}$ )	the ratio of LUT level & LUT count (= $f_{op}$ )
success predicate	all training cases solved (= 1.0 + $f_{dp}=0.0$ )	optimize as much as possible (i.e. $f_{op} \leq 0$ )

Table 6.3: Best circuits collected from HGPPLCS, GPPLCS and FlowMap algorithm on six problems

Version	Type	ADD2	CMP3	MUX6	PSL6	MUL3	OCN6
HGPPLCS	LUT	4	5	2	5	15	7
	Level	2	2	2	2	3	2
GPPLCS	LUT	4	6	2	5	15	6
	Level	2	3	2	3	4	3
FlowMap	LUT	16	23	3	11	50	113
	Level	3	3	2	3	3	4

Table 6.4: Successful rate of evolving circuit problems in HGPPLCS and GPPLCS

Version	ADD2	CMP3	MUX6	PSL6	MUL3	OCN6
HGPPLCS	100%	100%	100%	100%	50%	58%
GPPLCS	100%	100%	100%	100%	54%	100%

It is found that the circuits evolved from HGPPLCS may have a greater number of 4-LUT count than the ones from GPPLCS. In the 6-bit one's counter problem (OCN6), although the best circuit evolved from HGPPLCS is 1 LUT level less than the one from GPPLCS, it utilizes 1 4-LUT more. The reason lies on the FlowMap algorithm. Since FlowMap only guarantees a depth optimal mapping solution on a given input circuit, the number of 4-LUT of the solution may not be smaller than the circuit found in GPPLCS. However, the depth of the circuit is always the smallest.

HGPPLCS shows a perfect synergy between GPPLCS and FlowMap. The population based GPPLCS provides FlowMap with a group of diversified Boolean circuit with the same functionality while FlowMap returns a better mapping solutions than GPPLCS.

The successful rate of the HGPPLCS and the GPPLCS are nearly the same. From the rate shown in Table 6.4, it is found that the MUL3 and OCN6 problems are more difficult than



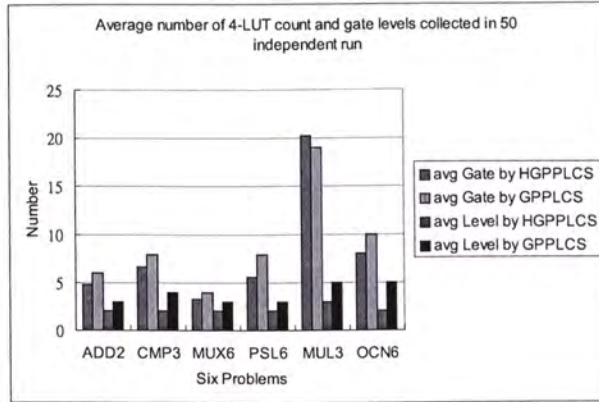


Figure 6.3: Average number of 4-LUT count and LUT level collected from HGPPLCS and GPPLCS on the six problems in 50 runs

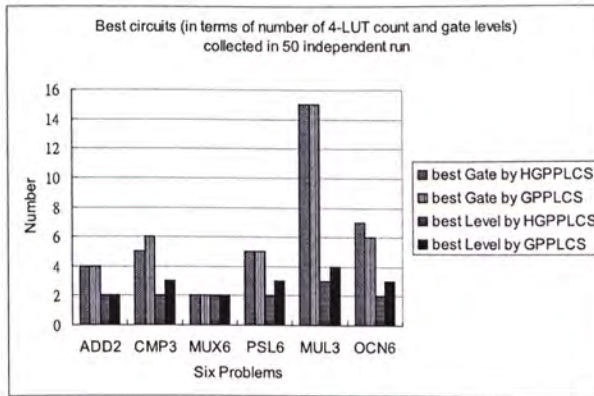


Figure 6.4: Best number of 4-LUT and LUT level collected from HGPPLCS and GPPLCS on the six problems in 50 runs

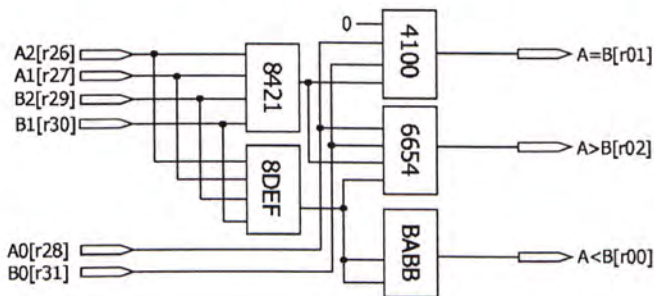


Figure 6.5: The best 3-bit comparator evolved by the HGPPLCS

others. As HGPPLCS first evolves circuit in 2-LUT form and then relies on FlowMap to give a 4-LUT mapping, the searching space in HGPPLCS is much larger than those in GPPLCS which evolves 4-LUT instead. Thus, it is expected that the successful rate of HGPPLCS is lower or equal to that of GPPLCS.

## 6.5 Chapter Summary

In this chapter, we have presented a Hybridized Genetic Parallel Programming based Logic Circuit Synthesizer (HGPPLCS). It makes use of a Genetic Parallel Programming based Logic Circuit Synthesizer (GPPLCS) and the FlowMap algorithm. HGPPLCS applies a dual phase approach to evolve a 2-LUT circuit. Then the circuit is passed to FlowMap for further optimization. Finally, FlowMap returns a depth optimal mapping solution based on the given input circuit. Experimental results show that HGPPLCS improves the performance of GPPLCS in terms of qualities of circuits. The qualities of evolved circuits are the best among the three methods (HGPPLCS, GPPLCS and FlowMap).

---

□ End of chapter.

## Chapter 7

### A Memetic GPPLCS

By including a deterministic local search operator - DAOMap [13] in Genetic Parallel Programming (GPP), a Memetic GPP based Logic Circuit Synthesizer (MGPPLCS) is developed. To show the effectiveness of the proposed MGPPLCS, six combinational logic circuit problems are used for evaluations. Each problem is run for 20 times. Experimental results show that MGPPLCS is both more efficient and effective than GPP. On average, MGPPLCS requires 1 order of magnitude fewer evaluations to identify higher quality solutions. Both the lookup table counts and the propagation delays of the circuits collected are better than those obtained by conventional design or evolved by GPP alone. For example, in a 6-bit priority selector experiment, we evolved combinational digital circuits with 5.1 four-input lookup tables in 2 LUT level on average. It utilizes 2 lookup tables and 1 LUT levels less than circuits evolved by GPPLCS alone.

This chapter is organized as follows. Section 7.1 gives our motivation. MGPPLCS is presented in Section 7.2. The experimental settings can be found in Section 7.3. It is followed by experimental results and evaluations in Section 7.4. Finally, Section 7.5 is a chapter summary.

## 7.1 Motivation

Evolutionary Algorithms (EAs) are a class of search and optimization techniques that work on a principle inspired by nature: Darwinian Evolution. It is well established that hybridization with other techniques in EAs can greatly improve the efficiency of search. Algorithms hybridize a non-genetic local search to refine the qualities of solutions with a genetic algorithm are called memetic algorithms [53]. This inspires the idea of using a deterministic local search operator in GPPLCS.

DAOMap algorithm [13] proposed by Prof. Jason Cong is a technology mapping algorithm for depth minimization in lookup table (LUT)-based FPGA designs, which is optimum for any K-bounded Boolean network. DAOMap can return a depth optimal mapping solution with possible area optimization based on a given Boolean circuit. Thus, DAOMap is an ideal local search operator for GPP so that it can improve GPP in both efficiency and effectiveness. Any individuals found in GPP can be refined by DAOMap. A large number of evaluations can be saved to locate optima. Moreover, DAOMap can force GPP to explore more optima by recording the previous optima found. This new GPPLCS with a local search operator - DAOMap becomes the MGPPLCS.

## 7.2 Overall system architecture

Based on GPPLCS, a combinational logic circuit design system, MGPPLCS is developed. Basically, the architecture of GPPLCS and MGPPLCS are the same. The difference is the application of local search operator - DAOMap in MGPPLCS. The core of the MGPPLCS system consists of an Evolution Engine (EE) and MLP (see Fig. 7.1). EE manipulates the genetic parallel programs and performs genetic and local search operations. MLP

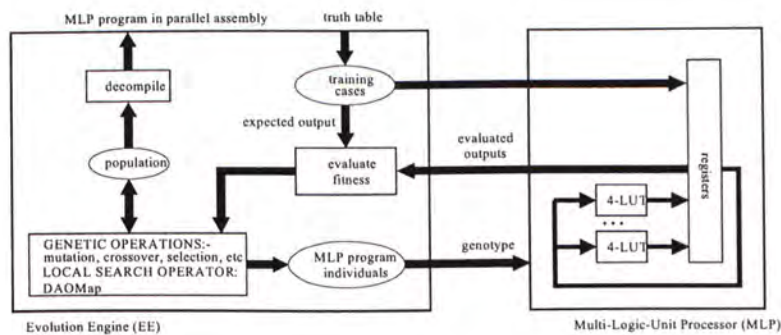


Figure 7.1: The system block diagram of MGPPLCS

is responsible for the genetic parallel programs evaluation.

Similar to GPPLCS, all combinational digital circuits are evolved by a dual phase (i.e. design and optimization phases) approach. Different sets of genetic operators including crossover, bit mutation and sub-instruction swapping are used in different stages. In the design phase, the MGPPLCS system aims at finding a 100% functional program (correct program). The raw fitness is given by the ratio of unsolved training cases. In the optimization phase, the raw fitness then put emphasis on the LUT count, the propagation delay and the program length. In other words, the major objective of the optimization stage is reducing the LUT count and then the propagation delay. Obviously, we apply our local search operator in this phase. DAOMap can be applied to significantly improve MGPPLCS by obtaining the local optimal circuits efficiently and effectively (see Fig.7.2). The population-based MGPPLCS provides DAOMap with a group of diversified Boolean circuits with same functionality which cannot be obtained by any deterministic algorithms while DAOMap returns the refined individuals (optima). In this way, a global optimal circuit can be evolved with efficiency and global search power from DAOMap and EA respectively.

However, it should be noted that refined individuals are not put back to the population. Since any introns will be removed

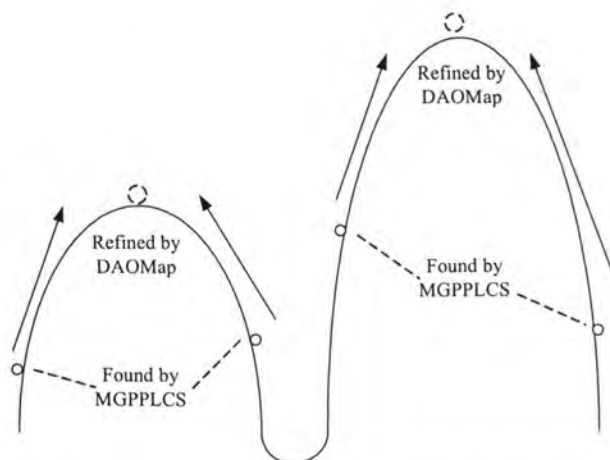


Figure 7.2: DAOMap refines the fitness of individuals in GPPLCS

after the refinement, refined individuals would not benefit evolution. To some extent, refined individuals in a population would dominate and let GPP trap in a local optima. As a result, refined individuals are not placed in the population.

Instead, refined individuals serve as a similarity measure. Refined individuals (optima) are recorded in terms of number of LUTs and LUT's level. Any newly evolved individuals will retain only when it is different from the previous recorded optima in there two values. In each tournament, the MGPPPLCS generates two new genetic programs (children). They are refined by DAOMap. If a child is structurally equivalent to any of optimum found before (number of LUTs and LUT's level are found in the list), it will be discarded. This is to maintain a reasonable diversity in the search. In this way, it serves as a diversity measure and MGPPPLCS can then be forced to evolve new optima. See Figure 7.3.

**Algorithm** *MGPPLCS***Input:** Truth table of circuits**Output:** Circuits in 4-LUT format

1. Initialize population
2. Evaluate population
3. **if**  $f_{dp} > 0$  /\* design phase \*/
4.   **then**
5.       Perform breeding operations:
6.       Tournament selection, Bit Mutation with  $P_{btmut}$  and PI crossover with  $P_{xover}$
7.   **else** /\* optimization phase \*/
8.       Perform breeding operations:
9.       Tournament selection, SI swapping with  $P_{siswap}$ , SI deletion with  $P_{sidel}$  and PI crossover with  $P_{xover}$
10.       Optimize circuits with DAOMap
11. Evaluate children
12. **if**  $f_{children} > f_{parents} \wedge children \neq parents$
13.   **then**
14.       Replace parents with children
15.   **else**
16.       Discard children
17. **if**  $t < t_{max}$
18.   **then**
19.       **if** Design phase  $\wedge f_{dp} > 0$
20.         **then**
21.           GOTO Step 3
22.         **else**
23.           **if** Optimization phase
24.             **then**
25.               GOTO Step 3
26.             **else**
27.               Terminate
28.   **else**
29.       Terminate
- 30.

Figure 7.3: Algorithm of MGPPLCS

Table 7.1: Six combinational logic circuit problems used in MGPPLCS. The  $N_{in}$  and  $N_{out}$  denote the numbers of inputs and outputs respectively. The  $N_{row}$  ( $=2^{N_{in}}$ ) denotes the number of rows in the truth tables. The  $N_{case}$  ( $=N_{row} \times N_{out}$ ) denotes the total number of training cases.

Name	Description	$N_{in}$	$N_{out}$	$N_{row}$	$N_{case}$
ADD2	2-bit full-adder	5	3	32	96
CMP3	3-bit comparator	6	3	64	192
MUX6	6-bit multiplexer	6	1	64	64
PSL6	6-bit priority selector	6	4	64	256
MUL3	3-bit multiplier	6	6	64	384
OCN6	6-bit one's counter	6	3	64	192

### 7.3 Experimental settings

MGPPLCS was evaluated on the same six problems as in Chapters 5 and 6. They are 2-bit full adder (ADD2), 6-bit comparator (CMP3), 4-to-1 multiplexer (MUX6), 6-bit priority selector (PSL6), 3-bit multiplier (MUL3) and 6-bit one's counter (OCN6). (see Table 7.1).

All experimental settings are listed in Table 7.2 below. Having investigated the difficulties of the six benchmark problems shown in Table 7.1, we set the maximum program length to 25 PIs. This provides enough sub-instructions (for both effective operations and introns) to evolve correct programs. Hence, at most 400 (25 X 16) operations can be used to build a solution. Noticeably, in the optimization stage, we force the system to optimize the size of the correct programs as much as possible. Thus, all runs terminate after 40,000,000 tournaments.

In order to show the effectiveness of MGPPLCS, we tried the six problems on GPPLCS, DAOMap and FlowMap. The GPPLCS adopt the same experimental settings as MGPPLCS which is shown in Table 7.2 except all runs terminate after 40,000,000 tournaments. Moreover, no local search operator will be used in GPPLCS. To ensure a fair comparison between



MGPPLCS and GPPLCS, all evolutions of combinational logic circuits for the six combinational logic circuit problems are run on the same PC configuration (Pentium 4 CPU 2.80GHz with 512 MB RAM) with 20 independent runs.

Results from DAOMap and FlowMap algorithm are collected from the experiments which were run on UCLA RASP FPGA/CPLD Technology Mapping and Synthesis Package [1]. Firstly, we used the ESPRESSO [7] to optimize the truth tables of the six Boolean problems into optimal (or near optimal) sum of product (SOP) forms. Then the resulting SOP expressions were passed to produce 4-input LUT networks with the DAOMap algorithm as well as FlowMap algorithm.

## 7.4 Experimental results and evaluations

From the 20 runs of the six individual problems, it is shown that MGPPLCS evolved the best circuits among the four methods (MGPPLCS, GPPLCS, DAOMap and FlowMap). Table 7.3 shows the best circuits collected from the four methods while Table 7.4 shows the average value. Please note that all runs are successful. That means we can evolve solutions in every run. Since DAOMap and FlowMap are deterministic algorithms, the mapping solutions are always the same regardless of the number of times it is tried. Thus, the result will be the same in both tables.

It is shown that MGPPLCS and GPPLCS outperform DAOMap and FlowMap since they depend heavily on the given input circuits. The mapping solution will not be of a good quality if the input circuits provided are in a bad form (e.g. in SOP forms). Obviously, the MGPPLCS successfully improves the GPPLCS. On the six problems, both the average number of LUT count and LUT's level in the circuits evolved from MGPPLCS are smaller than that from GPPLCS. Moreover, the number of tournaments

Table 7.2: Experimental settings used in MGPPLCS

both design and optimization phases		
maximum program length ( $L_{max}$ )	25 parallel instructions (PIs)	
initialization	bit random, average 12.5 ( $L_{max}/2$ ) PIs	
selection method	tournament (size=10)	
4-input LUT function set	b0000, ..., bFFFF, nop	
inputs	$R_{32-N_{in}} \dots R_{31}$	
outputs	outputs: $R_0 \dots R_{N_{out}-1}$	
constants	logic 0, logic 1	
population size	2000	
termination( $t_{max}$ )	40,000,000 tournaments	
PI crossover Prob. ( $P_{xover}$ )	0.1	
	design phase	optimization phase
bit mutation Prob. ( $P_{btmut}$ )	0.002	0.0
Sub instruction (SI). swapping Prob. ( $P_{siswp}$ )	0.0	0.5
SI. deletion Prob. ( $P_{sidel}$ )	0.0	0.1
DAOMap local search	-	yes
Dynamic Sample Weighting (DSW) (weights update freq.)	10,000 tournaments	-
preselection	yes	-
raw fitness	the ratio of unsolved training cases (= 1.0 + $f_{dp}$ )	the ratio of LUT level & LUT count (= $f_{op}$ )
success predicate	all training cases solved (= 1.0 + $f_{dp}=0.0$ )	optimize as much as possible (i.e. $f_{op} \leq 0$ )

Table 7.3: Best circuits collected from MGPPLCS, GPPLCS, DAOMap and FlowMap algorithm on six problems

Version	Type	ADD2	CMP3	MUX6	PSL6	MUL3	OCN6
MGPPLCS	LUT	<b>4</b>	<b>5</b>	<b>2</b>	<b>5</b>	<b>15</b>	<b>6</b>
	Level	<b>2</b>	<b>2</b>	<b>2</b>	<b>2</b>	<b>3</b>	<b>2</b>
	Tournament ( $\times 10^6$ )	<b>0.097</b>	<b>0.17</b>	<b>0.29</b>	<b>0.12</b>	<b>6.01</b>	<b>5.89</b>
GPPLCS	LUT	<b>4</b>	6	<b>2</b>	6	<b>15</b>	<b>6</b>
	Level	<b>2</b>	3	<b>2</b>	3	4	3
	Tournament ( $\times 10^6$ )	8.72	4.81	5.05	4.51	99.40	13.39
DAOMap	LUT	20	29	3	10	50	118
	Level	3	3	<b>2</b>	3	<b>3</b>	4
FlowMap	LUT	16	23	3	11	50	113
	Level	<b>2</b>	3	<b>2</b>	3	4	3

Table 7.4: Circuits collected from MGPPLCS, GPPLCS, DAOMap and FlowMap on six problems (Average value)

Version	Type	ADD2	CMP3	MUX6	PSL6	MUL3	OCN6
MGPPLCS	LUT	<b>4.1</b>	<b>5.05</b>	<b>2.05</b>	<b>5.1</b>	<b>15.9</b>	<b>6.45</b>
	Level	<b>2</b>	<b>2.1</b>	<b>2.2</b>	<b>2</b>	<b>3.25</b>	<b>2.2</b>
	Tournament ( $\times 10^6$ )	<b>0.18</b>	<b>0.72</b>	<b>0.70</b>	<b>0.46</b>	<b>9.32</b>	<b>8.52</b>
GPPLCS	LUT	6.1	8.15	4.1	7.45	19.1	8.1
	Level	3	4.25	2.85	3	4.85	3
	Tournament ( $\times 10^6$ )	8.63	4.42	5.10	4.30	98.94	12.84
DAOMap (Deterministic algorithm)	LUT	20	29	3	10	50	118
	Level	3	3	2	3	3	4
FlowMap (Deterministic algorithm)	LUT	16	23	3	11	50	113
	Level	2	3	2	3	4	3

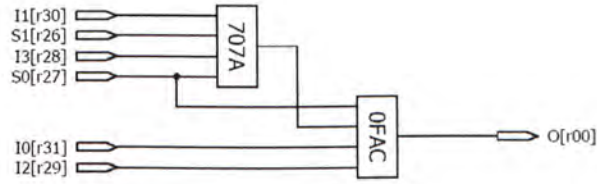


Figure 7.4: 6-bit multiplexer evolved by the MGPPLCS

used in MGPPLCS are always smaller than that in GPPLCS by 1 order of magnitude. Although MGPPLCS may not always get a better circuit than GPPLCS in all six problems, MGPPLCS performs well on average. In the 3-bit comparator problem (CMP3), the best circuit evolved from MGPPLCS is 1 4-LUT and 1 LUT level less than the one from GPPLCS and so does the case in 6-bit priority selector (PSL6). Figure 7.4 shows the 3-bit multiplier.

MGPPLCS shows a perfect synergy between GPPLCS and DAOMap. The population based GPPLCS provides DAOMap with a group of diversified Boolean circuits with the same functionality while DAOMap returns a better mapping solutions than GPPLCS.

## 7.5 Chapter Summary

In this chapter, we have presented a Memetic Genetic Parallel Programming Logic Circuit Synthesizer (MGPPLCS). It makes use of a Genetic Parallel Programming Logic Circuit Synthesizer (GPPLCS) and DAOMap algorithm. MGPPLCS applies a two-stage approach to evolve a 2-LUT circuit. During the second stage, a local search operator - DAOMap is applied to refine individuals. Experimental results show that MGPPLCS improves the performance of GPPLCS. The qualities of evolved circuits are the best among the three methods (MGPPLCS, GPPLCS and DAOMap).

---

**End of chapter.**

## Chapter 8

### Conclusion

This thesis has presented a novel Genetic Parallel Programming based Logic Circuit Synthesizer (GPPLCS) designed for tackling technology mapping problems in the automatic logic circuit synthesis optimization. It consists of two core components, an Evolution Engine (EE) and a Multi Logic Unit Processor (MLP). The EE is responsible for the genetic operations, the control strategies and the application specific processes. The MLP is a general-purpose, multiple instruction-streams multiple data-streams (MIMD) register machine which is implementable on modern commercial Field Programmable Gate Arrays (FPGAs). GPP evolves genetic programs in a specific parallel format (MLP programs).

Four improvements have been proposed and implemented to improve the GPPLCS. In Chapter 4, a hardware design and implementation of a Multi Logic Unit Processor (MLP) has been shown. In order to execute parallel genetic programs for fitness evaluation in hardware, the hardware based MLP has been proposed and implemented. Experimental results show that evolving combinational logic circuits can be sped up with a cooperation of software version EE and the hardware MLP. Speedup ratios varied from 10 to 36 are obtained in the hardware-assisted GPPLCS compared with the pure software version GPPLCS.

In Chapter 5, a new model of cooperation between multi MLP

and EE have been proposed. This new architecture of GPPLCS (MMGPPLCS) is designed for optimal logic circuit synthesis in FPGAs. It has one EE and several MLPs. Simulation results show that the performance of MMGPPLCS is nearly the same as that of the current GPPLCS in terms of the number of tournaments but expecting time for each tournament can be reduced significantly.

In Chapter 6, a Hybridized GPPLCS (HGPPLCS) has been presented. By integrating the GPPLCS and the FlowMap algorithm, better circuits can be found. We first evolve circuits in 2-input lookup table (2-LUT) and rely on FlowMap to give circuits with a 4-LUT format. Experimental results show that both the lookup table counts and the propagation delays of the circuits collected are better than those obtained by conventional design or evolved by GPPLCS alone.

We have gone one step further in Chapter 7. A novel Memetic GPPLCS (MGPPLCS) has been proposed and implemented. DAOMap is included in GPPLCS as a non-genetic local search operator. It is shown that better circuits with smaller number of LUTs and shorter propagation delay are evolved with a smaller number of tournaments.

## 8.1 Future work

This work can be improved or extended in two main directions.

With the success of MMGPPLCS, a hardware implementation of GPPLCS is a feasible way to speed up the evolution process. A full-scale hardware based GPPLCS system can be implemented in the latest FPGAs for speeding up design phase. The increased clock rates (550 MHz) in the latest generation of FPGA, Virtex-5 compared with Virtex-E (133 MHz) in Pilchard enable us to achieve a faster hardware design of MLP. Since we have already got a hardware implemented MLP, we need to

design and implement a hardware evolution engine to perform genetic operations.

Both HGPPLCS and MGPPLCS give us a possibility to solve some benchmark problems in technology mapping problems. In the current moment, it takes a few hours to evolve a solution program for difficult problems. With the speedup in both design phase and optimization phase, some large scale real life problems such as five-input XOR function in MCNC benchmark problems can be solved.

---

□ End of chapter.



# Bibliography

- [1] *UCLA RASP FPGA/CPLD Technology Mapping and Synthesis Package*. [http://ballade.cs.ucla.edu/software\\_release/rasp/htdocs/](http://ballade.cs.ucla.edu/software_release/rasp/htdocs/).
- [2] *Virtex E Platform FPGAs: Introduction and Overview*, Xilinx, Inc. 2002.
- [3] M. Abd-El-Barr, S. M. Sait, B. A. B. Sarif, and U. Al-Saiari. A modified ant colony algorithm for evolutionary design of digital circuits. In R. Sarker, R. Reynolds, H. Abbass, K. C. Tan, B. McKay, D. Essam, and T. Gedeon, editors, *Proceedings of the 2003 Congress on Evolutionary Computation CEC2003*, pages 708–715, Canberra, 8-12 Dec. 2003. IEEE Press.
- [4] A. H. Aguirre, B. P. Buckles, and C. A. C. Coello. A genetic programming approach to logic function synthesis by means of multiplexers. In *Evolvable Hardware*, pages 46–53. IEEE Computer Society, 1999.
- [5] P. J. Angeline. Two self-adaptive crossover operators for genetic programming. In P. J. Angeline and K. E. Kinneary, Jr., editors, *Advances in Genetic Programming 2*, chapter 5, pages 89–110. MIT Press, Cambridge, MA, USA, 1996.
- [6] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming: An Introduction: On the*

- Automatic Evolution of Computer Programs and Its Applications*. Heidelberg and San Francisco CA, resp., 1998.
- [7] R. K. Brayton, G. D. Hachtel, C. T. McCullen, and A. L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer.
- [8] S. M. Cheang, K. H. Lee, and K. S. Leung. Applying sample weighting methods to genetic parallel programming. In R. Sarker, R. Reynolds, H. Abbass, K. C. Tan, B. McKay, D. Essam, and T. Gedeon, editors, *Proceedings of the 2003 Congress on Evolutionary Computation CEC2003*, pages 928–935, Canberra, 8-12 Dec. 2003. IEEE Press.
- [9] S. M. Cheang, K. H. Lee, and K. S. Leung. Evolving data classification programs using genetic parallel programming. In R. Sarker, R. Reynolds, H. Abbass, K. C. Tan, B. McKay, D. Essam, and T. Gedeon, editors, *Proceedings of the 2003 Congress on Evolutionary Computation CEC2003*, pages 248–255, Canberra, 8-12 Dec. 2003. IEEE Press.
- [10] S. M. Cheang, K. H. Lee, and K. S. Leung. Designing optimal combinational digital circuits using a multiple logic unit processor. In M. Keijzer, U.-M. O'Reilly, S. M. Lucas, E. Costa, and T. Soule, editors, *Genetic Programming 7th European Conference, EuroGP 2004, Proceedings*, volume 3003 of *LNCS*, pages 23–34, Coimbra, Portugal, 5-7 Apr. 2004. Springer-Verlag.
- [11] S. M. Cheang, K. H. Lee, and K. S. Leung. Use of genetic parallel programming to design multi-output combinational logic circuits. In *2nd Intl. Conf. Artificial Intelligence in Engineering and Technology (ICAIET'2004), Proceedings*, pages 828–835, 2004.

- [12] S. M. Cheang, K. S. Leung, and K. H. Lee. Genetic parallel programming: Design and implementation. *Evolutionary Computation*, pages 129–156, 2006.
- [13] D. Chen and J. Cong. DAOmap: a depth-optimal area optimization mapping algorithm for FPGA designs. In *ICCAD*, pages 752–759, 2004.
- [14] K.-C. Chen, J. Cong, Y. Ding, A. B. Kahng, and P. Trajmar. DAG-map: Graph-based FPGA technology mapping for delay optimization. *IEEE Design & Test of Computers*, 9(3):7–20, 1992.
- [15] C. A. Coello, A. D. Christiansen, and A. H. Aguirre. Using genetic algorithms to design combinational digital circuits. In *Smart Engineering Systems: Neural Networks, Fuzzy Logic and Evolutionary Programming*, pages 391–396, 1996.
- [16] C. A. Coello, A. D. Christiansen, and A. H. Aguirre. Automated design of combinational logic circuits using genetic algorithms. In *Int. Conf. Artificial Neural Nets and Genetic Algorithms (ICANNGA97)*, pages 335–338, 1997.
- [17] C. A. Coello, A. D. Christiansen, and A. H. Aguirre. Use of evolutionary techniques to automate the design of combinational circuits. In *Int. J. Smart Engineering System Design*, pages 299–314, 2000.
- [18] C. A. C. Coello, E. Alba, G. Luque, and A. H. Aguirre. Comparing different serial and parallel heuristics to design combinational logic circuits. In *Evolvable Hardware*, pages 3–12. IEEE Computer Society, 2003.
- [19] C. A. C. Coello, E. H. Luna, and A. H. Aguirre. Use of particle swarm optimization to design combinational logic

- circuits. In A. M. Tyrrell, P. C. Haddow, and J. Torrens, editors, *Evolvable Systems: From Biology to Hardware, Fifth International Conference, ICES 2003*, volume 2606 of *LNCS*, pages 398–409, Trondheim, Norway, 17–20 Mar. 2003. Springer-Verlag.
- [20] C. A. C. Coello, R. L. Zavala, B. M. García, and A. H. Aguirre. Ant colony system for the design of combinational logic circuits. 2000.
- [21] J. Cong and Y. Ding. Flowmap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs. Jan. 13 1994.
- [22] J. C. Culberson. On the futility of blind search: An algorithmic view of “No free lunch”. *Evolutionary Computation*, 6(2):109–127, 1998.
- [23] L. Davis, editor. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, 1991.
- [24] M. Dorigo and G. Di Caro. The ant colony optimization meta-heuristic. In D. Corne, M. Dorigo, and F. Glover, editors, *New Ideas in Optimization*, pages 11–32. McGraw-Hill, London, 1999.
- [25] D. E. Goldberg, K. Deb, H. Kargupta, and G. Harik. Rapid, accurate optimization of difficult problems using fast messy genetic algorithms. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 56–64, San Mateo, CA, 1993. Morgan Kaufman.
- [26] D. E. Goldberg and S. Vössner. Optimizing global-local search hybrids. In *GECCO*, pages 220–228, 1999.

- [27] H. Hemmi, J. Mizoguchi, and K. Shimohara. Development and evolution of hardware behaviors. *Lecture Notes in Computer Science*, 1062:250–265, 1996.
- [28] M. I. Heywood and A. N. Zincir-Heywood. Register based genetic programming on FPGA computing platforms. In R. Poli, W. Banzhaf, W. B. Langdon, J. Miller, P. Nordin, and T. C. Fogarty, editors, *Proceedings of the Third European Conference on Genetic Programming (EuroGP-2000)*, volume 1802 of *LNCS*, pages 44–59, Edinburgh, Scotland, 2000. Springer Verlag.
- [29] T. Higuchi, H. Iba, and B. Manderick. Evolvable hardware. In *Massively Parallel Artificial Intelligence*, pages 399–421. MIT Press, Combridege, MA, 1994.
- [30] T. Higuchi, T. Niwa, T. Tanaka, H. Iba, H. de Garis, and T. Furuya. Evolving hardware with genetic learning: A first step toward building a darwin machine. In *Proc. 2nd Int. Conf. Simulation Adaptive Behavior (SAB92)*, pages 417–424, 1992.
- [31] H. Hirsh, W. Banzhaf, J. R. Koza, C. Ryan, L. Spector, and C. Jacob. Genetic programming. *IEEE Intelligent Systems*, 15(3):74–84, May-June 2000.
- [32] J.-H. Hong and S.-B. Cho. MEH: Modular evolvable hardware for designing complex circuits. In R. Sarker, R. Reynolds, H. Abbass, K. C. Tan, B. McKay, D. Essam, and T. Gedeon, editors, *Proceedings of the 2003 Congress on Evolutionary Computation CEC2003*, pages 92–99, Canberra, 8-12 Dec. 2003. IEEE Press.
- [33] H. Iba, M. Iwata, and T. Higuchi. Gate-level evolvable hardware: empirical study and application. In *Evolution-*

- ary Algorithms in Engineering Applications*, pages 259–276, 1997.
- [34] H. Iba, M. Iwata, and T. Higuchi. Machine learning approach to gate-level evolvable hardware. *Lecture Notes in Computer Science*, 1259:327–343, 1997.
- [35] I. Kajitani, T. Hoshino, M. Iwata, and T. Higuchi. Variable length chromosome GA for evolvable hardware. In *International Conference on Evolutionary Computation*, pages 443–447, 1996.
- [36] T. Kalganova. An extrinsic function-level evolvable hardware approach. In R. Poli, W. Banzhaf, W. B. Langdon, J. F. Miller, P. Nordin, and T. C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'2000*, volume 1802 of *LNCS*, pages 60–75, Edinburgh, 15-16 Apr. 2000. Springer-Verlag.
- [37] T. Kalganova, J. F. Miller, and T. C. Fogarty. Some aspects of an evolvable hardware approach for multiple-valued combinational circuit design. In M. Sipper, D. Mange, and A. Perez-Urbe, editors, *Evolvable Systems: From Biology to Hardware Second International Conference, ICES '98*, volume 1478 of *LNCS*, pages 78–89, Lausanne, Switzerland, Sept. 23-25 1998. Springer-Verlag.
- [38] J. Kennedy and R. C. Eberhart. Particle swarm optimization. In *Proc. of the IEEE Int. Conf. on Neural Networks*, pages 1942–1948, Piscataway, NJ, 1995. IEEE Service Center.
- [39] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

- [40] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.
- [41] J. R. Koza, F. H. Bennett III, J. L. Hutchings, S. L. Bade, M. A. Keane, and D. Andre. Rapidly reconfigurable field-programmable gate arrays for accelerating fitness evaluation in genetic programming. In J. R. Koza, editor, *Late Breaking Papers at the 1997 Genetic Programming Conference*, pages 121–131, Stanford University, CA, USA, 13–16 July 1997. Stanford Bookstore.
- [42] P. Leong, M. Leong, O. Cheung, T. Tung, C. Kwok, M. Wong, and K. Lee. Pilchard - a reconfigurable computing platform with memory slot interface. In *8th Annual IEEE Symposium on Field Programmable Custom Computing Machines, FCCM, 2001*, 2001.
- [43] K. S. Leung, K. H. Lee, and S. M. Cheang. Evolving parallel machine programs for a Multi-ALU processor. In D. B. Fogel, M. A. El-Sharkawi, X. Yao, G. Greenwood, H. Iba, P. Marrow, and M. Shackleton, editors, *Proceedings of the 2002 Congress on Evolutionary Computation CEC2002*, pages 1703–1708. IEEE Press, 2002.
- [44] K. S. Leung, K. H. Lee, and S. M. Cheang. Parallel programs are more evolvable than sequential programs. In E. C. C. Ryan, T. Soule, M. Keijzer, E. Tsang, R. Poli, editor, *Proceedings of the Sixth European Conference on Genetic Programming (EuroGP-2003)*, volume 2610 of *LNCS*, pages 107–118, Essex, UK, 2003. Springer Verlag.
- [45] S. J. Louis. *Genetic Algorithms as a Computational Tool for Design*. PhD thesis, Department of Computer Science, Indiana University, Aug. 1993.

- [46] S. J. Louis. Genetic learning for combinational logic design. *Soft Comput*, 9(1):38–43, 2005.
- [47] S. W. Mahfoud. Crowding and preselection revisited. Technical Report IlliGAL Report No 92004, University of Illinois, Urbana, 1992.
- [48] P. Martin. A pipelined hardware implementation of genetic programming using FPGAs and Handel-C. In J. A. Foster, E. Lutton, J. Miller, C. Ryan, and A. G. B. Tettamanzi, editors, *Genetic Programming, Proceedings of the 5th European Conference, EuroGP 2002*, volume 2278 of *LNCS*, pages 1–12, Kinsale, Ireland, 3-5 Apr. 2002. Springer-Verlag.
- [49] J. F. Miller. An empirical study of the efficiency of learning boolean functions using a cartesian genetic programming approach. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1135–1142, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.
- [50] J. F. Miller and P. Thomson. Aspects of digital evolution: Evolvability and architecture. In A. E. Eiben, T. Bäck, M. Schoenauer, and H.-P. Schwefel, editors, *Parallel Problem Solving from Nature – PPSN V*, pages 927–936, Berlin, 1998. Springer. Lecture Notes in Computer Science 1498.
- [51] J. F. Miller and P. Thomson. Cartesian genetic programming. In R. Poli, W. Banzhaf, W. B. Langdon, J. Miller, P. Nordin, and T. C. Fogarty, editors, *Proceedings of the Third European Conference on Genetic Programming (EuroGP-2000)*, volume 1802 of *LNCS*, pages 121–132, Edinburgh, Scotland, 2000. Springer Verlag.



- [52] J. F. Miller and V. K. Vassilev. Principles in the evolutionary design of digital circuits — part I, Oct. 28 2000.
- [53] P. Moscato. On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms. Technical Report C3P 826, Caltech Concurrent Computation Program, California Institute of Technology, Pasadena, CA, 1989.
- [54] M. Murakawa, S. Yoshizawa, I. Kajitani, and T. Furuya. Hardware evolution at function level. *Lecture Notes in Computer Science*, 1141:62–71, 1996.
- [55] A. Nicholson. Evolution and learning for digital circuit design, Apr. 17 2000.
- [56] B. Shackelford, G. Snider, R. J. Carter, E. Okushi, M. Yasuda, K. Seo, and H. Yasuura. A high-performance, pipelined, FPGA-based genetic algorithm machine. *Genetic Programming and Evolvable Machines*, 2(1):33–60, 2001.
- [57] K. Shahill. *VHDL for Programmable Logic*. Addison Wesley, 1998.
- [58] J. Torresen. Scalable evolvable hardware applied to road image recognition. In *Evolvable Hardware*, pages 245–252. IEEE Computer Society, 2000.
- [59] J. Torresen. A scalable approach to evolvable hardware. *Genetic Programming and Evolvable Machines*, 3(3):259–282, 2002.
- [60] K. H. Tsoi. Pilchard user reference (v1.0). 2004.
- [61] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, Apr. 1997.

- [62] Xilinx. Programmable logic design quick start handbook. 2006.
- [63] X. Yao and T. Higuchi. Promises and challenges of evolvable hardware. *IEEE Transactions on Systems, Man, and Cybernetics, Part C*, 29(1):87–97, 1999.

## List of Publications

1. W.S. Lau, G. Li, K.H. Lee, K.S. Leung and S.M. Cheang:  
Multi-logic-Unit Processor: A Combinational Logic Circuit  
Evaluation Engine for Genetic Parallel Programming,  
Proceedings of the 8th European Conference on Genetic  
Programming, Lecture Notes in Computer Science, Vol.  
3447, pp. 167-177, Springer, 30 March - 1 April 2005.
2. W.S. Lau, K.H. Lee and K.S. Leung:  
A Hybridized Genetic Parallel Programming Logic Circuit  
Synthesizer,  
Proceedings of the 8th annual conference on Genetic and  
Evolutionary Computation, pp. 839 - 846, Seattle, Wash-  
ington, USA.



CUHK Libraries



004366709