Applying Genetic Parallel Programming to Synthesize Combinational Logic Circuits

Sin Man Cheang, *Senior Member, IEEE*, Kin Hong Lee, *Senior Member, IEEE*, and Kwong Sak Leung, *Senior Member, IEEE*

***Abstract*—Experimental results show that parallel programs can be evolved more easily than sequential programs in genetic parallel programming (GPP). GPP is a novel genetic programming paradigm which evolves parallel program solutions. With the rapid development of lookup-table-based (LUT-based) field programmable gate arrays (FPGAs), traditional circuit design and optimization techniques cannot fully exploit the LUTs in LUT-based FPGAs. Based on the GPP paradigm, we have developed a combinational logic circuit learning system, called GPP logic circuit synthesizer (GPPLCS), in which a multilogic-unit processor is used to evaluate LUT circuits. To show the effectiveness of the GPPLCS, we have performed a series of experiments to evolve combinational logic circuits with two- and four-input LUTs. In this paper, we present eleven multi-output Boolean problems and their evolved circuits. The results show that the GPPLCS can evolve more compact four-input LUT circuits than the well-known LUT-based FPGA synthesis algorithms.**

***Index Terms*—Circuit design, digital circuits, evolvable hardware, genetic programming (GP), parallel programming.**

I. Introduction

GENETIC programming (GP) [4], [5], [52], [53] is a branch of evolutionary computation that evolves solutions in computer-program form. GP uses a many-to-one genotype-phenotype mapping [83] that increases evolvability [85]. There are two major streams in GP, tree structured GP (tree-based GP) [52] and linear structured GP (linear GP) [6], [35], [69]. In tree-based GP, a genetic program is represented in a tree structure. Genetic operators (e.g., crossover and mutation) are used to manipulate branches and leaf nodes of program trees. In linear GP, a genetic program is represented in a linear list of program instructions. Genetic operators manipulate instructions, opcodes, and operands directly. Based on linear GP, we have developed a genetic parallel programming (GPP) paradigm [14] in which multiple instructions perform multiple operations in parallel.

To design a modern computer with millions of logic gates, we need efficient CAD tools. We have well-developed optimization techniques based on Boolean algebra to optimize combinational logic circuits. However, these techniques are restricted by assumptions such as gate types and circuit structures. In recent years, a new type of reprogrammable very large scale integration (VLSI) device, known as lookup-table (LUT)-based field programmable gate arrays (FPGAs) [81], has been developed rapidly and is used widely in prototyping and medium-volume products. LUT-based FPGAs use LUTs to implement Boolean functions. Since the conventional sum-of-products (SOPs) and product-of-sums (POSs) representations of Boolean functions usually cannot make full use of the advantages of LUTs, we propose a GPP system that is able to evolve compact LUT circuits from a truth table.

In the last decade, evolvable hardware (EHW) [36], [37], [39], [74], [84] has become an important topic in automatic circuit design. EHW refers to a piece of hardware that can change its architecture to adapt to its environment. EHW uses bio-inspired methods, e.g., genetic algorithms (GAs) [32], [40], GP, etc., to design hardware [10], [27], [34], [45], [46], [54], [63], [67]. There are two categories of EHW: 1) extrinsic EHW [48] simulates evolution by software, i.e., chromosomes are evaluated on a software simulator of its hardware, and only the best chromosome is downloaded to the hardware and 2) intrinsic EHW [75] performs evolution directly in its hardware and every chromosome is downloaded to the hardware. EHW can also be subdivided into gate-level [16], [36], [62] and function-level [38], [66] representations. Gate-level EHW evolves solutions for simple problems with low-level logic gates, whereas function-level EHW evolves solutions for more complicated problems with high-level functional units. One of the usages of EHW is to learn combinational circuits directly from truth tables [19], [20], [43], [64], [76], [78]. Most EHW researches adopt two-dimensional (2-D), fixed geometric phenotypes to represent combinational circuits. The main disadvantage of using a fixed geometric phenotype is that it needs to determine the geometry before starting the evolution. Determining a suitable geometry for a Boolean problem is not a trivial task because the complexity of the problem is usually unknown. Using too large a geometry increases the search space unnecessarily, whereas using too small a geometry is inadequate to solve the problem.

In this paper, we propose a combinational circuit learning system—the GPP logic circuit synthesizer (GPPLCS). We use a variable-length parallel program structure to represent combinational circuits in order to preserve introns (instructions which do not contribute to the final output of a genetic program) in the early stages. Although introns do not affect the function of the final solution program, research results show that the existence

Manuscript received July 4, 2005; revised October 30, 2005 and July 12, 2006. This work was supported in part by the Research Grants Council of Hong Kong Special Administrative Region, China under Grant CUHK4192/03E and Grant CUHK4127/04E.

S. M. Cheang is with the Department of Computing, Hong Kong Institute of Vocational Education, Kwai Chung, Hong Kong SAR, China (e-mail: smcheang@vtc.edu.hk).

K. H. Lee and K. S. Leung are with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Shatin, Hong Kong SAR, China (e-mail: khlee@cse.cuhk.edu.hk; ksleung@cse.cuhk.edu.hk).

Digital Object Identifier 10.1109/TEVC.2006.884044

of introns in genetic programs in the early and middle stages of a run can benefit evolution [3]. Nevertheless, programs bloat continuously in the final stage. We tackle the bloating problem with a two-stage approach [2], [17], [47]. In the first stage, the GPPLCS aims at finding a 100% functional program (correct program) using a fitness function which is concerned only with the functional correctness of genetic programs. The first stage terminates as soon as the first correct genetic program is evolved. In the second stage, the GPPLCS uses another set of genetic operators together with an optimization-oriented fitness function to improve the qualities of correct programs.

The main purpose of this paper is to show the effectiveness and applicability of the GPPLCS. We have performed experiments on 11 benchmark Boolean problems. Experimental results show that the GPPLCS can synthesize and optimize multi-output LUT circuits automatically. The qualities of the evolved LUT circuits are comparable to those using conventional design methods.

The rest of this paper is organized as follows: Section II states the problem; Section III gives a brief overview of the GPP and its MAP; Section IV describes the GPPLCS; Section V gives details of the experiments and their settings; Section VI presents the results; and finally, Section VII presents our conclusions.

II. Background of Problem

This paper investigates the problem of designing a compact combinational circuit by using two- and four-input LUTs (2-LUTs and 4-LUTs). Although a Boolean function can be represented in many different forms, the most popular one is the SOP form. Frequently, the goal of logic synthesis is to find a SOP form in the simplest way by reducing the numbers of product terms and literals. Boolean algebra can be used to simplify Boolean expressions; however, the algebraic simplification process depends entirely on a designer's intuition, and the final solutions are seldom unique.

A combinational circuit is a feed-forward hardware gate network which implements a Boolean function. In order to improve the performance and/or reduce the production and running costs, additional optimization criteria (e.g., power consumption, propagation gate delay, etc.) are usually taken into account. Unfortunately, it is very difficult to find such minimal expressions/circuits or to prove that a given expression/circuit is minimal [9]. In spite of this, many systematic techniques for combinational circuit optimization have been developed. The two most famous Boolean function simplification techniques are the Karnaugh Map [49] and the Quine–McCluskey Algorithm [65], [71].

A Karnaugh Map represents a Boolean function in a 2-D array of cells. It can be used to minimize a Boolean function in SOP form. The Karnaugh Map has two disadvantages: 1) there are too many different methods of groupings 1s on a Karnaugh map which generate different Boolean expressions and 2) the size of a map grows exponentially to the number of input variables. The Quine–McCluskey Algorithm is an exhaustive search method which systematizes the procedures used to simplify a Boolean function. It can be programmed easily. Theoretically speaking, the Quine–McCluskey Algorithm is useful for Boolean functions with any number of input variables. Practically, it is only useful for Boolean functions with a small number of input variables. Besides the Karnaugh Map and the Quine–McCluskey Algorithm, there are other heuristic methods (e.g., ESPRESSO [7]) that can be used to produce minimal (or near-minimal) SOP expressions. Although the SOP form is a widely accepted form for most of the Boolean functions, it is not an effective representation form for some Boolean functions such as n-parity functions. There are some other algorithms (e.g., Reed–Muller canonical form [33]) that can handle XOR operators. However, a disadvantage of these approaches is their exponential complexity.

The optimization algorithms mentioned above can produce combinational circuits in a two-level minimal SOP form. However, in some cases, the main design concern is to minimize the gate count rather than the propagation gate delay. For example, the propagation gate delay of a BCD-to-seven-segment LED-display decoder is not the main concern because the outputs of the circuit are used to drive LED-display for human vision. If the main objective is to minimize the gate count, it is often better to design a multilevel combinational circuit. To design a compact multilevel combinational circuit, we need to factorize and decompose a Boolean function into multiple levels of subfunctions. A sophisticated heuristic minimization algorithm for multilevel logic synthesis has been proposed in [8]. This algorithm aims to reduce the number of literals in a multilevel Boolean expression.

A. Multilevel LUT Circuit

Conventional combinational circuit design techniques are based on some assumptions of restrictions (e.g., gate types and the circuit structure). The invention of LUT-based FPGA redefines the rules. LUT-based FPGA is a novel type of reprogrammable VLSI. It has been developed rapidly and used widely in prototyping circuits or products with higher-cost but lower-volume productions. One of the major differences between LUT-based FPGAs and the conventional programmable logic devices is that the former use k-input lookup-tables (k-LUTs), whereas the latter is based on a fixed AND–OR (or OR–AND) matrix that implements Boolean functions. A k-LUT is a $2^k \times 1$-bit memory module, which uses the address lines as its k inputs and returns the contents of the addressed location as the output of a Boolean function. The Boolean function is implemented by loading different bit patterns into the k-LUT. In other words, a k-LUT can implement any k-input-1-output Boolean function with a fixed hardware cost and gate delay. Unfortunately, conventional representation forms of Boolean functions (e.g., SOP form) usually cannot maximize the advantages of k-LUTs. It was shown that for $k \geq 4$, the problem of area-optimal k-LUT mapping is NP-hard [28].

The simplest way to accomplish the task is to divide a large truth table into a number of smaller truth tables so that they can be stored in k-LUTs individually. Then, the outputs of these k-LUTs are merged by multiplexers [78] (see Fig. 1).

As shown in Fig. 1, the four low-order inputs $(I_1$–$I_4)$ are used as the four address lines of all 4-LUTs. The outputs of the 4-LUTs are selected by 2-to-1 multiplexers which are configured as a binary tree. The remaining inputs $(I_5$–$I_n)$ are connected to the multiplexer network to select outputs from the

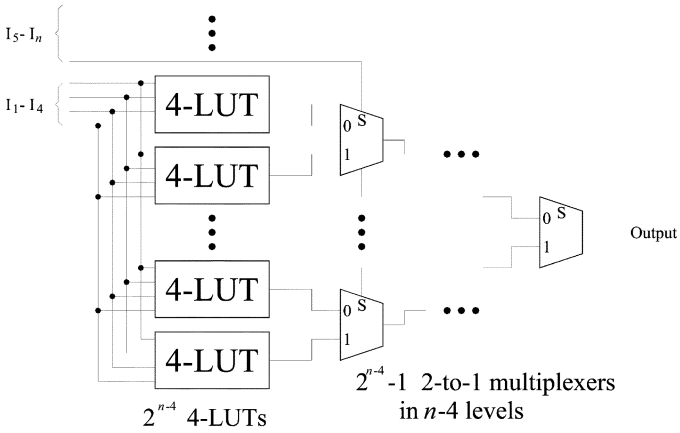

Fig. 1. Implementation of an n-input Boolean function with only 4-LUTs and 2-to-1 multiplexers.

4-LUTs. Since a 2-to-1 multiplexer has three inputs and one output, it can also be implemented by a 4-LUT. Therefore, the total number of 4-LUTs needed to implement an n-input-m -output truth table is given by

$$G = m(2^{(n-3)} - 1) \tag{1}$$

and the number of LUT delay levels is given by

$$D = n - 3. \tag{2}$$

As shown in (1), the LUT count is $O(2^n)$ to the number of inputs (n).

Many LUT logic synthesis techniques have been proposed to generate LUT circuits for LUT-based FPGAs [24]. Most techniques include two steps, Logic Optimization and Technology Mapping. The Logic Optimization aims to produce an equivalent gate-level circuit that will possess a good mapping solution according to one or several mapping objectives, such as smaller numbers of LUTs (area minimization) or LUT-levels (delay minimization). ESPRESSO and DOGMA [26] are commonly used minimization algorithms for Logic Optimization. The Technology Mapping aims to cover the optimized gate-level circuit with k-LUTs based on the structural representation of the circuit. Existing LUT-based mapping algorithms can be roughly divided into three major categories, according to their optimization objectives: 1) area minimization algorithms include Chortle [29], Chortle-crf [30], MIS-pga [68], Xmap [50], Vismap [82], and TechMap [73]; 2) delay minimization algorithms include Chortle-d [31], TechMap-L [73], and FlowMap [22]; and 3) delay and area minimization algorithms include FlowMap-r [23] and CutMap [25], FlowSYN [21], and DAOMap [15]. Amongst these algorithms, the FlowMap family algorithms, i.e., FlowMap-r, FlowSYN, etc., were shown to be superior to the others in terms of both execution performance and resulting LUT circuits [21], [22].

In this paper, we propose a GPP system that is able to evolve more compact LUT circuits from truth tables. Comparisons of the results of our GPP system and the FlowMap family algorithms are given in Section VI.

B. Evolving Combinational Circuits

Different research groups have proposed various bio-inspired methods for multilevel combinational circuit design. Simple genetic algorithms (SGAs) [16], [41], [67], [77] encode a combinational circuit by using a fixed-length chromosome. Variable-length genetic algorithms (VGA) [43], [44] are extension of SGAs. A chromosome encodes only the effective part of the architecture bits of a combinational circuit. Compared with SGAs, VGA chromosomes are smaller. Thus, it is possible for VGAs to evolve larger circuits in a shorter evolutionary time. Tree-based GP [2], [52] uses a tree structure to represent an individual combinational circuit. Aside from these, evolutionary strategy [64], ant colony algorithms [1], [18], particle swarm optimization [19], genetic algorithms with simulated annealing [51], and case injected genetic algorithms [60] have also been used to design combinational circuits.

Most of the existing representations for combinational circuits adopt 2-D geometric structures. Higuchi *et al.* [37] adopted the structure of a programmable logic device (PLD) to evolve combinational circuits. A PLD consists of a fused array and an OR logic cell. The fused array can be programmed to represent product terms of a Boolean function. Multiple product terms are connected to the OR logic cell. In Cartesian GP [63], a combinational circuit is represented by a 2-D array of logic gates. Each gate has some inputs and one output. All inputs to the circuit and outputs of the gates can be connected to their higher level gates. The final outputs can be extracted from any initial inputs of the circuit and/or outputs of gates at any levels. Louis [60] and Coello *et al.* [16] have adopted a 2-D array of two-input logic gates. Except for the first level gates, a gate $G(i, j)$ (the ith gate in the jth level) gets its first input from $G(i, j-1)$ and second 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. This representation reduces the genotype length by restricting the connectivity of a circuit. Torresen [77] has used a different gate array which relaxes the restrictions imposed in Louis's array. A gate input can be connected to any one gate output at its previous level. Murakawa *et al.* have proposed a function-based FPGA $(\mathrm{F^2PGA})$ [66] to evolve hardware solutions for calculation-intensive applications such as digital signal processing [38]. In a $\mathrm{F^2PGA}$, there are multiple layers of programmable floating processing units that can perform different high-level mathematical functions.

III. Genetic Parallel Programming (GPP)

Until now, only a few GP researchers have investigated parallel program representations. One example is the Paragen system [72] that uses tree-based GP to parallelize a sequential program to a parallel program. Based on linear GP, we have developed a GPP paradigm [14] that evolves parallel programs directly. There are some similarities between GPP and parallel distributed genetic programming (PDGP) [70]. Both GPP and PDGP are suitable for a high degree of parallelism. They can efficiently and effectively reuse partial results. PDGP represents programs in direct graphs without using genotype-phenotype mapping. It uses sophisticated crossover and mutation to manipulate subgraphs. GPP represents programs in a linear list of parallel instructions with a specific genotype-phenotype

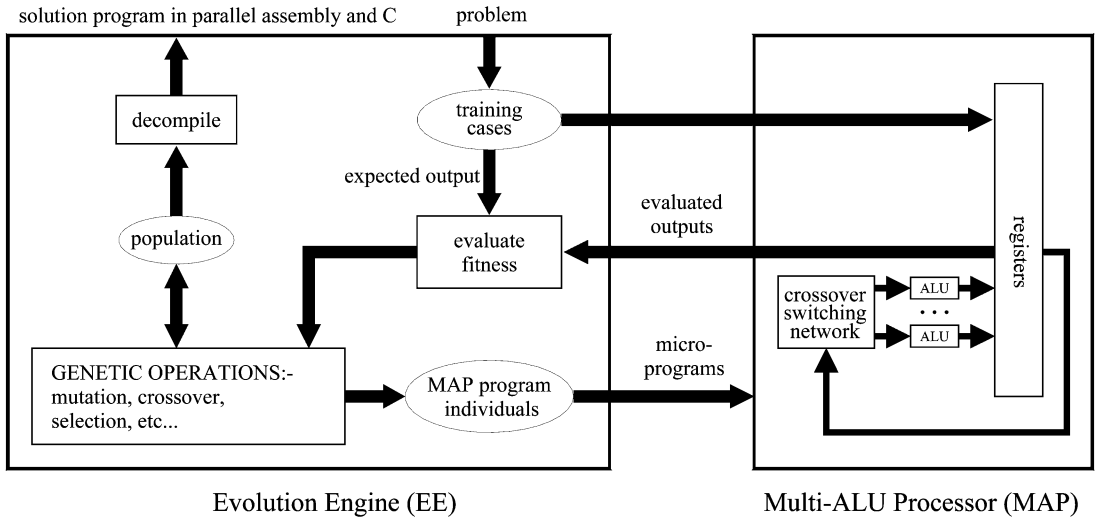

Fig. 2. The block diagram of GPP.

mapping. It uses simple and efficient genetic operators (e.g., bit mutation) to manipulate genotypes directly.

We have used GPP to evolve compact parallel programs to solve different types of problems. Some results have been published in a number of papers. In [57] and [58], we have presented the details of the GPP paradigm together with a number of benchmark problems which include numeric function regression, artificial ant, and recursive definition functions. In [12], we have presented an application of GPP to learn parallel programs to classify medical data. In [11] and [59], we have reported on preliminary studies on the GPP accelerating phenomenon. It reveals that parallel programs can be evolved with less computational effort relative to their sequential counterparts. In the summary paper [14], we have presented the details of the GPP paradigm and a sophisticated investigation on the GPP accelerating phenomenon by using 14 benchmark problems. A GPP system consists of an evolution engine (EE) and a multi-arithmetic-logic-unit (multi-ALU) processor (MAP), as shown in Fig. 2.

A. Evolution Engine (EE)

The EE performs all genetic operations, manipulates genetic programs in the population, and loads genetic programs to the MAP for fitness evaluation. The EE produces solutions in a parallel assembly program form (MAP program). An example of a MAP program, which calculates the *Fibonacci* sequence [57], is shown in Fig. 29. The program was evolved on a 4-ALU MAP. The input i and output $Fib(i)$ are stored in R_0 and R_8, respectively. The parallel instruction "0:" performs decement-by-1 (*dec*), increment-by-1 (*inc*), and addition (*add*) in parallel in different ALUs. The branch subinstruction "*jgt alu0 0*" performs a conditional branch.

B. Multi-Arithmetic-Logic-Unit (multi-ALU) Processor (MAP)

The MAP is a general-purpose, tightly coupled, multiple instruction-streams multiple data-streams (MIMD) architecture (see Fig. 3). It is designed to allow execution of multiple operations in parallel in each clock cycle. Since MAP is designed

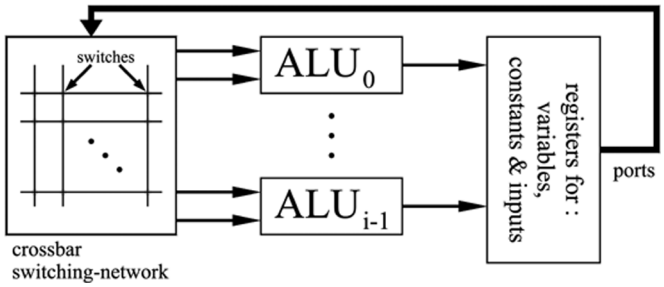

Fig. 3. The block diagram of the MAP.

for genetic program evaluation, it is able to interpret an arbitrary bit pattern as a valid instruction without causing processor fatal errors. This closure property is especially important for GPP because of its random nature, based on GP. In other words, genetic programs generated from the EE can be executed without preevaluation error correction. It speeds up the evolutionary process significantly.

The MAP consists of four core components: ALUs, registers, ports, and a crossbar switching-network. The ALUs perform arithmetic or logic operations specified in their function sets. Each ALU receives register values through the crossbar switching-network and ports and writes back a result to a variable register. The output register sets of ALUs are mutually exclusive. This prevents multiple ALUs from writing to a single register simultaneously. Each ALU also maintains status flags which are used to determine program flows. The registers are further subdivided into variable and read-only registers. Variable registers store working variables that can be written by ALUs. Read-only registers store program constants and inputs which are preloaded by the EE. The ports transfer values from registers to the inputs of ALUs through a crossbar switching-network. In each clock cycle, each port selects and transfers the value of a register. The crossbar switching-network distributes port values to ALUs. All switches in the crossbar switching-network are programmable (either closed or opened) so that it can provide enough flexibility to share port values.

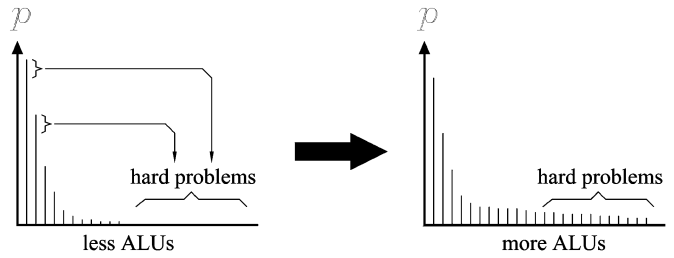

Fig. 4. Flattening of the probability distribution of different problems. p is the probability of a given problem which can be solved by a randomly generated MAP program.

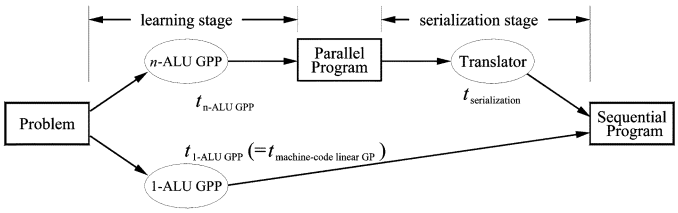

Fig. 5. Two-stage evolution with GPP.

C. GPP Accelerating Phenomenon

For a given problem, writing a fully optimal parallel program that can make use of the parallelism of the MAP is a difficult task. It involves four complicated tasks: 1) write a sequential algorithm for the problem; 2) map out the dependences of sequential instructions of the sequential algorithm; 3) resolve processor resources constraints (e.g., the numbers of ports and ALUs); and 4) assign multiple subinstructions to parallel instructions. This gives us an initial impression that evolving a parallel program is more difficult than evolving a sequential program. To our surprise, experimental results contradict our intuition. After performing a series of preliminary experiments on some benchmark problems with different numbers of ALUs, we observed that the evolutionary efficiency (in terms of both Koza's computational effort [52] and wall-clock evolutionary time) increases when the number of ALUs increases [14]. We call this *GPP accelerating phenomenon*. For further investigation into the phenomenon, a random search experiment was performed. The experimental results revealed that the parallel program representation based on the MAP could flatten the probability distribution of Boolean problems [11] (see Fig. 4).

In other words, using more ALUs in a MAP can increase the probabilities of finding solutions to hard problems. The parallel program representation slightly redistributes the probabilities from easy problems to hard problems. This phenomenon creates a novel approach to evolving a sequential (1-ALU) program in two stages: 1) a learning stage to evolve a highly parallel (n-ALU) program with the GPP in less evolution time ($t_{n\text{-ALU GPP}} < t_{1\text{-ALU GPP}}$) and 2) a serialization stage to serialize the evolved parallel program into a sequential program (see Fig. 5).

The architecture of a 1-ALU MAP is equivalent to the architecture of a single-ALU register machine commonly used in machine-code linear GP. Thus, evolving programs in 1-ALU GPP is the same as evolving programs in machine-code linear GP, i.e., $t_{1\text{-ALU GPP}} = t_{\text{machine-code linear GP}}$. The serialization stage is straightforward and spends linear processing time ($t_{\text{serialization}}$) with respect to the size of the parallel program. Relative to the evolution time $t_{n\text{-ALU GPP}}$ spent in the learning stage, $t_{\text{serialization}}$ is insignificant. Thus, the whole evolutionary process can be sped up greatly [see (3)]

$$t_{n\text{-ALU GPP}} + t_{\text{serialization}} < t_{\text{machine-code linear GP}}. \quad (3)$$

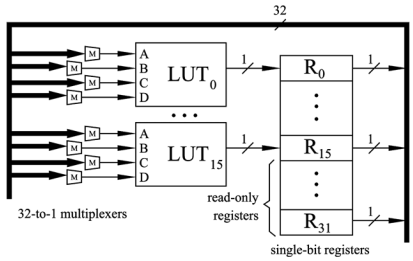

Fig. 6. A 4-MLP used to perform all 4-LUT experiments in this paper.

TABLE I
THE 16 BOOLEAN FUNCTIONS THAT CAN BE REPRESENTED BY A 2-LUT. EACH FUNCTION IS DENOTED BY "bX," WHERE X IS THE HEXADECIMAL VALUE OF ITS CONTENTS

	i/p	address							i/p	address					
	A	1	1	0	0				A	1	1	0	0		
	B	1	0	1	0	func	symbols		B	1	0	1	0	func	symbols
b0		0	0	0	0	0	0	b8		1	0	0	0	AB	8
b1		0	0	0	1	$\overline{A+B}$	1	b9		1	0	0	1	$\overline{A \oplus B}$	9
b2		0	0	1	0	$\overline{A}B$	2	bA		1	0	1	0	B	A
b3		0	0	1	1	$\overline{A}$	3	bB		1	0	1	1	$\overline{A}+B$	B
b4		0	1	0	0	$A\overline{B}$	4	bC		1	1	0	0	A	C
b5		0	1	0	1	$\overline{B}$	5	bD		1	1	0	1	$A+\overline{B}$	D
b6		0	1	1	0	$A \oplus B$	6	bE		1	1	1	0	A+B	E
b7		0	1	1	1	$\overline{AB}$	7	bF		1	1	1	1	1	1

IV. GPP WITH A MULTILOGIC-UNIT PROCESSOR (MLP)

Based on the architecture of the MAP, we developed a new Boolean function processor, known as a MLP [13].

A. Multilogic-Unit Processor (MLP)

By eliminating the crossbar switching-network and replacing all ALUs by k-LUTs in the MAP, we obtain a k-LUT MLP (k-MLP). For example, a 4-MLP is shown in Fig. 6. It consists of 16 4-LUTs, 16 variable registers (R_0–R_{15}), and 16 read-only registers (R_{16}–R_{31}).

In a MLP, each variable register can only be modified by a dedicated LUT (e.g., as shown in Fig. 6, LUT_i writes to R_i only). In each processor clock cycle, multiple LUTs take input values from registers and then perform Boolean operations simultaneously. Finally, LUTs write single-bit results to their corresponding output variable registers. For example, the 4-MLP shown in Fig. 6 can perform up to 16 operations in each clock cycle, and 16 intermediate results can be carried forward to subsequent parallel instructions through variable registers. The

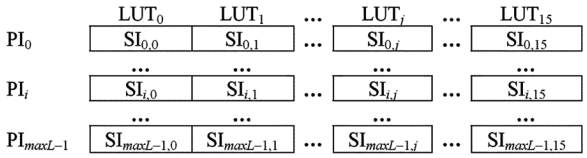

Fig. 7. The genotype of a $maxL$-parallel-instruction (PI_0-$\mathrm{PI}_{\max L-1}$) 16-subinstruction ($\mathrm{SI}_{*,0}$-$\mathrm{SI}_{*,15}$) MLP program. $\mathrm{SI}_{i,j}$ denotes the subinstruction for LUT_j in the ith parallel-instruction (PI_i).

function of a k-LUT can be changed by modifying its contents. For a 2-LUT, 2^4 different Boolean functions can be implemented (see Table I). For a 4-LUT, 2^{16} different Boolean functions can be implemented.

Since MLP is designed to manipulate single-bit logic values, the width of data buses and registers are small (relative to MAP). For example, the 4-MLP shown in Fig. 6 occupies 32 single-bit registers, 16 4-LUTs, 64 32-to-1 multiplexers, and 64 single-bit data paths. We have implemented a 4-MLP on a Xilinx XCV1000E FPGA. The FPGA-based 4-MLP runs at 100 MHz. Experimental results shown that the FPGA-based 4-MLP can speed up the pure software GPPLCS by over 20 times [55]. Since the FPGA-based 4-MLP utilizes only 20% of the hardware resources available in an XCV1000E, four 4-MLPs can be placed on a single XCV1000E. All the four 4-MLPs can be driven by a single MLP program to evaluate multiple training cases in parallel. With the rapid development of FPGAs, more MLPs can be implemented on new FPGA devices. The actual wall-clock evaluation time of a MLP program (P) can be estimated by

$$t_P = L_P \times \left\lceil \frac{N_{\mathrm{row}}}{N_{\mathrm{MLP}}} \right\rceil \div C_{\mathrm{MLP}} \tag{4}$$

where

L_P	program length (in parallel instructions) of P;
C_{MLP}	processor clock speed (in Hz);
N_{row}	number of rows in the truth table;
N_{MLP}	is the number of MLPs.

In (4), the ratio $N_{\mathrm{row}}/N_{\mathrm{MLP}}$ calculates the number of iterations needed to evaluate a MLP program for all rows in a truth table. The maximum speedup will be achieved when $N_{\mathrm{MLP}} \geq N_{\mathrm{row}}$.

B. Genotype-Phenotype Mapping

A MLP is a generic evaluation engine for LUT circuit design. Theoretically speaking, any k-LUT circuit can be represented by a k-MLP program. Like a MAP, a MLP can accept an arbitrary bit pattern (genotype) as a valid program (phenotype) without causing processor fatal errors. The genotype of a genetic parallel program can be loaded and executed in the MLP directly without preevaluation correction. This saves unnecessary processing time. Fig. 7 shows the genotype of a MLP program for a MLP with 16 LUTs. It is represented by a sequence of parallel instructions.

TABLE II
CONTROL CODES IN 2- AND 4-MLP SUBINSTRUCTIONS

fields	no. bits	encoding
2-MLP opcode	5	00000 – 01111 => b0 – bF 10000 – 11111 => nop
4-MLP opcode	17	00…0 – 01…1 => b0000 – bFFFF 10…0 – 11…1 => nop
operand	5	00000 – 11111 => R_0 – R_{31}

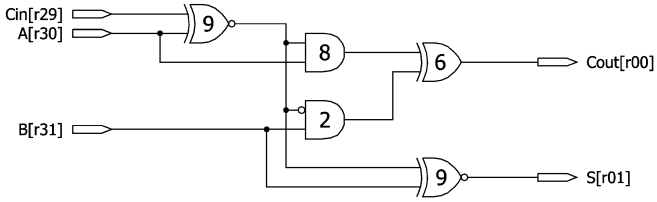

Fig. 8. The 1-bit binary full-adder (ADD1) shown in Fig. 30.

For a 2-MLP, each subinstruction consists of a 5-bit opcode and two 5-bit operands (see Table II). Since the 2-MLP consists of 16 2-LUTs, a total of 240 bits $((5 + 2 \times 5) \times 16)$ are used to encode a parallel instruction. If we allow at most 25 parallel instructions ($maxL = 25$) in a MLP program, the genotype may contain up to 6000 (240 × 25) bits. For a 4-MLP, there are 2^{16} different Boolean functions for a 4-LUT. Each subinstruction consists of a 17-bit opcode and four 5-bit operands (see Table II). Similar to the 2-MLP case, if the maximum program length is 25 parallel instructions, the genotype may contain up to 14 800 $((17 + 4 \times 5) \times 16 \times 25)$ bits.

In this paper, we represent phenotypes of MLP programs in parallel assembly form. A 2-MLP program example of a 1-bit binary full-adder is shown in Fig. 30. It consists of two sections, a *#data* section and a *#program* section. The *#data* section defines constant, input, and output variables. Before starting an execution, a MLP always initializes all variable registers (i.e., R_0–R_{15}) to logic "0." The *CONSTANTS* line in the *#data* section initializes read-only registers R_{16}–R_{21} to logic "0" and R_{22}–R_{28} to logic "1." The *INPUTS* line defines input variables (i.e., *Cin*, A, and B) and assigns them to read-only registers (i.e., R_{29}, R_{30}, and R_{31}). The *OUTPUTS* line defines output variables (i.e., *Cout* and S) and assigns them to variable registers (i.e., R_0 and R_1). The *#program* section contains parallel instructions that perform Boolean operations. For example, the numbered lines in the *#program* section in Fig. 30 list out three parallel instructions. For easy interpretation, all *nop* (no-operation) subinstructions are hidden from the original 2-MLP program. In each parallel instruction, subinstructions are separated by commas. Each subinstruction consists of three parts: 1) a function name (i.e., *b0*-*bF* and *nop*); 2) input registers; and 3) an output register. For example, the "*b6 r14 r00 r00*" subinstruction in parallel instruction "2:" performs *b6* (XOR) on R_{14} and R_0, and then stores the result back to R_0. Fig. 8 shows the corresponding 2-LUT circuit of the 2-MLP program. Obviously, each 2-LUT matches to a 2-LUT subinstruction in the 2-MLP program.

Fig. 31 shows a 2-bit binary full-adder evolved by the GP-PLCS with a 4-MLP. The Boolean function of each 4-LUT subinstruction is denoted by a four-digit hexadecimal number

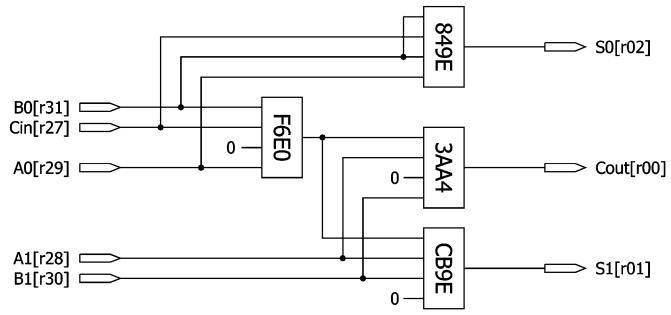

Fig. 9. The 2-bit full-adder (ADD2) shown in Fig. 31.

TABLE III
11 BENCHMARK BOOLEAN PROBLEMS. N_i, N_o, N_r, AND N_c DENOTE THE NUMBERS OF INPUTS, OUTPUTS, ROWS, AND TRAINING CASES, RESPECTIVELY, IN INDIVIDUAL TRUTH TABLES

name	description	N_i	N_o	$N_r(=2^{N_i})$	$N_c(=N_r \times N_o)$
ADD1	1-bit full-adder	3	2	8	16
ADD2	2-bit full-adder	5	3	32	96
CEX1	Coello's example 1	4	1	16	16
CEX2	Coello's example 2	5	1	32	32
CEX3	Coello's example 3	4	3	16	48
MUX6	4-to-1 multiplexer	6	1	64	64
CMP3	3-bit comparator	6	3	64	192
OCN6	6-bit one's counter	6	3	64	192
PSL6	6-bit priority selector	6	4	64	256
MUL2	2-bit multiplier	4	4	16	64
MUL3	3-bit multiplier	6	6	64	384

which represents the 16-bit memory contents of the 4-LUT. For example, as shown in Fig. 31, the "$bF6E0\ldots$" subinstruction in parallel instruction "0:" is implemented by loading "1111 0110 1110 0000" to the 16 memory-bits of the 4-LUT. The corresponding 4-LUT circuit is shown in Fig. 9.

V. EXPERIMENTS AND SETTINGS

In order to investigate the effectiveness of the GPPLCS, we have used the system to evolve LUT circuits for different benchmark Boolean problems. In this section, we describe these experiments and their settings. Table III lists eleven benchmark Boolean problems presented in this paper.

The ADD1 and ADD2 are 1- and 2-bit binary full-adders, respectively. The CEX1, CEX2, and CEX3 are Boolean problems adopted from [20]. Their truth tables are shown in Table IV. The MUX6 is a single-bit 4-to-1 multiplexer that consists of six input bits (i.e., a 4-bit data and a 2-bit selection). The CMP3 has two 3-bit binary inputs (i.e., A and B). It compares the two 3-bit binary values and gives three output bits (i.e., "$A < B$", "$A = B$" and "$A > B$"). The OCN6 counts the total number of 1's in its six inputs and represents the count in a 3-bit binary output. The PSL6 is a priority-selector which encodes six prioritized inputs to a 3-bit binary output (see Table V). The MUL2 and MUL3 are 2- and 3-bit binary multipliers, respectively.

A. The Two-Stage Approach

The main objective of this research is to evolve compact multilevel LUT circuits. As we mentioned in Section I, the existence of introns in the early and middle stages of a GP evolution is helpful. In order to evolve the correct programs with GPPLCS, we retain introns in the genetic programs until we find the first correct program. However, the first correct program is usually not a compact solution in terms of quality measurements (e.g., the LUT count and the propagation LUT delay). To tackle this problem, we adopt a two-stage (i.e., design and optimization stages) approach with a multiobjective fitness function similar to that proposed in [17] and [47]. The GPPLCS is meant to improve the functionality of the 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 multiobjective fitness function, the GPPLCS uses a different set of genetic operators in the two stages. In the design stage, the GPPLCS aims at finding a 100% functional program (correct program). Its raw fitness is given by

$$f_{ds} = \frac{\text{number of unmatched training cases}}{\text{total number of training cases}}. \quad (5)$$

The design stage raw fitness f_{ds} is used to evaluate the functional fitness of a genetic program. As shown in (5), a partially correct genetic program has a value of f_{ds} greater than zero (i.e., some unmatched training cases exist), whereas a correct program has a value of f_{ds} equal to zero (i.e., all training cases are matched). Having found the first correct genetic program, the evolution proceeds to the optimization stage to optimize the

TABLE IV
THE TRUTH TABLES OF CEX1, CEX2, AND CEX3

				CEX1	CEX3			CEX2											
i/p				o/p	o/p			i/p					o/p	i/p					o/p
A	B	C	D	F	F1	F2	F3	A	B	C	D	E	F	A	B	C	D	E	F
0	0	0	0	1	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0
0	0	0	1	1	0	1	0	0	0	0	0	1	0	1	0	0	0	1	0
0	0	1	0	0	0	1	0	0	0	0	1	0	1	1	0	0	1	0	1
0	0	1	1	1	0	1	0	0	0	0	1	1	1	1	0	0	1	1	1
0	1	0	0	0	0	0	1	0	0	1	0	0	0	1	0	1	0	0	0
0	1	0	1	0	1	0	0	0	0	1	0	1	0	1	0	1	0	1	1
0	1	1	0	1	0	1	0	0	0	1	1	0	1	1	0	1	1	0	0
0	1	1	1	1	0	1	0	0	0	1	1	1	1	1	0	1	1	1	1
1	0	0	0	1	0	0	1	0	1	0	0	0	0	1	1	0	0	0	0
1	0	0	1	0	0	0	1	0	1	0	0	1	0	1	1	0	0	1	1
1	0	1	0	1	1	0	0	0	1	0	1	0	1	1	1	0	1	0	0
1	0	1	1	0	0	1	0	0	1	0	1	1	1	1	1	0	1	1	1
1	1	0	0	0	0	0	1	0	1	1	0	0	0	1	1	1	0	0	0
1	1	0	1	1	0	0	1	0	1	1	0	1	1	1	1	1	0	1	1
1	1	1	0	0	0	0	1	0	1	1	1	0	0	1	1	1	1	0	0
1	1	1	1	0	1	0	0	0	1	1	1	1	1	1	1	1	1	1	1

TABLE V
TRUTH TABLE OF PSL6 ($\times$ = DON'T CARE)

i/p						o/p			
I5	I4	I3	I2	I1	I0	IN_E	S2	S1	S0
0	0	0	0	0	0	0	×	×	×
×	×	×	×	×	1	1	0	0	0
×	×	×	×	1	0	1	0	0	1
×	×	×	1	0	0	1	0	1	0
×	×	1	0	0	0	1	0	1	1
×	1	0	0	0	0	1	1	0	0
1	0	0	0	0	0	1	1	0	1

correct genetic program based on some optimization-oriented criteria. In the optimization stage, the raw fitness is given by

$$f_{os} = \frac{G}{\max G} + \frac{D}{\max D} \times \frac{1}{\max G} + \frac{L}{\max L} \times \frac{1}{\max D \cdot \max G}. \quad (6)$$

As shown in (6), the optimization stage raw fitness f_{os} of a correct genetic program is calculated from three qualitative indices: 1) the LUT count (G) (i.e., the total number of non-*nop* subinstructions); 2) the propagation LUT delay (D); and 3) the program length (L) in terms of the number of parallel instructions. Since a genetic program consists of *nop* and introns, the L represents the number of LUT levels in the logical circuit diagram but not the actual LUT delay in hardware. This is because the *nop* and introns will not be placed in real hardware and their LUT delays will not be counted. Thus, we have $D \leq L$. The $\max G$, $\max D$, and $\max L$ are the maximum allowed values for the LUT count, the propagation LUT delay and the program length, respectively. For example, in a tournament of two genetic parallel programs, the one with a smaller G will win. If the two genetic parallel programs have the same G value, the one with a smaller D will win. If the two genetic parallel programs have the same G and D values, the one with a smaller L will win. In other words, the main objective of the optimization stage is to reduce the LUT count, and then the propagation LUT delay. The last multiplication term in (6) guides the evolution to shorten the lengths of the correct genetic programs. Normally, a shorter program has smaller G and D values.

By combining the two-stage raw-fitness functions (f_{ds} and f_{os}), a multiobjective fitness function of the whole evolutionary process is given by

$$f = \begin{cases} 1.0 + f_{ds}, & \text{if } f_{ds} > 0 \\ f_{os}, & \text{if } f_{ds} = 0 \end{cases}. \quad (7)$$

In (7), the constant 1.0 is used to distinguish the two stages. With this combined fitness function, a partially correct genetic program has an f greater than 1.0, whereas a correct genetic program has an f less than 1.0. In the design stage, whenever the EE finds the first genetic program with an f equal to 1.0, it proceeds to the optimization stage.

B. Genetic Operators

The GPPLCS inherits most genetic operators (e.g., subinstruction swapping) from GPP. Some of these operators will only be used in either the design or the optimization stage. A brief description of these operators is given as follows.

Parallel-Instruction Level Crossover—This is a two-point crossover to exchange two segments of parallel instructions from two parent MLP programs. All subinstructions in a parallel instruction will always be kept as a whole, as shown in Fig. 10. The probability to take this operator is P_{pi_xovr}.

Bit Mutation—It mutates individual bits in the genotype of a MLP program based on a probability P_{bt_mut}.

Subinstruction Swapping—It swaps two subinstructions inside a MLP program based on a probability P_{si_swp} (see Fig. 11). It can pack more non-*nop* subinstructions into fewer parallel instructions so as to increase the parallelism of a MLP program. This operator will only be used in the optimization stage since its goal is to improve the performance of a correct genetic program.

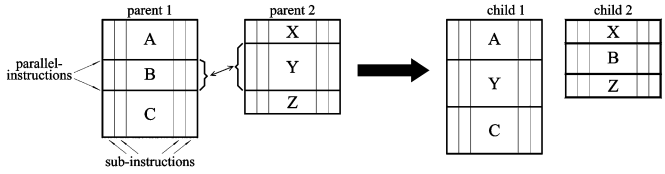

Fig. 10. Two-point parallel-instruction level crossover.

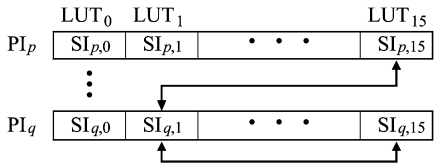

Fig. 11. Subinstruction swapping.

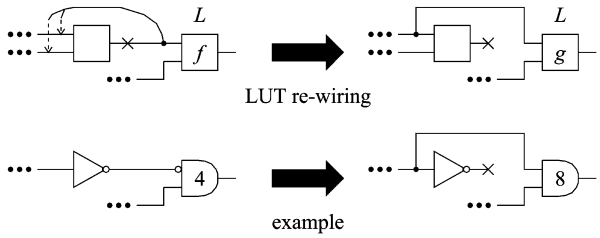

Fig. 12. LUT rewiring.

Subinstruction Deletion—This simply replaces a non-*nop* subinstruction with a *nop* subinstruction based on a probability P_{si_del}. This operator can delete inactive subinstructions (introns) from a correct genetic program, and therefore it is used only in the optimization stage.

Dynamic Sample Weighting (DSW) [56]—Training cases (samples) in a truth table are usually biased. For example, let us consider an n-input-1-output Boolean function (F_C) with exactly one minterm equal to "1," whereas all other $2^n - 1$ minterms are equal to "0." If we use the ratio of unmatched training cases as a fitness function $(f[P])$ for a genetic program (P) [see (5)], a correct genetic program (P_C) will have a fitness value equal to zero $(f[P_C] = 0)$. For a genetic program (P_Z) that represents a zero constant function (F_Z), this has an extremely good fitness $(f[P_Z] = 1/2^n)$. Obviously, it is a local optima on the fitness landscape. Although the difference of fitness $(f[P_Z] - f[P_C])$ between P_Z and P_C is only $1/2^n$, it is very difficult for the GPPLCS to evolve P_C from P_Z. This is because P_C and P_Z have a great genotype difference. DSW adjusts the weights (W_i) of training cases dynamically based on their past frequency of hits (H_i). At the beginning of every ΔT_{sw_upd} tournament, all W_i are recalculated (i.e., $W_i \propto 1/H_i$), and then all H_i are reset to zero. In other words, DSW balances the contributions of training cases to speed up the evolution. This operator is only used in the design stage.

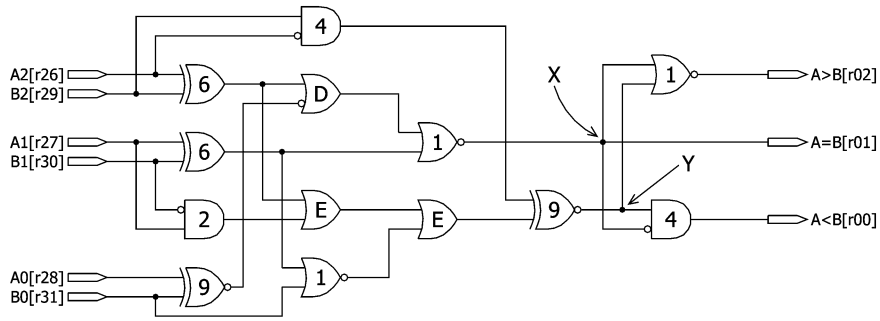

Fig. 13. The 3-bit binary comparator (CMP3) shown in Fig. 32.

Preselection [61]—In each tournament, two new genetic programs (children) will be born. The GPPLCS will test the similarity of each child to its parents. If a child is functionally equivalent to either one of its parents (i.e., it matches the same set of the training cases), it will be discarded. This operator maintains a reasonable diversity of search and is used only in the design stage.

LUT Rewiring—This is a heuristic LUT-elimination operator that intends to remove a redundant LUT in a correct circuit by rewiring an input of a LUT to one of its predecessor inputs. As shown in Fig. 12, the function of a LUT L is mutated from f to g and one of its inputs is rewired to one of its predecessor inputs. The example in Fig. 12 shows how a redundant NOT gate can be eliminated. The probability to take this operator is P_{rewir}. It is used only in the optimization stage.

C. Experiment Settings

As mentioned before, a FPGA-based MLP can significantly speed up the evolution by parallelizing fitness evaluations of multiple training cases. Since the speedup is caused purely by hardware parallelization, it does not change the fundamental functionality of the GPPLCS. Since the main purpose of this paper is to demonstrate that the GPPLCS can evolve compact LUT circuits directly from their truth tables, we ran all experiments on software emulators of MLPs, which are written in C programming language.

Table VI shows the GP parameters and the settings of all experiments presented in this paper. Having investigated the difficulties of the 11 Boolean problems shown in Table III, we set the maximum program length to 25 parallel instructions $(\max L = 25)$. This provides enough subinstructions (for both effective operations and introns) to evolve correct programs. Hence, at most, 400 (25×16) operations can be used to build a solution MLP program.

In general, we do not know the truly optimal solution circuits for individual benchmark problems. Thus, we set an unachievable success predicate $(f \leq 0.0)$ in the optimization stage (see the last row in Table VI). It forces the system to optimize correct genetic programs to the greatest extent possible. Thus, a run is set to terminate after 20 000 000 tournaments.

TABLE VI
GP PARAMETERS AND EXPERIMENTAL SETTINGS. "–" DENOTES THAT THOSE OPERATIONS ARE NOT USED IN THE SPECIFIC STAGE

	both stages	
no. of LUTs	16	
no. of variable reg.	16 (R_0 … R_{15})	
no. of read-only reg.	16 (R_{16} … R_{31})	
no. parallel-instructions	maximum 25 (*maxL*)	
initialization	bit random	
	average program length = *maxL*÷2	
selection method	tournament (size=10)	
LUT function set	2-LUTs: b0-bF, nop	
	4-LUTs: b0000-bFFFF, nop	
terminal set	inputs: $R_{32-Ninput}$ … R_{31}	
	outputs: R_0 … $R_{Noutput-1}$	
constants	logic 0, logic 1	
P_{pi_xovr}	0.1	
population size	2000	
experiments	50 independent runs	
termination (t_{max})	20,000,000 tournaments	
	design stage	**optimization stage**
P_{bt_mut}	0.002 (each bit)	2 bits (each program)
P_{si_swp}	-	0.5 (each program)
P_{si_del}	-	0.1 (each program)
ΔT_{sw_upd}	10000 tournaments	-
preselection	yes	-
P_{re_wir}	-	0.5 (each program)
fitness	$f = 1.0 + f_{ds}$	$f = f_{os}$
success predicate	$f = 1.0$ (i.e. $f_{ds} = 0.0$)	$f \leq 0.0$ (i.e. $f_{os} \leq 0.0$)

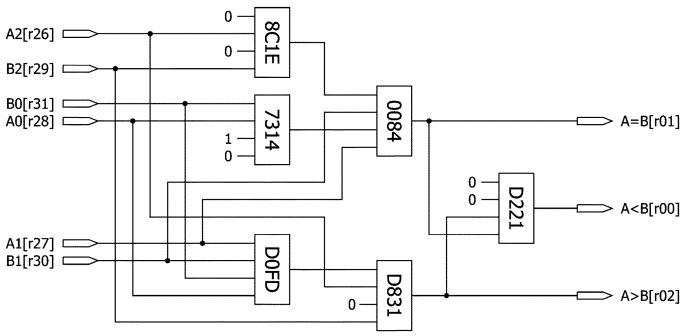

Fig. 14. The 3-bit binary comparator (CMP3) shown in Fig. 33.

VI. RESULTS AND EVALUATIONS

In this section, we detail the experimental results. From 50 runs of the GPPLCS on each of the 11 benchmark Boolean problems listed in Table III, we obtained many high-quality solutions. We have already shown two simple examples of the evolved MLP programs, i.e., the best 2-MLP program for 1-bit full-adder (ADD1) and 4-MLP program for 2-bit full-adder (ADD2) in Section IV.

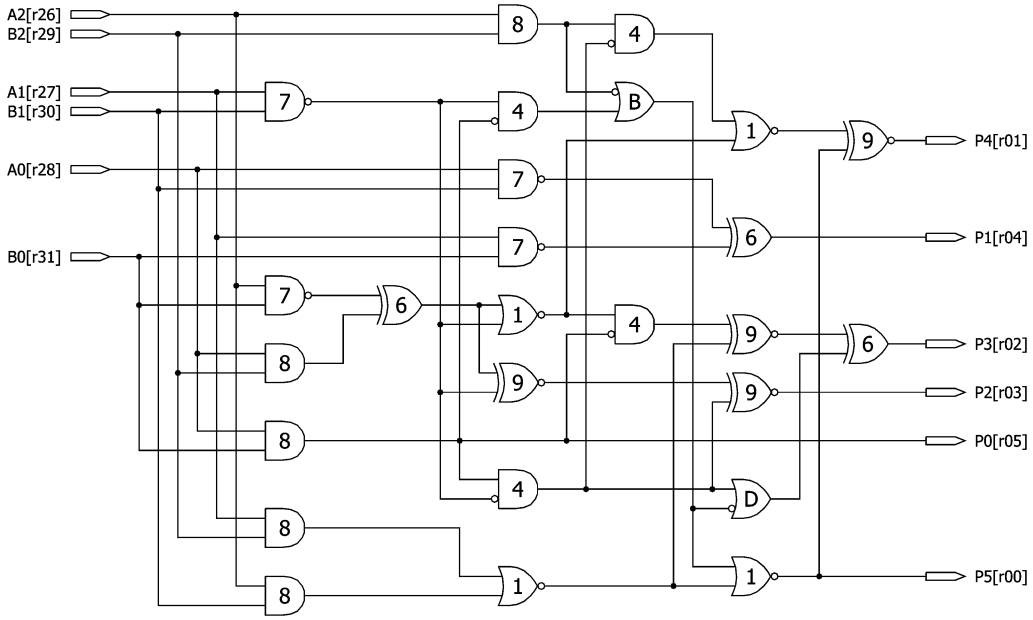

Fig. 15. The 3-bit binary multiplier (MUL3) shown in Fig. 34.

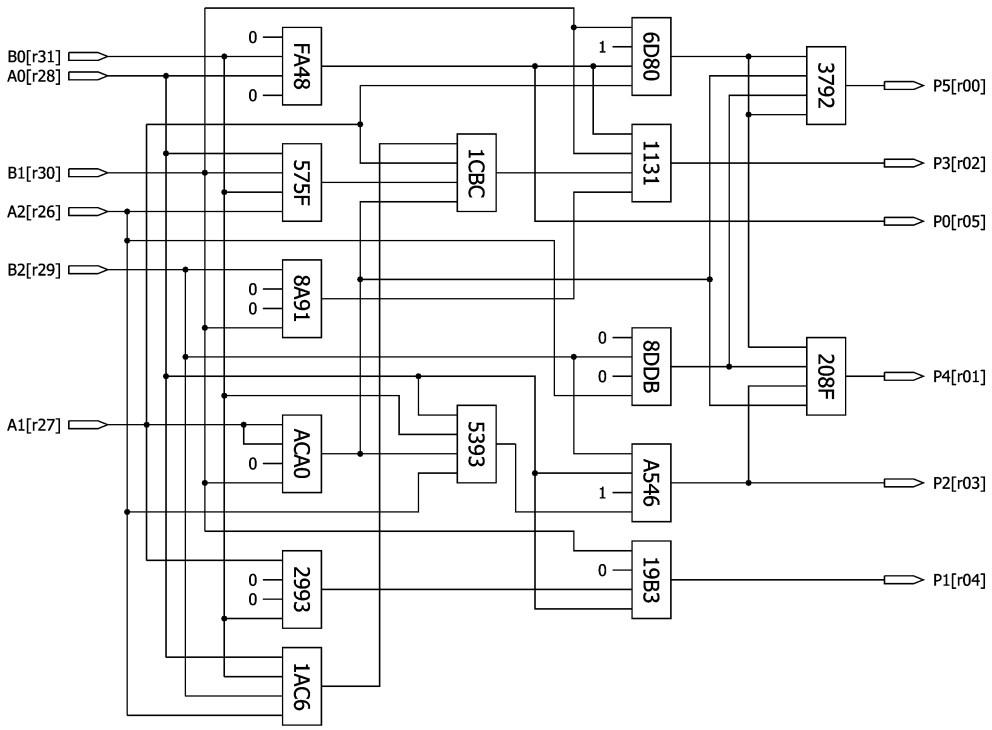

Fig. 16. The 3-bit binary multiplier (MUL3) shown in Fig. 35.

A. Evolved Solutions

This section presents the evolved MLP programs of two selected problems, the 3-bit binary comparator (CMP3) and the 3-bit multiplier (MUL3). The MLP programs of the remaining nine problems are shown in Figs. 17–28 and Figs. 36–47.

Fig. 32 shows the best 2-MLP program evolved by the GPPLCS for 3-bit comparator (CMP3). The program consists of six inputs (i.e., A0–A2, B0–B2), three outputs ("$A < B$," "$A = B$," and "$A > B$") and five parallel instructions ("0:" to "4:"). For easy interpretation, all *nop* subinstructions are hidden. The corresponding multilevel 2-LUT circuit is shown in Fig. 13. As we can see, both "$A < B$" and "$A > B$" share two common subfunctions (marked by X and Y in the figure). This circuit demonstrates that the GPPLCS can factorize and extract common factors of multiple Boolean functions. The circuit is a smart and complicated design. It is unlikely that a human designer would be able to work out such a design for CMP3. Having presented a multilevel 2-LUT circuit evolved with a 2-MLP, we shall further investigate a result with a 4-MLP. The

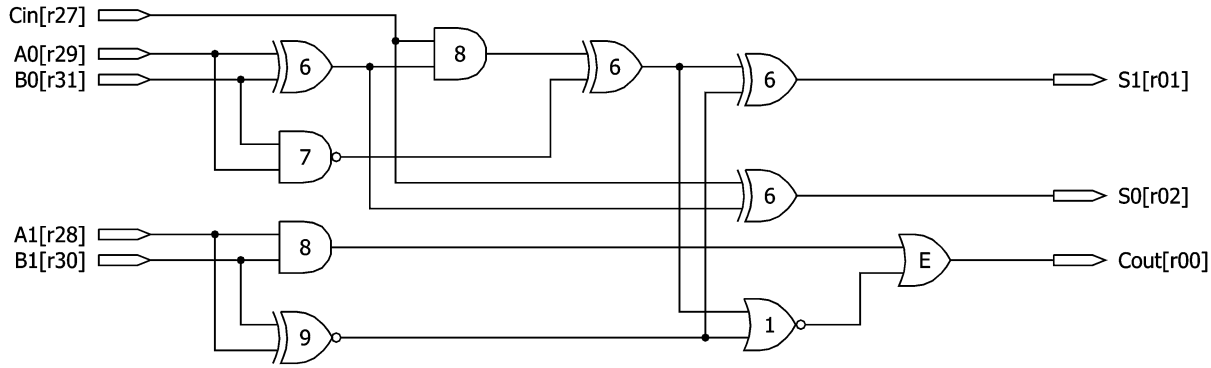

Fig. 17. The 2-bit full-adder (ADD2) shown in Fig. 36.

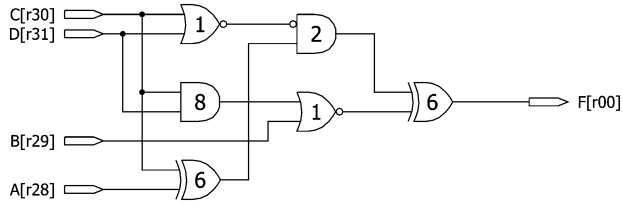

Fig. 18. Coello's example 1 (CEX1) shown in Fig. 37.

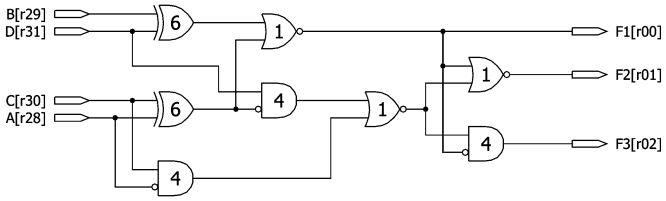

Fig. 20. Coello's example 3 (CEX3) shown in Fig. 39.

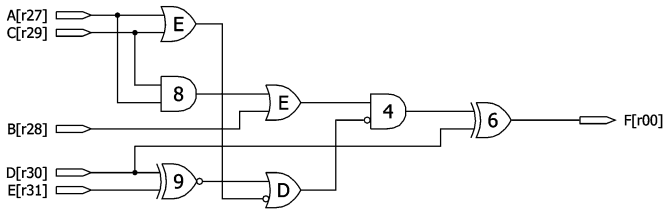

Fig. 19. Coello's example 2 (CEX2) shown in Fig. 38.

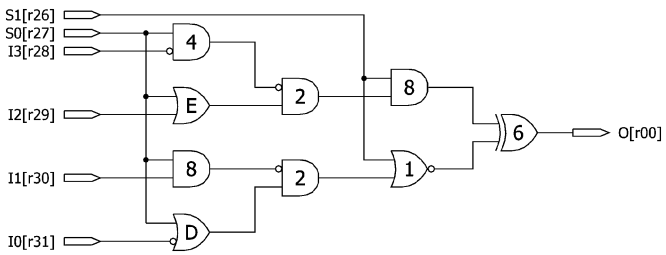

Fig. 21. The 4-to-1 multiplexer (MUX6) shown in Fig. 40.

best 4-MLP program for CMP3 evolved by the GPPLCS is shown in Fig. 33. The program consists of six 4-LUT subinstructions in three parallel instructions. The corresponding 4-LUT circuit is shown in Fig. 14.

Similar to the 2-MLP case, this is a multilevel combinational circuit in which the "$A < B$" output reuses the other outputs (i.e., "$A = B$" and "$A > B$") produced in the lower levels. By investigating the circuit shown in Fig. 14, we can find that some 4-LUTs have constant inputs (e.g., the one labeled by "7314"). These 4-LUTs can be replaced by LUTs with a smaller number of inputs. Thus, the LUT circuit can be further simplified with fewer-input LUTs.

Having presented the solutions for CMP3, we are going to describe the solutions to a more complex problem, that of the 3-bit binary multiplier (MUL3). It is generally acknowledged that binary multipliers are hard problems in evolutionary computation and are widely used to benchmark problems in the automatic circuit design and evolutionary computation communities [16], [42], [62]. A compact 2-MLP program for MUL3 is shown in Fig. 34. The program consists of 26 non-*nop* subinstructions in six parallel instructions. The program has two 3-bit binary inputs (i.e., A0–A2, B0–B2) and a 6-bit binary output (P0–P5). The corresponding 2-LUT circuit is shown in Fig. 15.

A compact 4-MLP program and the corresponding 4-LUT circuit for MUL3 are shown in Figs. 16 and 35, respectively. The program consists of 15 non-*nop* subinstructions in four parallel instructions.

B. Evaluations and Comparisons

In order to show the effectiveness of the GPPLCS, we adopt Koza's minimum computational effort measurement (E-measure) method [52]. It estimates the minimum number of genetic programs which would have to be bred and evaluated to give a certain probability of success. Since the computational effort for the initial population evaluation is small relative to the whole evolution process, we ignore this portion of the effort.

Let z be the expected probability of success, and $P(t)$ be the experimental cumulative probability of success for a run to yield satisfactory solutions at or before the tournament t. The number of independent runs required to satisfy the success predicate by tournament t with the probability of z is given by

$$R(t,z) = \begin{cases} 1.0, & \text{if } P(t) = 1.0 \\ \left\lceil \frac{\log(1-z)}{\log[1-P(t)]} \right\rceil, & \text{otherwise} \end{cases}. \quad (8)$$

The total number of tournaments that need to be processed in order to yield a satisfactory solution with a specific value of z

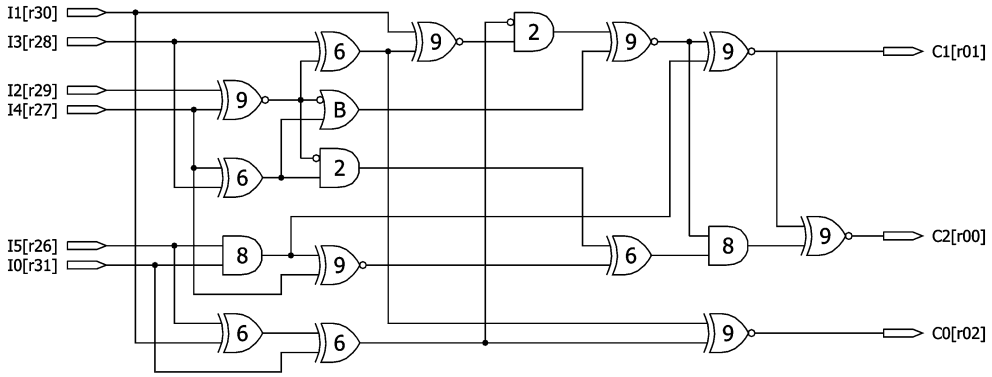

Fig. 22. The 1s counter (OCN6) shown in Fig. 41.

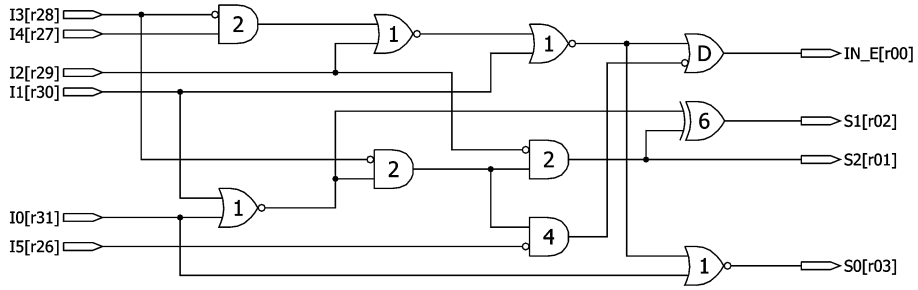

Fig. 23. The 6-bit priority-selector (PSL6) shown in Fig. 42.

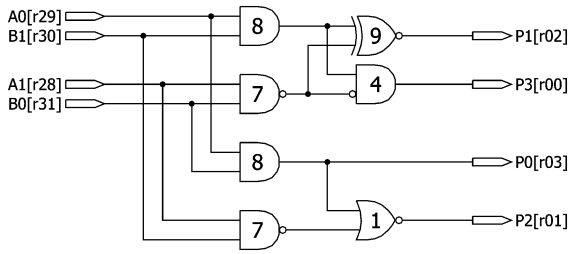

Fig. 24. The 2-bit multiplier (MUL2) shown in Fig. 43.

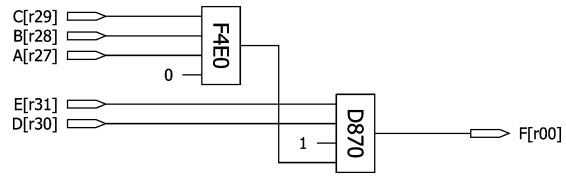

Fig. 25. Coello's example 2 (CEX2) shown in Fig. 44.

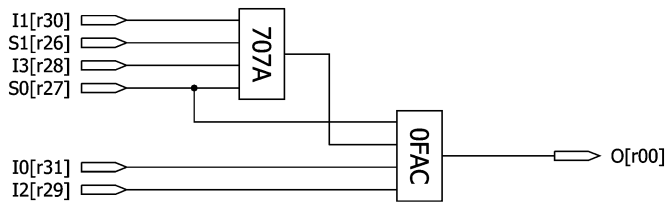

Fig. 26. The 4-to-1 multiplexer (MUX6) shown in Fig. 45.

is given by

$$T(t,z) = t\,R(t,z). \tag{9}$$

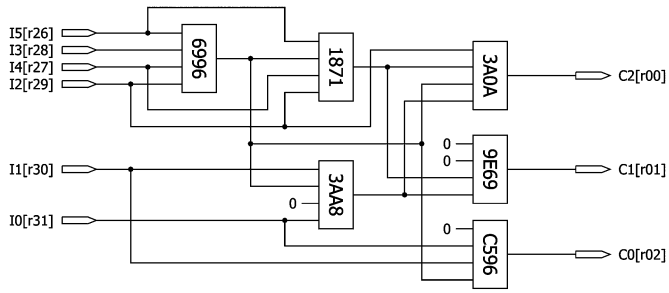

Fig. 27. The 1s counter (OCN6) shown in Fig. 46.

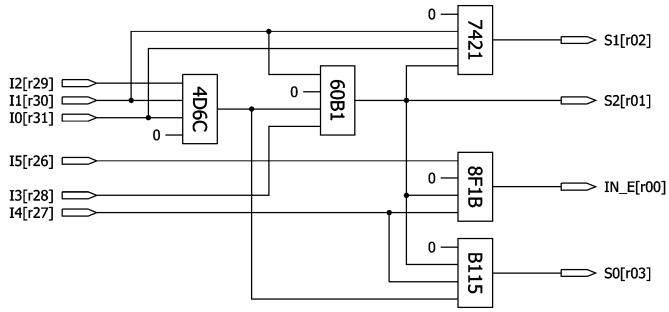

Fig. 28. The 6-bit priority-selector (PSL6) shown in Fig. 47.

The minimum number of tournaments (computational effort E) required to yield a satisfactory solution for the problem is given by

$$E = \min_{t=0}^{t_{\max}}[T(t,z)]. \tag{10}$$

```
#data
CONSTANTS: r17=1,r18=2,(r16,r19-r31)=0
INPUTS:    r0<=i
OUTPUTS:   r8=>Fib(i)
#program
#- --BRANCH-- -ALU0---- -ALU1----- -ALU2-----
#             -ALU3---------
0: jgt alu0 0,dec r0 r0,inc r14 r5,inc r14 r8,
              add r5 r14 r14
1: end       ,nop      ,nop       ,nop       ,
              nop
```

Fig. 29. A MAP program for calculating the *Fibonacci* sequence. *nop* is no-operation.

```
#data
CONSTANTS: (r16-r21)=0,(r22-r28)=1
INPUTS:    (r29,r30,r31)<=(Cin,A,B)
OUTPUTS:   (r00,r01)=>(Cout,S)
#program
0: b9 r29 r30 r04
1: b8 r04 r30 r00,b2 r04 r31 r14
2: b6 r14 r00 r00,b9 r31 r04 r01
```

Fig. 30. The best 2-MLP program for 1-bit binary full-adder (ADD1).

```
#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
0: bF6E0 r31 r27 r08 r29 r00
1: b3AA4 r00 r28 r06 r30 r00,
   bCB9E r00 r28 r30 r21 r01,
   b849E r31 r27 r31 r29 r02
```

Fig. 31. The best 4-MLP program for 2-bit binary full-adder (ADD2).

```
#data
CONSTANTS: (r16-r20)=0,(r21-r25)=1
INPUTS:    (r26,r27,r28,r29,r30,r31)
           <=(A2,A1,A0,B2,B1,B0)
OUTPUTS:   (r00,r01,r02)=>(A<B,A=B,A>B)
#program
0: b6 r29 r26 r01,b6 r27 r30 r06,b2 r30 r27 r08,
   b9 r31 r28 r10
1: bD r01 r10 r00,bE r08 r01 r02,b4 r29 r26 r08,
   b1 r31 r06 r14
2: b1 r00 r06 r01,bE r02 r14 r14
3: b9 r14 r08 r00
4: b4 r00 r01 r00,b1 r00 r01 r02
```

Fig. 32. The best 2-MLP program for 3-bit binary comparator (CMP3).

```
#data
CONSTANTS: (r16-r20)=0,(r21-r25)=1
INPUTS:    (r26,r27,r28,r29,r30,r31)
           <=(A2,A1,A0,B2,B1,B0)
OUTPUTS:   (r00,r01,r02)=>(A<B,A=B,A>B)
#program
0: b8C1E r13 r26 r19 r29 r02,
   b7314 r31 r28 r25 r16 r09,
   bD0FD r27 r30 r31 r28 r14
1: b0084 r02 r30 r09 r27 r01,
   bD831 r14 r26 r10 r29 r02
2: bD221 r15 r19 r02 r01 r00
```

Fig. 33. The best 4-MLP program for 3-bit binary comparator (CMP3).

In (10), E is equal to the minimal value of $T(t,z)$ over all the tournament t between 0 and the maximum allowed tournament $t_{\max}$.

Based on the results of 50 runs, three performance indices are calculated: 1) Koza's minimum computational effort (E) (with $z = 0.99$); 2) the LUT count (G); and 3) the propagation LUT delay (D). The experimental results for 2-MLP are shown in Table VII. For comparison, published results of Cartesian GP (CGP), conventional design (CON), and GA with

```
#data
CONSTANTS: (r16-r20)=0,(r21-25)=1
INPUTS:    (r26,r27,r28,r29,r30,r31)
           <=(A2,A1,A0,B2,B1,B0)
OUTPUTS:   (r00,r01,r02,r03,r04,r05)
           =>(P5,P4,P3,P2,P1,P0)
#program
0: b7 r27 r30 r02,b7 r26 r31 r03,b8 r31 r28 r05,
   b8 r27 r29 r07,b8 r26 r30 r09,b8 r29 r28 r15
1: b6 r15 r03 r06
2: b8 r26 r29 r00,b4 r02 r05 r01,b7 r28 r30 r02,
   b9 r06 r02 r03,b1 r09 r07 r07,b7 r31 r27 r08,
   b1 r06 r02 r10,b4 r05 r02 r15
3: b4 r00 r15 r00,bB r00 r01 r01,b4 r10 r05 r14
4: b1 r07 r01 r00,bD r15 r01 r01,b9 r03 r15 r03,
   b6 r02 r08 r04,b1 r10 r00 r11,b9 r14 r07 r13
5: b9 r11 r00 r01,b6 r13 r01 r02
```

Fig. 34. The best 2-MLP program for 3-bit binary multiplier (MUL3).

```
#data
CONSTANTS: (r16-r20)=0,(r21-25)=1
INPUTS:    (r26,r27,r28,r29,r30,r31)
           <=(A2,A1,A0,B2,B1,B0)
OUTPUTS:   (r00,r01,r02,r03,r04,r05)
           =>(P5,P4,P3,P2,P1,P0)
#program
0: bFA48 r00 r31 r28 r01 r05,
   b575F r28 r30 r31 r26 r07,
   b8A91 r29 r01 r16 r30 r09,
   bACA0 r27 r27 r01 r30 r10,
   b2993 r27 r01 r01 r31 r11,
   b1AC6 r28 r31 r29 r26 r12
1: b5393 r28 r31 r10 r26 r07,
   b1CBC r12 r27 r07 r10 r13
2: b6D80 r30 r23 r05 r27 r00,
   b1131 r05 r30 r13 r09 r02,
   bA546 r29 r28 r21 r07 r03,
   b19B3 r30 r00 r11 r28 r04,
   b8DDB r00 r29 r20 r26 r07
3: b3792 r00 r10 r07 r00 r00,
   b208F r00 r07 r03 r10 r01
```

Fig. 35. The best 4-MLP program for 3-bit multiplier (MUL3).

```
#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
0: b6 r31 r29 r00,b8 r28 r30 r06,b7 r29 r31 r08,
   b9 r28 r30 r11
1: b8 r00 r27 r01
2: b6 r08 r01 r03
3: b6 r03 r11 r01,b6 r00 r27 r02,b1 r03 r11 r14
4: bE r14 r06 r00
```

Fig. 36. The best 2-MLP program for the 2-bit full-adder (ADD2).

```
#data
CONSTANTS: (r16-r21)=0,(r22-r27)=1
INPUTS:    (r28,r29,r30,r31)<=(A,B,C,D)
OUTPUTS:   (r00)=>(F)
#program
0: b1 r30 r31 r01,b8 r30 r31 r02,b6 r30 r28 r09
1: b2 r01 r09 r02,b1 r29 r02 r08
2: b6 r08 r02 r00
```

Fig. 37. The best 2-MLP program for Coello's example 1 (CEX1).

```
#data
CONSTANTS: (r16-r21)=0,(r22-r26)=1
INPUTS:    (r27,r28,r29,r30,r31)<=(A,B,C,D,E)
OUTPUTS:   (r00)=>(F)
#program
0: bE r27 r29 r02,b8 r29 r27 r03,b9 r31 r30 r09
1: bD r09 r02 r00,bE r28 r03 r08
2: b4 r08 r00 r04
3: b6 r04 r30 r00
```

Fig. 38. The best 2-MLP program for Coello's example 2 (CEX2).

```
#data
CONSTANTS: (r16-r21)=0,(r22-r27)=1
INPUTS:    (r28,r29,r30,r31)<=(A,B,C,D)
OUTPUTS:   (r00,r01,r02)=>(F1,F2,F3)
#program
0: b6 r29 r31 r01,b6 r30 r28 r08,b4 r30 r28 r10
1: b1 r01 r08 r00,b4 r31 r08 r03
2: b1 r03 r10 r01
3: b1 r00 r01 r01,b4 r01 r00 r02
```

Fig. 39. The best 2-MLP program for Coello's example 3 (CEX3).

```
#data
CONSTANTS: (r16-r20)=0,(r21-25)=1
INPUTS:    (r26,r27,r28,r29,r30,r31)
          <=(S1,S0,I3,I2,I1,I0)
OUTPUTS:   (r00)=>(O)
#program
0: bE r29 r27 r00,b8 r30 r27 r02,b4 r27 r28 r09,
   bD r27 r31 r12
1: b2 r09 r00 r00,b2 r02 r12 r08
2: b8 r26 r00 r00,b1 r08 r26 r02
3: b6 r02 r00 r00
```

Fig. 40. The best 2-MLP program for 4-to-1 multiplexer (MUX6).

```
#data
CONSTANTS: (r16-r20)=0,(r21-25)=1
INPUTS:    (r26,r27,r28,r29,r30,r31)
          <=(I5,I4,I3,I2,I1,I0)
OUTPUTS:   (r00,r01,r02)=>(C2,C1,C0)
#program
0: b9 r29 r27 r00,b6 r27 r28 r05,b8 r26 r31 r08,
   b6 r26 r30 r14
1: b9 r08 r27 r01,bB r00 r05 r02,b6 r14 r31 r09,
   b2 r00 r05 r10,b6 r28 r00 r11
2: b9 r30 r11 r00
3: b2 r09 r00 r04
4: b9 r04 r02 r01,b6 r10 r01 r12
5: b9 r01 r08 r01,b9 r09 r11 r02,b8 r01 r12 r03
6: b9 r03 r01 r00
```

Fig. 41. The best 2-MLP program for 1s counter (OCN6).

```
#data
CONSTANTS: (r16-r20)=0,(r21-25)=1
INPUTS:    (r26,r27,r28,r29,r30,r31)
          <=(I5,I4,I3,I2,I1,I0)
OUTPUTS:   (r00,r01,r02,r03)=>(IN_E,S2,S1,S0)
#program
0: b2 r28 r27 r01,b1 r30 r31 r14
1: b1 r01 r29 r04,b2 r28 r14 r10
2: b2 r29 r10 r01,b4 r10 r26 r02,b1 r04 r30 r09
3: bD r09 r02 r00,b6 r14 r01 r02,b1 r09 r31 r03
```

Fig. 42. The best 2-MLP program for 6-bit priority-selector (PSL6).

simulated annealing (GASA) are also included. All these published results used for comparison are produced with their respective optimized system parameters. In all the cases, the figures shown in the "GPPLCS$_{ds}$ avg $G(D)$" column are very

```
#data
CONSTANTS: (r16-r21)=0,(r22-r27)=1
INPUTS:    (r28,r29,r30,r31)<=(A1,A0,B1,B0)
OUTPUTS:   (r00,r01,r02,r03)=>(P3,P2,P1,P0)
#program
0: b8 r29 r30 r00,b7 r28 r31 r02,b8 r31 r29 r03,
   b7 r28 r30 r09
1: b4 r00 r02 r00,b1 r09 r03 r01,b9 r02 r00 r02
```

Fig. 43. The best 2-MLP program for 2-bit multiplier (MUL2).

```
#data
CONSTANTS: (r16-r21)=0,(r22-r26)=1
INPUTS:    (r27,r28,r29,r30,r31)<=(A,B,C,D,E)
OUTPUTS:   (r00)=>(F)
#program
0: bF4E0 r29 r28 r27 r07 r15
1: bD870 r31 r30 r24 r15 r00
```

Fig. 44. The best 2-MLP program for Coello's example 2 (CEX2).

```
#data
CONSTANTS: (r16-r20)=0,(r21-25)=1
INPUTS:    (r26,r27,r28,r29,r30,r31)
          <=(S1,S0,I3,I2,I1,I0)
OUTPUTS:   (r00)=>(O)
#program
0: b707A r30 r26 r28 r27 r04
1: b0FAC r27 r04 r31 r29 r00
```

Fig. 45. The best 4-MLP program for 4-to-1 multiplexer (MUX6).

```
#data
CONSTANTS: (r16-r20)=0,(r21-25)=1
INPUTS:    (r26,r27,r28,r29,r30,r31)
          <=(I5,I4,I3,I2,I1,I0)
OUTPUTS:   (r00,r01,r02)=>(C2,C1,C0)
#program
0: b6996 r26 r28 r27 r29 r00
1: b1871 r26 r00 r27 r29 r07,
   b3AA8 r30 r00 r05 r31 r15
2: b3A0A r29 r07 r00 r15 r00,
   b9E69 r06 r11 r07 r15 r01,
   bC596 r19 r31 r30 r00 r02
```

Fig. 46. The best 4-MLP program for 1s counter (OCN6).

```
#data
CONSTANTS: (r16-r20)=0,(r21-25)=1
INPUTS:    (r26,r27,r28,r29,r30,r31)
          <=(I5,I4,I3,I2,I1,I0)
OUTPUTS:   (r00,r01,r02,r03)=>(IN_E,S2,S1,S0)
#program
0: b4D6C r29 r30 r31 r05 r02
1: b60B1 r30 r13 r02 r28 r01
2: b8F1B r26 r10 r01 r27 r00,
   b7421 r00 r30 r31 r01 r02,
   bB115 r00 r01 r27 r02 r03
```

Fig. 47. The best 4-MLP program for 6-bit priority-selector (PSL6).

large. It is clear that the correct genetic programs evolved in the design stage contain numerous introns. By comparing the figures in the "GPPLCS$_{ds}$ avg $G(D)$" and "GPPLCS$_{os}$ avg $G(D)$" columns, we notice that both the "avg G" and "avg D" decrease significantly in the optimization stage. For example, in the "Total" row of Table VII, the total "avg G" decreases from 2172 to 152 2-LUTs and the total "avg D" decreases from 189 to 56 levels. This demonstrates the effectiveness of the optimization stage. For ADD1, ADD2, and MUL2, the circuits evolved by the GPPLCS have the smallest G and D. For MUL3, although GPPLCS utilizes three more 2-LUTs than CGP, its delay

TABLE VII

SUMMARY OF 2-MLP EXPERIMENTAL RESULTS. THE GPPLCS_{ds} AND GPPLCS_{os} COLUMNS LIST THE RESULTS OF THE DESIGN AND OPTIMIZATION STAGES, RESPECTIVELY. THE "$\mathrm{GPPLCS}_{ds}E$" COLUMN LISTS THE MINIMUM NUMBERS OF TOURNAMENTS REQUIRED TO YIELD THE CORRECT GENETIC PROGRAMS OF THE INDIVIDUAL PROBLEMS IN THE DESIGN STAGE. THE "AVG $G(D)$" COLUMNS SHOW THE AVERAGE VALUES OF G AND D OF ALL EVOLVED CIRCUITS. THE "BEST $G(D)$" COLUMNS SHOW THE G AND D VALUES OF THE BEST-EVOLVED2-LUT CIRCUITS. THE CGP AND CON COLUMNS LIST THE PUBLISHED RESULTS OF CARTESIAN GP [64] AND CONVENTIONAL DESIGN [80], RESPECTIVELY. THE GASA COLUMN LISTS THE RESULTS OF GA WITH SIMULATED ANNEALING PUBLISHED IN [20]. THE UNDERLINED VALUES ARE THE SMALLEST IN INDIVIDUAL ROWS

	GPPLCS_{ds}			GPPLCS_{os}		CGP	CON	GASA
problems	E ($\times 10^6$)	avg $G(D)$	best $G(D)$	avg $G(D)$	best $G(D)$	best $G(D)$	best $G(D)$	best $G(D)$
ADD1	0.19	161(14)	95(9)	5(3)	5(3)	5(3)	5(3)	-
ADD2	1.33	203(17)	92(10)	12(5)	10(5)	10(6)	10(6)	-
CEX1	0.24	208(17)	123(10)	6(4)	6(3)	-	-	6(3)
CEX2	0.54	220(18)	136(12)	7(5)	7(4)	-	-	7(5)
CEX3	0.55	195(17)	104(10)	9(4)	8(4)	-	-	9(5)
MUX6	0.57	225(19)	140(12)	10(5)	9(4)	-	-	-
CMP3	1.70	207(18)	155(13)	15(6)	13(5)	-	-	-
OCN6	47.54	187(18)	143(13)	24(8)	17(7)	-	-	-
PSL6	1.17	194(17)	117(11)	12(4)	10(4)	-	-	-
MUL2	0.45	193(16)	121(9)	7(3)	7(2)	7(3)	8(3)	7(3)
MUL3	92.29	179(18)	136(16)	45(9)	26(6)	23(8)	30(8)	-
Total		2172(189)	1362(125)	152(56)	118(47)			
Ratios to the best GPPLCS_{os} $G(D)$		18.4(4.02)	11.5(2.66)	1.3(1.19)	**1.0(1.00)**			

TABLE VIII

SUMMARY OF FOUR-MLP EXPERIMENTAL RESULTS. THE PART COLUMN LISTS THE G AND D VALUES [CALCULATED BY (1) AND (2)] FOR DIRECTLY PARTITIONING INDIVIDUAL TRUTH TABLES IN 4-LUTS AND MERGING OUTPUTS OF 4-LUTS USING 2-TO-1 MULTIPLEXERS. THE FLOWMAP, FLOWMAP-r (WITH $r = 1$), FLOWSYN, CUTMAP, AND DAOMAP COLUMNS LIST RESULTS OF THE CORRESPONDING LUT MAPPING ALGORITHMS. THE UNDERLINED VALUES ARE THE SMALLEST IN INDIVIDUAL ROWS

	GPPLCS_{ds}			GPPLCS_{os}		PART	FlowMap	FlowMap-r	FlowSYN	CutMap	DAOMap
problems	E ($\times 10^6$)	avg $G(D)$	best $G(D)$	avg $G(D)$	best $G(D)$	$G(D)$	$G(D)$	$G(D)$	$G(D)$	$G(D)$	$G(D)$
ADD2	0.50	227(19)	144(13)	6(3)	4(2)	9(2)	16(3)	19(3)	6(2)	17(3)	20(3)
CEX2	0.04	239(16)	95(6)	2(2)	2(2)	3(2)	4(2)	4(2)	4(2)	4(2)	4(2)
MUX6	0.08	178(17)	65(6)	4(3)	2(2)	7(2)	3(2)	3(2)	3(2)	3(2)	3(2)
CMP3	1.80	148(16)	72(10)	8(4)	6(3)	21(3)	23(3)	23(3)	12(3)	23(3)	29(3)
OCN6	9.84	139(16)	44(6)	10(5)	6(3)	21(3)	113(4)	107(5)	8(2)	113(4)	118(4)
PSL6	0.47	205(19)	123(11)	8(3)	5(3)	28(3)	11(3)	11(3)	10(2)	11(3)	10(3)
MUL3	81.94	154(17)	105(14)	19(5)	15(4)	42(3)	50(3)	48(4)	20(3)	50(3)	50(3)
Total		1290(120)	648(66)	57(25)	40(19)	131(18)	220(20)	215(22)	63(16)	221(20)	234(20)
Ratios to the best GPPLCS_{os} $G(D)$		32.3(6.32)	16.2(3.47)	1.4(1.32)	**1.0(1.00)**	3.3(0.95)	5.5(1.05)	5.4(1.16)	1.6(0.84)	5.5(1.05)	5.9(1.05)

D is better by two—it is 6, compared with 8 in CGP. The underlined figures listed in the "GPPLCS_{os} best $G(D)$" column show that the GPPLCS can design compact combinational circuits for these problems both in terms of LUT count and propagation LUT delay.

Having presented the evolved 2-LUT circuits with 2-MLP, we are going to describe the 4-MLP results. We have selected seven problems with more than four inputs listed in Table III to test the evolution of solutions with 4-MLP. We do not test the remaining four problems because a truth table with four or fewer inputs can be loaded to a 4-LUT directly. The results of the 4-LUT and other five mapping algorithms, i.e., FlowMap, FlowMap-r (with $r = 1$), FlowSYN, CutMap, and DAOMap, are summarized in Table VIII. All experiments on the five mapping algorithms were run on the UCLA RASP FPGA/CPLD Technology Mapping and Synthesis Package [79]. First, we used ESPRESSO to optimize the truth tables of the seven Boolean problems into optimal (or near-optimal) SOP expressions. Then, the resulting SOP expressions were passed to produce 4-LUT circuits for the five LUT mapping algorithms. Obviously, all the G values in the "GPPLCS_{os} best $G(D)$" column are smaller than those of the other columns. The "Total" row in Table VIII shows that the GPPLCS produces 4-LUT circuits in the smallest total number $(= 40)$ of 4-LUTs. The FlowSYN algorithm, which has the closest performance to the GPPLCS, produces all the seven 4-LUT circuits with 1.6 times $(= 63/40)$ the total number of 4-LUTs to the GPPLCS. Since the GPPLCS used gate count minimization as its primary objective [see (6)], the total 4-LUT delay (19 levels) is slightly larger than those of the FlowMap (18 levels) and FlowSYN algorithms (16 levels) which are delay-optimized. Except for the FlowMap and the FlowSYN, 4-LUT circuits produced by GPPLCS were shown to be superior to those of FlowMap-r, CutMap, and DAOMap algorithms in terms of LUT count. The values shown in Table VIII demonstrate that the GPPLCS can evolve very compact 4-LUT circuits directly from their truth tables.

VII. CONCLUSION AND FUTURE WORK

In this paper, we have presented a GPP system, known as a GPPLCS, for combinational logic circuit design. It uses a

MLP that is based on the multi-arithmetic-logic-unit processor of GPP. The MLP is a multiple logic function unit, register-based architecture that can perform multiple logic operations in a processor clock cycle. The MLP is used to execute MLP programs, which can be used to represent any combinational logic circuits. A MLP program consists of a sequence of parallel instructions. Each parallel instruction consists of a fixed number of subinstructions (16 subinstruction MLPs are used in this paper) which perform logic functions to manipulate the contents of the registers.

The GPPLCS adopts a two-stage approach to separate the design and the optimization stages. In the design stage, the system evolves genetic programs based on a fitness function which aims at evolving a 100% functional program (correct program). In the optimization stage, the GPPLCS uses another set of genetic operators to optimize correct programs. Experimental results show that the GPPLCS can evolve compact 2-LUT and 4-LUT circuits. The qualities of the evolved circuits are in general better than existing published results. The advantages of the GPPLCS are summarized as follows.

1) It uses a generic register machine architecture which can emulate any k-LUT circuits. The architecture of MLP is so simple that numerous MLPs can be placed in a FPGA. With this highly parallelized, hardware-assisted fitness evaluation engine, the evolution will speed up significantly.
2) It employs a variable-length genotype so that introns can be built up in the early and middle stages of a run to assist evolution.
3) A 100% functional program evolved in the design stage can be saved as a seed program for different runs in the optimization stage. In the optimization stage, different optimization-oriented genetic operators and fitness criteria such as gate count and propagation gate delay are used to guide the optimization.

Experimental results show that the GPPLCS is superior to well-known LUT-based FPGA synthesis algorithms. This is because all these synthesis algorithms are based on a fixed gate-level implementation of a truth table with some primitive logic gates (e.g., AND, OR, and NOT). Building a LUT circuit with such a gate-level circuit actually limits the optimization capacities in the LUT mapping stage. The GPPLCS relaxes the restrictions on these primitive logic gates so that more compact LUT circuits can be evolved (see Table VIII).

The GPPLCS is a very flexible system. By changing the LUTs to other logic units, the system can also be used to evolve circuits for other FPGA devices.

According to the experimental results of the MUL3, the pure software GPPLCS can evolve 4-LUT circuits on any 6-input 6-output truth tables in reasonable time. However, evolving large circuits directly from a truth table is still a time-consuming problem in evolutionary computation. The actual complexity of a Boolean problem depends on the distribution of 1's and the number of "don't care" designations in the outputs of its truth table. For example, ADD2 has six times the number of outputs of ADD1 (see Table III), and the evolution time ratio is seven $(= 1.33/0.19)$ times that of ADD1 (see Table VII). However, MUL3 also has six times the number of outputs of MUL2, and the total evolution time ratio is 205 $(= 92.29/0.45)$ times.

Currently, we are building a FPGA-based multi-MLP. The initial results show that it can speed up the software simulation at least 20 times [55], and hence more complex Boolean problems (e.g., 4-bit multiplier) can be evolved. Further studies will be conducted to explore the effect of different settings on MLP configurations (e.g., the numbers of registers and inputs of LUTs).

REFERENCES

[1] 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 *Proc. Congr. Evol. Comput.*, 2003, pp. 708–715.

[2] A. H. Aguirre, C. A. Coello, and B. P. Buckles, "A genetic programming approach to logic function synthesis by means of multiplexers," in *Proc. 1st NASA/DoD Workshop Evolvable Hardware* , 1999, pp. 46–53.

[3] P. J. Angeline, "Two self-adaptive crossover operators for genetic programming," in *Advances in Genetic Programming 2*. Cambridge, MA: MIT Press, 1996, pp. 89–110.

[4] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone*, Generic Programming: An Introduction on the Automatic Evolution of Computer Programs and its Applications*. San Francisco, CA: Morgan Kaufmann, 1998.

[5] W. Banzhaf, J. R. Koza, C. Ryan, L. Spector, and C. Jocob, "Genetic programming," *IEEE Intell. Syst.*, vol. 17, no. 3, pp. 74–84, 2000.

[6] M. Brameier and W. Banzhaf, "A comparison of linear genetic programming and neural networks in medical data mining," *IEEE Trans. Evol. Comput.*, vol. 5, no. 1, pp. 17–26, Feb. 2001.

[7] K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli*, Logic Minimization Algorithms for VLSI Synthesis*. Norwell, MA: Kluwer, 1984.

[8] R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli, "Multilevel logic synthesis," *Proc. IEEE*, vol. 78, pp. 264–300, Feb. 1990.

[9] J. Brzozowski and M. Yoeli*, Digital Networks*. Englewood Cliffs, NJ: Prentice-Hall, 1976.

[10] S. J. Chang, H. S. Hou, and Y. K. Su, "Automated passive filter synthesis using a novel tree representation and genetic programming," *IEEE Trans. Evol. Comput.*, vol. 10, no. 1, pp. 93–100, Feb. 2006.

[11] S. M. Cheang, K. H. Lee, and K. S. Leung, J. Foster, Ed., "An empirical study of the accelerating phenomenon in genetic parallel programming," in *Proc. 5th Conf. Genetic Evol. Comput. (Late-Breaking Papers)*, 2003, pp. 54–61.

[12] ——, "Evolving data classification programs using genetic parallel programming," in *Proc. 2003 Congr. Evol. Comput.*, 2003, pp. 248–255.

[13] ——, "Designing optimal combinational digital circuits using a multiple logic unit processor," in *Proc. 7th Eur. Conf. Genetic Program.*, Berlin, Germany, 2004, vol. 3003, pp. 23–34.

[14] S. M. Cheang, K. S. Leung, and K. H. Lee, "Genetic parallel programming: Design and implementation," *Evol. Comput. J.*, vol. 14, no. 2, pp. 129–156, 2006.

[15] D. Chen and J. Cong, "Daomap: A depth-optimal area optimization mapping algorithm for FPGA designs," in *Proc. Int. Conf. Comput.-Aided Des.*, 2004, pp. 752–759.

[16] C. A. Coello, A. D. Christiansen, and A. H. Aguirre, "Use of evolutionary techniques to automate the design of combinational circuits," *Int. J. Smart Eng. Syst. Des.*, vol. 2, no. 4, pp. 299–314, 2000.

[17] C. A. Coello, A. H. Aguirre, and B. P. Buckles, "Evolutionary multiobjective design of combinational logic circuits," in *Proc. 2nd NASA/DoD Workshop Evolvable Hardware*, 2000, pp. 161–170.

[18] C. A. Coello, R. L. Zavala, B. M. García, and A. H. Aguirre, J. Miller, A. Thompson, P. Thomson, and T. C. Fogarty, Eds., "Ant colony system for the design of combinational logic circuits," in *Proc. 3rd Int. Conf. Evolvable Syst.: From Biology to Hardware*, 2000, vol. 1801, pp. 21–30.

[19] C. A. Coello, E. H. Luna, and A. H. Aguirre, A. M. Tyrrell, P. C. Haddow, and J. Torresen, Eds., "Use of particle swarm optimization to design combinational logic circuits," in *Proc. 5th Evolvable Syst.: From Biology to Hardware*, 2003, vol. 2606, pp. 398–409.

[20] C. A. Coello, E. A. Gabriel-Luque, and A. H. Aguirre, "Comparing different serial and parallel heuristics to design combinational logic circuits," in *Proc. NASA/DoD Conf. Evolvable Hardware*, 2003, pp. 3–12.

[21] J. Cong and Y. Ding, "Beyond the combinatorial limit in depth minimization for LUT-based FPGA designs," in *Proc. Int. Conf. Comput.-Aided Des.*, 1993, pp. 110–114.
[22] ——, "Flowmap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 13, no. 1, pp. 1–12, Jan. 1994.
[23] ——, "On area/depth trade-off in LUT-based FPGA technology mapping," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 2, no. 2, pp. 137–148, Jun. 1994.
[24] ——, "Combinational logic synthesis for LUT based field programmable gate arrays," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 1, no. 2, pp. 145–204, Apr. 1996.
[25] J. Cong and Y. Y. Hwang, "Simultaneous depth and area minimization in LUT-based FPGA mapping," in *Proc. ACM/SIGDA Int. Symp. FPGA*, 1995, pp. 68–74.
[26] ——, "Structural gate decomposition for depth-optimal technology mapping in LUT-based FPGA design," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 5, no. 2, pp. 193–225, Apr. 2000.
[27] T. R. Dastidar, P. P. Chakrabarti, and P. Ray, "A synthesis system for analog circuits based on evolutionary search and topological reuse," *IEEE Trans. Evol. Comput.*, vol. 9, no. 2, pp. 211–224, Apr. 2005.
[28] A. H. Farrachi and M. Sarrafzadeh, "Complexity of the lookup-table minimization problem for FPGA technology mapping," *IEEE Tran. Comput.-Aided Des. Integr. Circuits*, vol. 13, no. 11, pp. 1319–1332, Nov. 1994.
[29] R. J. Francis, J. Rose, and K. Chung, "Chortle: A technology mapping program for lookup table-based field programmable gate arrays," in *Proc. 27th ACM/IEEE Des. Autom. Conf.*, 1990, pp. 613–619.
[30] R. J. Francis, J. Rose, and Z. Vranesic, "Chortle-CRF: Fast technology mapping for lookup table-based FPGAs," in *Proc. 28th ACM/IEEE Des. Autom. Conf.*, 1991, pp. 227–233.
[31] ——, "Technology mapping of lookup table-based FPGAs for performance," in *Proc. IEEE Int. Conf. Comput.-Aided Des.*, 1991, pp. 568–571.
[32] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Reading, MA: Addison-Wesley, 1989.
[33] D. Green, *Modern Logic Design*. Reading, MA: Addison-Wesley, 1986.
[34] H. Hemmi, T. Hikage, and K. Shimohara, "Adam: A hardware evolutionary system," in *Proc. 1997 IEEE Int. Conf. Evol. Comput.*, 1997, pp. 193–196.
[35] M. I. Heywood and A. N. Zincir-Heywood, "Dynamic page-based crossover in linear genetic programming," *IEEE Trans. Syst., Man, Cybern.—Part B*, vol. 32, no. 3, pp. 380–388, Jun. 2002.
[36] T. Higuchi, T. Niwa, T. Tanaka, H. Iba, H. de Garis, and T. Furuya, "Evolving hardware with genetic learning: A first step towards building a Darwin machine," in *Proc. 2nd Int. Conf. on Simulation of Adaptive Behavior: From Animals to Animats II*, 1993, pp. 417–424.
[37] T. Higuchi, H. Iba, and B. Manderick, "Evolvable hardware," in *Massively Parallel Artificial Intelligence*, H. Kitano and J. A. Hendler, Eds. Cambridge, MA: MIT Press, 1994, pp. 399–421.
[38] T. Higuchi, M. Murakawa, M. Iwata, I. Kajitani, W. Liu, and M. Salami, "Evolvable hardware at function level," in *Proc. IEEE Int. Conf. Evol. Comput.*, 1997, pp. 187–192.
[39] T. Higuchi, M. Iwata, D. Keymeulen, H. Sakanashi, M. Murakawa, I. Kajitani, E. Takahashi, K. Toda, M. Salami, N. Kajihara, and N. Otsu, "Real-world applications of analog and digital evolvable hardware," *IEEE Trans. Evol. Comput.*, vol. 3, no. 3, pp. 220–235, Sep. 1999.
[40] J. H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis With Applications to Biology, Control, and Artificial Intelligence*. Ann Arbor, MI: Univ. Michigan Press, 1975.
[41] J. H. Hong and S. B. Cho, "Meh: Modular evolvable hardware for designing complex circuits," in *Proc. Congr. Evol. Comput.*, 2003, pp. 92–99.
[42] B. Hounsell and T. Arslan, D. Whitley, D. Goldberg, E. Cantú-Paz, L. Spector, I. Parmee, and H. G. Beyer, Eds., "A novel evolvable hardware framework for the evolution of high performance digital circuits," in *Proc. 2nd Conf. Genetic Evol. Comput.*, 2000, pp. 525–532.
[43] H. Iba, M. Iwata, and T. Higuchi, "Gate-level evolvable hardware: Empirical study and application," in *Evolutionary Algorithms in Engineering Applications*, D. Dasgupta and Z. Michalewicz, Eds. Berlin, Germany: Springer-Verlag, 1997, pp. 259–276.
[44] I. Kajitani, T. Hoshino, M. Iwata, and T. Higuchi, "Variable length chromosome GA for evolvable hardware," in *Proc. IEEE Int. Symp. Circuits Syst.*, 1996, pp. 443–447.
[45] I. Kajitani, T. Hoshino, D. Nishikawa, H. Yokoi, S. Nakaya, T. Yamauchi, T. Inuo, N. Kajihara, M. Iwata, D. Keymeulen, and T. Higuchi, M. Sipper, D. Mange, and A. Pérez-Uribe, Eds., "A gate-level EHW chip: Implementing GA operations and reconfigurable hardware on single LSI," in *Proc. 2nd Int. Conf. Evolvable Syst.: From Biology to Hardware*, Berlin, Germany, 1998, vol. 1478, pp. 1–12.
[46] I. Kajitani, M. Murakawa, D. Nishikawa, H. Yokoi, and N. Kajihara, "An evolvable hardware chip for prosthetic hand controller," in *Proc. 7th Int. Conf. Microelectron. Neural, Fuzzy, Bio-Inspired Syst.*, 1999, pp. 179–186.
[47] T. Kalganova and J. F. Miller, "Evolving more efficient digital circuits by allowing circuit layout evolution and multi-objective fitness," in *Proc. 1st NASA/DoD Workshop Evolvable Hardware*, 1999, pp. 54–63.
[48] T. Kalganova, R. Poli, W. Banzhaf, W. B. Langdon, J. Miller, P. Nordin, and T. C. Fogarty, Eds., "An extrinsic function-level evolvable hardware approach," in *Proc. 3rd Eur. Conf. Genetic Program.*, 2000, vol. 1802, pp. 60–75.
[49] M. Karnaugh, "A map method for synthesis of combinational logic circuits," *AIEE Trans. Commun. Electron.*, vol. 72, no. I, pp. 593–599, 1953.
[50] K. Karplus, "Xmap: A technology mapper for table-lookup field-programmable gate arrays," in *Proc. 28th ACM/IEEE Design Autom. Conf.*, 1991, pp. 240–243.
[51] S. Kirkpatrick, C. Gellatt, and M. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, pp. 671–680, 1983.
[52] J. R. Koza, *Genetic Programming: on the Programming of Computers by Means of Natural Selection*. Cambridge, MA: MIT Press, 1992.
[53] J. R. Koza, F. H. Bennett, III, D. Andre, and M. A. Keane, *Genetic Programming III: Darwinian Invention and Problem Solving*. San Mateo, CA: Morgan Kaufmann, 1999.
[54] V. Krishnan and S. Katkoori, "A genetic algorithm for the design space exploration of datapaths during high-level synthesis," *IEEE Trans. Evol. Comput.*, vol. 10, no. 3, pp. 213–229, Jun. 2006.
[55] W. S. Lau, K. H. Lee, K. S. Leung, and S. M. Cheang, M. Keijzer, A. Tettamanzi, P. Collet, J. van Hemert, and M. Tomazzini, Eds., "Multi-logic-unit processor: A combinational logic circuit evaluation engine for genetic parallel programming," in *Proc. 8th Eur. Conf. Genetic Program.*, 2005, vol. 3447, pp. 167–177.
[56] K. S. Leung, K. H. Lee, and S. M. Cheang, Y. Liu, K. Tanaka, M. Iwata, T. Higuchi, and M. Yasunaga, Eds., "Balancing samples' contributions on GA learning," in *Proc. 4th Evolvable Syst.: From Biology to Hardware*, 2001, vol. 2210, pp. 256–266.
[57] ——, "Evolving parallel machine programs for a multi-ALU processor," in *Proc. Congr. Evol. Comput.*, 2002, pp. 1703–1708.
[58] ——, "Genetic parallel programming—evolving linear machine codes on a multiple ALU processor," in *Proc. Int. Conf. Artif. Intell. Eng. Technol.*, Sabah, Malaysia, 2002, pp. 207–213.
[59] ——, "Parallel programs are more evolvable than sequential programs," in *Proc. 6th Eur. Conf. Genetic Program.*, vol. 2610, pp. 107–118.
[60] S. J. Louis, "Genetic learning for combinational logic design," *J. Soft Comput.*, vol. 9, no. 1, pp. 38–43, 2004.
[61] S. W. Mahfoud, R. Männer and B. Merick, Eds., "Crowding and preselection revisited," in *Proc. 2nd Int. Conf. Parallel Problem Solving From Nature*, Amsterdam, The Netherlands, 1992, pp. 27–36.
[62] J. F. Miller, P. Thomson, and T. Fogarty, "Designing electronic circuits using evolutionary algorithms. Arithmetic circuits: A case study," in *Genetic Algorithms and Evolution Strategies in Engineering and Computer Science*, D. Quagliarella, Ed. Chechester, U.K.: Wiley, 1997, pp. 105–131.
[63] J. F. Miller, W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, Eds., "An empirical study of the efficiency of learning Boolean functions using a Cartesian genetic programming approach," in *Proc. 1st Conf. Genetic Evol. Comput.*, 1999, pp. 1135–1142.
[64] J. F. Miller, D. Job, and V. K. Vassilev, "Principles in the evolutionary design of digital circuits—Part I," *Genetic Programming and Evolvable Machines*, vol. 1, no. 1, pp. 7–35, 2000.
[65] E. J. McCluskey, "Minimization of Boolean functions," *Bell Syst. Tech. J.*, vol. 35, no. 5, pp. 1417–1444, 1956.
[66] M. Murakawa, S. Yoshizawa, I. Kajitani, T. Furuya, M. Iwata, and T. Higuchi, H. M. Voigt, W. Ebeling, I. Rechenberg, and H. P. Schwefel, Eds., "Hardware evolution at function level," in *Proc. 4th Int. Conf. Parallel Problem Solving From Nature*, 1996, vol. 1141, pp. 62–71.
[67] A. Nicholson, D. Whitley, D. Goldberg, E. Cantú-Paz, L. Spector, I. Parmee, and H. G. Beyer, Eds., "Evolution and learning for digital circuit design," in *Proc. 2nd Conf. Genetic Evol. Comput.*, 2000, pp. 519–524.

[68] R. Murgai, N. Shenoy, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Improved logic synthesis algorithms for table look up architectures," in *Proc. IEEE Int. Conf. Comput.-Aided Des.*, 1991 , pp. 564–567.
[69] M. Oltean and C. Grosan, "A comparison of several linear genetic programming techniques," *Complex Syst.*, vol. 14, no. 4, pp. 285–313, 2004.
[70] R. Poli, "Parallel distributed genetic programming," in *New Ideas in Optimization*, D. Come, M. Dorigo, and F. Glover, Eds. New York: McGraw-Hill, 1999.
[71] W. V. Quine, "A way to simplify truth functions," *Amer. Math. Monthly*, vol. 62, no. 9, pp. 627–631, 1955.
[72] C. Ryan, *Automatic Re-Engineering of Software Using Genetic Programming*. Norwell, MA: Kluwer, 2000.
[73] P. Sawkar and D. Thomas, "Area and delay mapping for table-look-up based field programmable gate arrays," in *Proc. 29th ACM/IEEE Des. Autom. Conf.*, 1992, pp. 368–373.
[74] M. Sipper, E. Sanchez, D. Mange, M. Tomassini, A. Pérez-Uribe, and A. Stauffer, "A phylogenetic, ontogenetic, and epigenetic view of bio-inspired hardware systems," *IEEE Trans. Evol. Comput.*, vol. 1, no. 1, pp. 83–97, Apr. 1997.
[75] A. Thompson, T. Higuchi, M. Iwata, and W. Liu, Eds., "An evolved circuit, intrinsic in silicon, entwined with physics," in *Proc. 1st Int. Conf. Evolvable Syst.: From Biology to Hardware*, 1996, vol. 1259, pp. 390–405.
[76] J. Torresen, M. Sipper, D. Mange, and A. Pérez-Uribe, Eds., "A divide-and-conquer approach to evolvable hardware," in *Proc. 2nd Int. Conf. Evolvable Syst.: From Biology to Hardware*, 1998, vol. 1478, pp. 57–65.
[77] ——, "A scalable approach to evolvable hardware," *Genetic Program. Evolvable Mach.*, vol. 3, no. 3, pp. 259–282, 2002.
[78] ——, "Evolving multiplier circuits by training set and training vector partitioning," in *Proc. 5th Evolvable Syst.: From Biology to Hardware*, 2003, vol. 2606, pp. 228–237.
[79] UCLA, "RASP FPGA/CPLD technology mapping & synthesis package." [Online]. Available: http://ballade.cs.ucla.edu/software_release/rasp/htdocs/
[80] V. K. Vassilev, D. Job, and J. F. Miller, "Towards the automatic design of more efficient digital circuits," in *Proc. 2nd NASA/DoD Workshop Evolvable Hardware*, 2000, pp. 151–160.
[81] *Virtex™ II Platform FPGAs: Introduction and Overview*. San Jose, CA: Xilinx, Inc., 2003.
[82] N. S. Woo, "A heuristic method for FPGA technology mapping based on the edge visibility," in *Proc. 28th ACM/IEEE Des. Autom. Conf.*, 1991, pp. 248–251.
[83] J. R. Woodward, "GA or GP? that is not the question," in *Proc. Congr. Evol. Comput.*, 2003, pp. 1056–1063.
[84] X. Yao and T. Higuchi, "Promises and challenges of evolvable hardware," *IEEE Trans. Syst., Man, Cybern.—Part C: Appl. Rev.*, vol. 29, no. 1, pp. 87–97, Feb. 1999.
[85] T. Yu and J. F. Miller, J. Miller, M. Tomassini, P. L. Lanzi, C. Ryan, A. G. B. Tettamanzi, and W. B. Langdon, Eds., "Neutrality and evolvability of Boolean function landscape," in *Proc. 4th Eur. Conf. Genetic Program.*, 2001, vol. 2038, pp. 204–217.

Sin Man Cheang (M'98–SM'04) received the B.Sc. and M.Phil. degrees in computer science and the Ph.D. degree in computer science and engineering from the Chinese University of Hong Kong, Shatin, in 1987, 1990, and 2005, respectively.

He is a Lecturer in the Department of Computing, Hong Kong Institute of Vocational Education (Kwai Chung Campus), the Vocational Training Council, Hong Kong SAR, China. He joined the department in 1997. His research interests are in genetic algorithms, genetic programming, evolvable hardware, and data mining.

Dr. Cheang is a Chartered Member of the British Computer Society (BCS), a member of the Hong Kong Institution of Engineers (HKIE), and a chartered engineer.

Kin Hong Lee (SM'90) received the degrees in computer science from the University of Manchester, Manchester, U.K.

Before he joined The Chinese University of Hong Kong in 1984, he had worked for Burroughs Machines, Scotland, and ICL, Manchester. He is now an Associate Professor of the Department of Computer Science and Engineering, Chinese University of Hong Kong. His current interests are computer hardware and bio-informatics. He has published over 60 papers in these two fields.

Kwong Sak Leung (M'77–SM'89) received the B.Sc. (Eng.) and Ph.D. degrees from the Queen Mary College, University of London, London, U.K., in 1977 and 1980, respectively.

He worked as a Senior Engineer on contract R&D at ERA Technology and later joined the Central Electricity Generating Board to work on nuclear power station simulators in England. He joined the Department of Computer Science and Engineering, Chinese University of Hong Kong, in 1985, where he is currently Professor of Computer Science and Engineering. His research interests are in soft computing including evolutionary computation, parallel computation, probabilistic search, information fusion and data mining, fuzzy data, and knowledge engineering. He has been chair and member of many program and organizing committees of international conferences. He has authored and coauthored over 200 papers and 2 books in fuzzy logic and evolutionary computation.

Dr. Leung is a Chartered Engineer, a member of IET and the Association for Computing Machinery (ACM), a Fellow of the Hong Kong Institution of Engineers (HKIE), and a distinguished fellow of the Hong Kong Computer Society (HKCS), Hong Kong. He is on the Editorial Board of *Fuzzy Sets and Systems* and an Associate Editor of the *International Journal of Intelligent Automation and Soft Computing*.