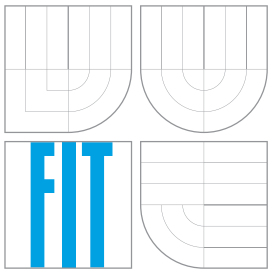
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

EVOLUČNÍ PŘÍSTUP K SYNTÉZE A OPTIMALIZACI BĚŽNÝCH A POLYMORFNÍCH OBVODŮ

EVOLUTIONARY APPROACH TO SYNTHESIS AND OPTIMIZATION OF ORDINARY AND POLYMORPHIC CIRCUITS

DISERTAČNÍ PRÁCE
PHD THESIS

AUTOR PRÁCE — Ing. ZBYŠEK GAJDA
AUTHOR

VEDOUCÍ PRÁCE — doc. Ing. LUKÁŠ SEKANINA, Ph.D.
SUPERVISOR

BRNO 2011

Abstract

This thesis deals with the evolutionary design and optimization of ordinary and polymorphic circuits. New extensions of Cartesian Genetic Programming (CGP) that allow reducing of the computational time and obtaining more compact circuits are proposed and evaluated. Second part of the thesis is focused on new methods for synthesis of polymorphic circuits. Proposed methods, based on polymorphic binary decision diagrams and polymorphic multiplexing, extend the ordinary circuit representations with the aim of including polymorphic gates. In order to reduce the number of gates in circuits synthesized using proposed methods, an evolutionary optimization based on CGP is implemented and evaluated. The implementations of polymorphic circuits optimized by CGP represent the best known solutions if the number of gates is considered as the target criterion.

Keywords

Polymorphic gate, polymorphic circuit, digital circuit design, evolutionary design, evolutionary optimization, Cartesian Genetic Programming.

Bibliographic Citation

Zbyšek Gajda: Evolutionary Approach to Synthesis and Optimization of Ordinary and Polymorphic Circuits, PhD thesis, Brno, FIT BUT, 2011

Abstrakt

Tato disertační práce se zabývá evolučním návrhem a optimalizací jak běžných, tak polymorfních digitálních obvodů. V práci jsou uvedena a vyhodnocena nová rozšíření kartézského genetického programování (Cartesian Genetic Programming, CGP), která umožňují zkrácení výpočetního času a získávání kompaktnějších obvodů. Další část práce se zaměřuje na nové metody syntézy polymorfních obvodů. Uvedené metody založené na polymorfních binárních rozhodovacích diagramech a polymorfním multiplexovaní rozšiřují běžné reprezentace digitálních obvodů, a to s ohledem na začlenění polymorfních hradel. Z důvodu snížení počtu hradel v obvodech syntetizovaných uvedenými metodami je provedena evoluční optimalizace založená na CGP. Implementované polymorfní obvody, které jsou optimalizovány s využitím CGP, reprezentují nejlepší známá řešení, jestliže je jako cílové kritérium brán počet hradel obvodu.

Klíčová slova

Polymorfní hradlo, polymorfní obvod, návrh digitálních obvodů, evoluční návrh, evoluční optimalizace, kartézské genetické programování.

Citace

Zbyšek Gajda: Evolutionary Approach to Synthesis and Optimization of Ordinary and Polymorphic Circuits, disertační práce, Brno, FIT VUT v Brně, 2011

Evolutionary Approach to Synthesis and Optimization of Ordinary and Polymorphic Circuits

Prohlášení

Prohlašuji, že jsem tuto disertační práci vypracoval samostatně pod vedením doc. Lukáše Sekaniny.

.....
Zbyšek Gajda
3. března 2011

Poděkování

Na tomto místě bych rád poděkoval Lukáši Sekaninovi, který byl mým školitelem během doktorského studia. Velmi si cením jeho neutuchající podpory během výzkumu a jeho neocenitelné pomoci při psaní článků i této práce.

Dále děkuji zaměstnancům Ústavu počítačových systémů za skvělé pracovní podmínky a přátelské prostředí, jmenovitě vedoucímu ústavu, Zdeňku Kotáskovi, kolegům, Michalu Bidlovi, Jiřímu Jarošovi, Martinu Strakovi, Josefu Strnadelovi, Václavu Šimkovi, Jaroslavu Škarvadovi, Richardovi Růžičkovi, a asistentce ústavu, Marii Gaďorkové.

V neposlední řadě děkuji rodičům, Marii a Janu Gajdovým, a všem dobrým duším, které mi dodávaly sílu a přinášely inspiraci.

Výsledky této práce vznikly za podpory Grantové agentury České republiky a Ministerstva školství, mládeže a tělovýchovy v rámci projektů: *Metody návrhu polymorfních číslicových obvodů*, GAČR, GA102/06/0599, 2006-2008, *Návrh a obvodová realizace zařízení pro automatické generování patentovatelných invencí*, GAČR, GA102/07/0850, 2007-2009, *Integrovaný přístup k výchově studentů DSP v oblasti paralelních a distribuovaných systémů*, GAČR, GD102/05/H050, 2005-2008 a *Výzkum informačních technologií z hlediska bezpečnosti*, CEZ MŠMT, MSM0021630528, 2007-2013.

Contents

Chapter 1

Introduction

Although the foundation of the digital circuit design dates from the 1950s, there are still many open areas to explore. New opportunities occasionally grow when new technologies are invented. Polymorphic electronics can be considered as one of them. Polymorphic electronics can give us an alternative to conventional digital electronics in some application domains. Researchers expect the utilization of this technology in connection with embodied intelligence where capabilities of sensing and logic operations can be combined in one compact structure, a polymorphic circuit. In continuing research based on graphene technology, scientists are developing components with polymorphic functionality that can be reconfigured on the fly into six different logic gates [99]. Unfortunately, conventional synthesis algorithms are not directly applicable for solving the polymorphic circuit synthesis problem which is, in fact, a more difficult case than the classic digital circuit synthesis problem.

A synthesis algorithm operates over a circuit representation. Various models have been devised to represent digital circuits in such a form which is suitable for synthesis algorithms. Boolean expressions, truth tables and Binary Decision Diagrams (BDD) among others have been utilized. The synthesis algorithms are capable of transforming the initial circuit representation (which is derived from the behavioral specification) onto a circuit representation which is suitable for subsequent circuit fabrication. Usually the goal of logic synthesis is to represent a circuit in the simplest way. The circuit representation, together with the synthesis algorithm, determines the space of possible implementations that one can obtain as a result of the synthesis process.

It remains unclear how to represent a gate-level polymorphic circuit and to define such transformations over a chosen representation which will lead to an efficient implementation of the polymorphic circuit using a given set of ordinary and polymorphic gates. A partial success was achieved using evolutionary design methods which do not pose any requirements on the representation or the set of transformations. However, because the methods are search-based, they are not scalable and only relatively small polymorphic circuits have evolved from scratch. Recall that a similar discipline, evolutionary optimization, allows us to optimize the circuits which have been created by the usage of conventional methods.

This thesis further develops the concept of evolutionary circuit design and optimization that have been known for almost 20 years. The evolutionary approach is primarily applied to the area of polymorphic circuit synthesis with the aim of proposing new synthesis algorithms which lead to more compact circuits.

1.1 Thesis Organization

This thesis is organized as follows. The basic principles of digital circuit design are introduced in Chapter 2. This chapter is mainly devoted to basic general design and optimization methods. It introduces some application-specific methods too. An introduction to the evolutionary circuit design is given in Chapter 3 together with the description of Evolvable Hardware concepts and some issues related to evolutionary circuit design. The state of the art of polymorphic electronics area is summarized in Chapter 4. Chapters 2, 3, 4 represent prerequisites for understanding the remaining chapters. The goals of the thesis are formulated in Chapter 5. Subsequent chapters represent the main contributions of the thesis.

Several extensions for the evolutionary algorithm, especially standard Cartesian Genetic Programming (CGP), are proposed in Chapter 6. We divide the evaluation process into the design phase and optimization phase that allows for a more accurate experimental setup to be applied. We modify the standard fitness function to support an optimization with priorities. We also propose a new selection strategy. For speeding up of the complex-problem evaluation, we introduce the short-circuit evaluation and the training-set reordering to the parallel simulation. The proposed extensions are utilized in the following experiments.

The optimization of ordinary digital circuits is performed in Chapter 7. For evaluation of the results of optimization, we utilize various benchmarks including the multipliers, majority circuits and the LGSynth91 benchmark set. The optimization is performed using the extended CGP and its results are summarized in Section 7.4.

The gate-level evolutionary design with the aim of reducing the number of utilized transistors is investigated in Chapter 8. The prioritized fitness function utilized in Cartesian Genetic Programming is examined on the adder and majority benchmarks. In order to reduce the number of transistors in obtained circuits, we add specific polymorphic gates to the set of available gates. Results of the experiments are summarized in Section 8.5.

Proposed methods for polymorphic-circuits design are described and evaluated in Chapter 9. In particular, the standard Cartesian Genetic Programming, Polymorphic Multiplexing and Polymorphic Binary Decision Diagrams are utilized in the design. We propose a set of benchmarks to compare the algorithms for polymorphic circuit synthesis.

Chapter 10 subsequently deals with the evolutionary optimization of polymorphic circuits conducted by the extended CGP. The best-obtained polymorphic circuits are summarized in Section 10.4

Conclusions, the main contributions of this thesis and future work are given in Chapter 11.

Chapter 2

Overview of Digital Circuit Design

The understanding of terms, which are common in digital circuit design, is important for one's orientation in the following chapters. So, let us briefly introduce some principles, structures and methods which are widely used in digital circuit design. Note that the following overview is mostly based on [109].

2.1 Digital Circuits – Principles

Digital logic encapsulates the analog world by mapping an infinite set of real values for a physical quality into two subsets which correspond to just two possible numbers or logic values, 0 and 1. As a result, digital logic circuits can be analyzed and designed functionally, using Boolean algebra, truth tables and other abstract means, to describe the operation by binary digits or bits (0s and 1s) in a circuit.

A logic circuit, whose output values depend only on its current input values, is called a *combinational circuit*. Its operation is fully described by a *truth table* that lists all combinations of input values and the output value(s) produced by each of them. A circuit with memory, whose outputs depend on the current input and the sequence of past inputs, is called a *sequential circuit*. The behavior of such a circuit may be described by a *state table* that specifies its output and the next state as functions of its current state and input.

The most basic digital combinational circuits are called *gates*. In general, a gate has one or more inputs and produces an output which is a function of the current input value(s). While the inputs and outputs may be analog effects such as voltage, they are modeled as taking on just two discrete logic values, 0 and 1. The most common implementations of gates are the AND, OR, NOT, NOR, NAND and XOR gates.

A combinational circuit may contain an arbitrary number of logic gates but no feedback loops. A *feedback loop* is a signal path of a circuit that allows the output of a gate to propagate back to the input of that same gate; such a loop generally creates sequential circuit behavior.

In a real design problem, we usually start out with an informal description of the circuit. Often the most challenging and creative part of design is to formalize the circuit description. A *formal circuit description* defines the circuit's input and output signals and specifies its functional behavior by means of the truth tables and logic expressions. Once we have created the formal description, we can usually follow a *synthesis procedure* to obtain a *logic diagram* for a circuit with required functional behavior.

The synthesis procedures are mostly introduced for a one-output circuit description,

but combinational circuits may have one or more outputs. Most synthesis techniques can be extended in an obvious way from a single-output to multiple-output circuit synthesis procedure, for instance, repeating the steps for each output of a circuit description.

2.2 Standard Representations of Logic Functions

Five popular representations for a combinational logic function are suggested [109]: a truth table; an algebraic sum of minterms (a canonical sum); a minterm list using $\sum$ notation; an algebraic product of maxterms (a canonical product); and a maxterm list using $\prod$ notation. Each one of these representations specifies exactly the same information. For the purpose of the thesis, we introduce the truth table representation and canonical sum.

Truth table

The most common representation of a logic function is the *truth table*. This representation simply lists the output of the circuit for every possible input combination. Traditionally, the input combinations are arranged in rows in ascending binary counting order, and the corresponding output values are written in a column that is next to the rows.

Canonical sum

The logic function can also be expressed algebraically. In order to do so, we first need to give some definitions: a *literal* is a variable or the complement of a variable; a *product term* is a single literal or a logical product of two or more literals; a *sum-of-product expression* is a logical sum of product terms; and a *minterm* is a product term in which no variable appears more than once.

There is a close correspondence between the truth table and minterms. A minterm can be defined as a product term that is logic 1 in exactly one row of the truth table. Based on the correspondence between the truth table and minterms, we can easily create an algebraic representation of a logic function from its truth table. The *canonical sum* of a logic function is a sum of the minterms corresponding to truth-table rows (input combinations) for which the function produces logic 1 at the output. The term *sum of products* (SOP) is widely used for the canonical sum.

2.3 Combinational-Circuit Synthesis

We can translate any logic expression into an equivalent sum-of-product expression, by "multiplying it out". Such an expression may be realized directly with the AND and OR gates. The inverters are also required for complement inputs [109]. The realized circuit is known as a two-level *AND-OR circuit*.

We may insert a pair of inverters between each AND-gate output and corresponding OR-gate input in a two-level AND-OR circuit. These inverters have no effect on the output function of the circuit. If these inverters are absorbed into AND and OR gates, we get AND-NOT gates at the first level and a NOT-OR gate at the second level. These are just two different symbols for the same type of gate, a NAND gate. Thus, a two-level AND-OR circuit can be transformed into a two-level *NAND-NAND circuit* by substituting some gates [109]. This simple transformation can save a considerable amount of transistors on a chip.

One should note that similarly, the canonical product can be realized by an OR-AND circuit and also this can be transformed into a NOR-NOR circuit.

It is often uneconomical to realize a logic circuit directly from the first logic expression. Canonical sum expressions are especially expensive because the number of possible minterms grows exponentially with the number of variables. Thus, we usually *minimize* a combinational circuit by reducing its number and size of gates that are necessary to realize the required function.

2.3.1 Two-level minimization methods

The traditional minimization methods have as their starting point a truth table or, equivalently, a minterm list. Most of the methods are based on a generalization of the combining theorems in Boolean algebra. That is, if two product terms differ only in the complementing or not of one variable, we can combine them into a single term with one less variable. So we save one gate and the remaining gate has one fewer input. Figure 2.1 shows an example of minimization of the logic term, $z = x.y + \bar{x}.y \rightarrow z = (x + \bar{x}).y \rightarrow z = y$

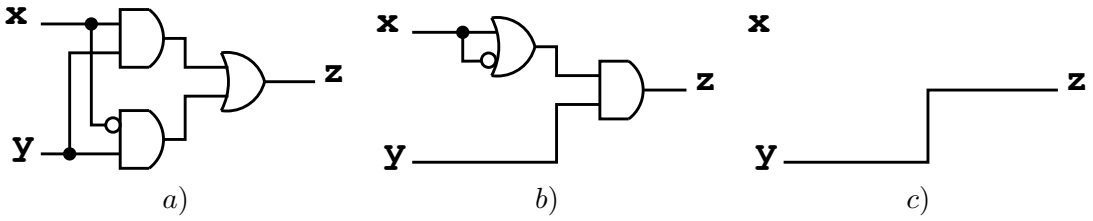


Figure 2.1: Two-level minimization. The circuits (a), (b), (c) are logically equivalent.

Karnaugh maps

A logic function can be expressed graphically into a *Karnaugh map* [36, 107] which is a method to simplify a function expression. The Karnaugh map reduces the need for extensive calculations by taking advantage of humans' pattern-recognition capability, permitting the fast identification and elimination of potential race hazards. In the Karnaugh map, variables are transferred (generally from a truth table) and ordered according to the principles of the Gray code in which only one variable changes in between squares. Once the map is generated and the output possibilities are transcribed, the data is arranged into the largest possible groups containing 2^n cells (where n is the number of inputs in an involved subexpression) and minterms are obtained by using basic Boolean operations.

Quine-McCluskey algorithm

The *Quine-McCluskey algorithm* [66, 55, 56] (also known as the method of prime implicants) is another method used for minimization of logic functions. It is functionally identical to Karnaugh mapping, but the tabular form makes it more efficient for use in computer algorithms, and it also gives a deterministic way to check that the minimal form of a logic function has been reached. It is sometimes referred to as the tabulation method. The algorithm, and also other algorithms based on this one, is characterized by an implementation of two phases which are known as a prime-implicant generation and a covering-problem solution.

Note that an implicant is a "covering" (a sum term) of one or more minterms in a sum of products of a logic function. A prime implicant of a function is an implicant that cannot be covered by a more general implicant.

Other minimization methods

The above mentioned methods may be considered as basic methods. Many researchers have discovered more effective ways to minimize combinational logic functions [109].

Some minimization problems are just too big to be solved by an "exact" algorithm. Rather than finding a provably minimal expression for a logic function, heuristic methods attempt to find a near-minimal one. Even for problems that can be solved by an "exact" method, a heuristic method may find a good solution much faster. The most successful heuristic program, *Espresso-II* [5], produces minimal or near-minimal results for a majority of problems.

Multiply-output minimization can be handled by straightforward, fairly mechanical modifications to single-output minimization methods. However, by looking at multiply-output minimization as a problem in multivalued (non-binary) logic, the designers of the *Espresso-MV* [68] algorithm were able to make substantial performance improvements over Espresso-II.

The number of inputs sometimes reaches hundreds or thousands. Thus, even advanced-minimization methods (such as Espresso) are unusable here, since their runtime is prohibitively large for such functions. *Boolean Minimizer* (BOOM) [15] is capable of handling the functions having thousands of input variables in a reasonable time.

2.3.2 Multi-level representations

Multi-level representations of logic functions have become the key data structures in electronic design because they provide a good compromise between the compactness of representation and the efficiency of manipulation.

Binary Decision Diagrams

Binary Decision Diagrams (BDDs) [13] are data structures that allow for a canonical representation and efficient manipulation of logic functions [7]. BDD is an acyclic graph which is composed of decision nodes and terminal nodes. The BDD decision nodes are labeled by input variables. The nodes decide on the basis of binary values of the variables. Terminal nodes possess possible output values which are chosen by the input values of the nodes. An example of the two-input XOR function, which is represented in BDD, is shown in Figure 2.2.

The most common variation of BDD is Ordered Binary Decision Diagram (OBDD) [7] which has input variables in an ordered form. A special variation of BDD is Multi-Terminal BDD (MTBDD). As its terminal values can be integers, it supports a multivalued logic.

Note that the BDD representation can easily be implemented by multiplexers or look-up tables[1]).

[1]Look-up table (LUT) is a programmable logic component that can implement an arbitrary logic function up to a fixed number of inputs.

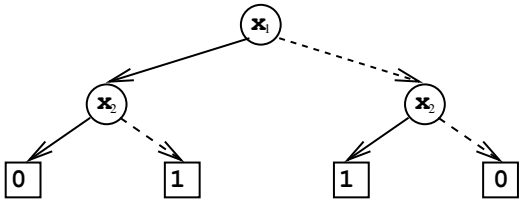


Figure 2.2: Representation of the eXclusive-OR (XOR) function in Binary Decision Diagram (BDD). Dashed arrows represent *'false'* decision (when $x_1, x_2 = 0$). Solid arrows represent *'true'* decision (when $x_1, x_2 = 1$).

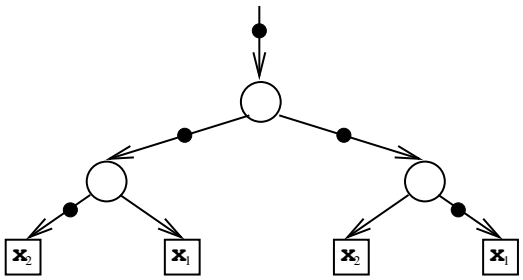


Figure 2.3: Representation of the eXclusive-OR (XOR) function in And-Invert Graph (AIG). Nodes represent the AND operators; terminal nodes represent input variables; and the marked edges represent negation.

And-invert graph

And-Invert Graph (AIG) [29, 63] is a directed, acyclic graph that represents a structural implementation of the logical functionality of a circuit or network. An AIG consists of: two-input nodes representing logical conjunction (the AND operators); terminal nodes labeled with variable names; and edges optionally containing markers indicating logical negation (see example of the XOR function in Figure 2.3). This representation of a logic function is rarely structurally efficient for large circuits, but is an efficient representation for manipulation of logic functions.

Conversion from the network of logic gates to AIGs is fast and scalable. It only requires that every gate can be expressed in terms of AND gates and inverters. AIGs allows also a reverse conversion (mapping) of them into networks composed of arbitrary gates.

2.3.3 Synthesis tools

SIS [88], MVSIS [8] and ABC System [2] are the most common open *synthesis tools* used in the synthesis-algorithm community. The ABC System, supports various representations of logic functions, such as the sum of products, Binary Decision Diagrams, but primarily And-Invert Graphs. It can perform many operations over these representations (primarily over AIGs), such as conversions, minimizations, combinational equivalence checking and synthesis or mapping. The above mentioned tools can be freely used by the research community.

2.4 Application-Specific Circuits

The existence of hierarchy and modularity together with a frequent repetition of several basic building elements are the fundamental properties of digital circuits. It is natural that efficient design methods have appeared in the field of digital-circuit design [78, 109]. The methods have been developed from principles of decomposition, minimization and re-usability.

Application-specific designs are specialized implementations for certain tasks. Their main characteristics is that they are generic structures where the structures are designed using building components, such as full adders. The building components can either be generic structures or designed by minimization methods. Another advantage of the application-specific structures is that they are capable of outperforming the minimization methods in certain cases.

Some examples of application-specific circuits are introduced in the following subsections. They will be used as benchmark circuits in the following chapters. It should be noted that optimizations presented in the thesis are mostly focused on area reduction. Hence basic implementations of the generic circuits were chosen without considering delay or other targets.

2.4.1 Combinational multiplier

A *combinational multiplier* can be designed in a basic way as the Carry-Ripple multiplier or the Carry-Save multiplier. While the Carry-Ripple multiplier is more intuitive (see Figure 2.4a), the Carry-Save multiplier is faster but the size of both is the same in terms of gates. The multipliers are composed of full-adders (FA), half-adders (HA) and the AND gates. This implementation is perfectly scalable.

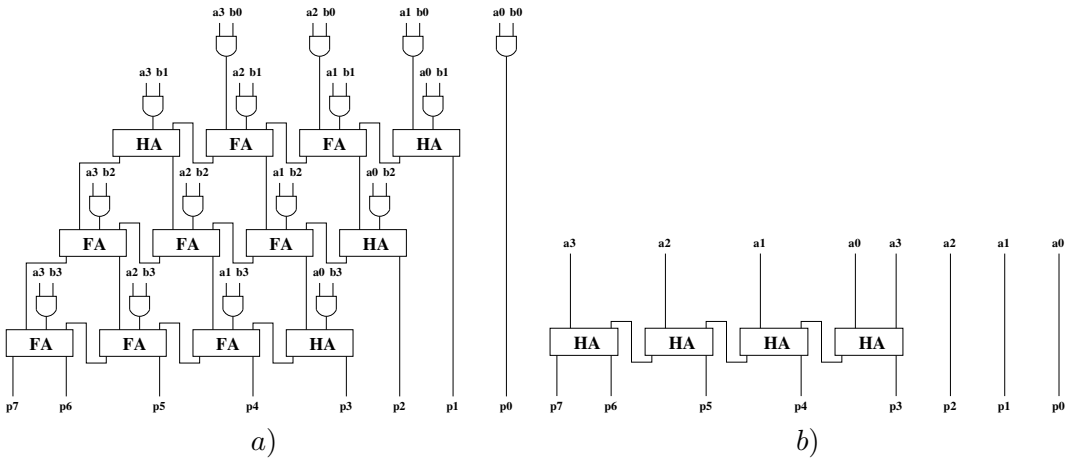


Figure 2.4: Examples of a binary multiplier: a) the 4bx4b multiplier b) a constant-coefficient multiplier which multiplies four-bit input by the 1001 binary value

Constant-coefficient multiplier

A *constant-coefficient multiplier* multiplies an input value by a single predefined constant. Multipliers of this type can be useful in the design of FIR filters [84]. The benefit of the

constant-coefficient multiplier (with respect to a universal multiplier) is that it can be implemented on a smaller area and with shorter delay. This is because it does not utilize adders and the AND gates on positions where the constant vector holds the logic 0 (see Figure 2.4b).

2.4.2 Binary sorters

Generic sorters usually implement a common sorting algorithm, such as the Bubble sort or the Merge sort. However, the input-values exchanges are performed by minimum- and maximum-decision components in a circuit implementation which is usually based on sorting-network principles [40]. *Binary sorters* sort a binary vector. The transformation of an ordinary sorter into a binary sorter is a straightforward process. Minimum (min) and maximum (max) components (see Figure 2.5a) are just converted into AND and OR gates (see Figure 2.5b). Note that the area-optimal sorters have very different schemes for the various number of inputs [120].

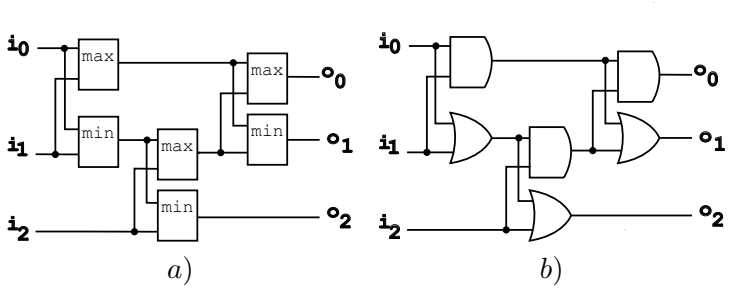


Figure 2.5: Examples of the three-input sorter: a) scheme of the ordinary sorter b) scheme of the binary sorter

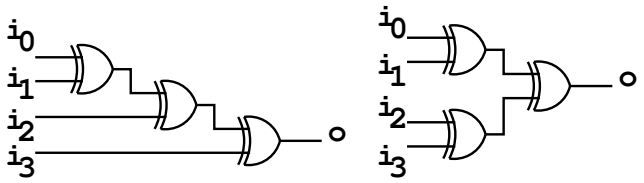
Binary majority circuits

A *binary majority circuit* detects the majority (dominance, median) of the input values. It returns logic 1 only, if more logic 1s than logic 0s are given at the circuit inputs. The majority circuit can be derived from binary sorters where the "middle" output (o_1 in Figure 2.5b) is considered only. The decide components, which have no influence on the "middle" output, can be wiped out.

2.4.3 Even parity

In some cases, it is inefficient to represent a logic function by the standard canonical form. Some problems can easily be expressed by *Reed-Muller canonical form* [24] which is usually an exclusive-or sum of products.

An *even-parity circuit* detects the even parity of the input values. The parity can be implemented in a serial way (see Figure 2.6a) or a cascade way (see Figure 2.6b) through the utilization of the XOR gates. The ways differ only in delays where the cascade implementation is faster. The area-cost is the same for both implementations.



a) *b)*

Figure 2.6: An example of the four-input even parity: a) the scheme of the serial parity b) the scheme of the cascade parity

Chapter 3

Evolutionary Design of Digital Circuits and Evolvable Hardware

The aim of this chapter is to briefly introduce the evolutionary approach to engineering design which is needed for orientation in the following chapters. In particular, evolutionary algorithms will be employed for the design and optimization of digital circuits. A brief introduction to Evolvable Hardware is also included.

3.1 Evolutionary Algorithms

Evolutionary algorithms (EAs) are stochastic search algorithms inspired by Darwin's theory of evolution [47, 78]. The *search space* is a space which contains all possible considered solutions of the problem. Instead of working with one solution at a time (as random search, hill climbing algorithm or other search methods do [57]), these algorithms work with a *population* of *candidate solutions* (individuals). Every new population is formed by using genetically inspired operators (such as *crossover* and *mutation*) and by selection pressure which guides the evolution towards better areas of the search space. The evolutionary algorithms receive guidance by evaluating every candidate solution to define its *fitness value*. The fitness value, calculated by a *fitness function*, indicates how well the solution fulfills the problem objective (specification).

For the most part, a higher fitness value implies a greater chance that an individual will "live" for a longer period of time and produce offspring who will inherit a parental genetic information. This leads to the production of a novel genetic information and to novel solutions of the problem. The fundamental structure of an evolutionary algorithm is captured in Figure 3.1 and its data-flow diagram is illustrated in Figure 3.2.

Because the objects in the search space can be arbitrary structures (e.g. real-valued vectors, circuits), it is often helpful to distinguish between a *representation space* (genotype space) and a *solution space* (phenotype space) [78]. As shown in Figure 3.3, encoded solutions (*genotypes*) have to be mapped onto actual solutions (*phenotypes*). In general, the *problem representation* consists of a representation space and the *encoding function*. The fitness function is applied to evaluate phenotypes. While the fitness function works with phenotypes, genetic operators (e.g. crossover, mutation) are defined over genotypes. There is one important rule in this concept of *genotype-phenotype mapping*: a small change in the genotype should induce a small change in the phenotype in order to obtain a reasonably efficient search algorithm.

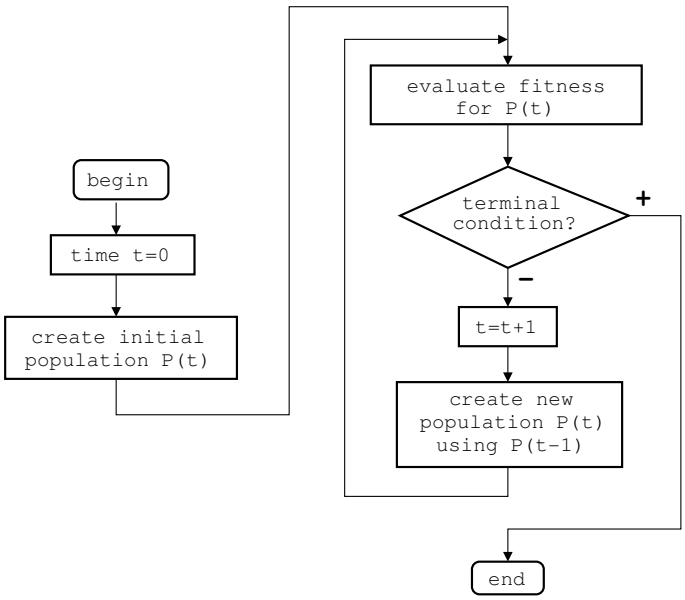


Figure 3.1: Control-flow diagram of an evolutionary algorithm.

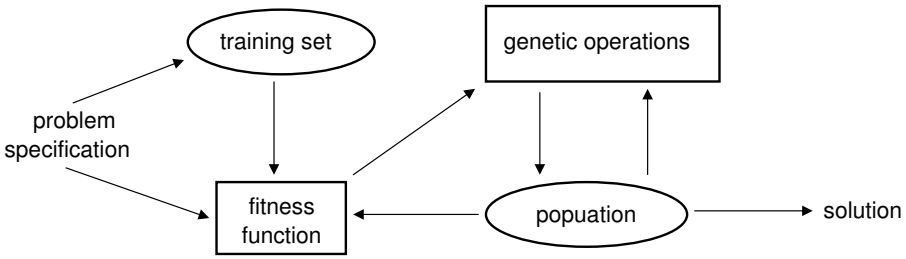


Figure 3.2: Data-flow diagram of an evolutionary algorithm.

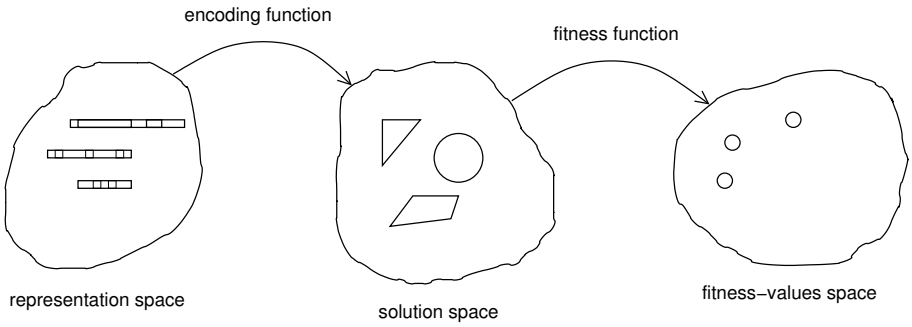


Figure 3.3: The concept of genotype-phenotype mapping.

Any evolutionary algorithm can be viewed as utilizing one or more genetic operators which produce new candidate solutions from those previously visited in the search space. Effective algorithms achieve the balancing of two conflicting goals: *exploiting* the best solution which has been found up to now; and *exploring* the search space [57]. Each search space is different and even identical spaces can appear very different under different representations and fitness functions. So there is no way to choose a single search method that can serve well in all cases [57].

Traditionally, four main variants of an evolutionary algorithm are presented [47, 78]: Genetic Algorithm, Genetic Programming, Evolution Strategy and Evolutionary Programming. For the purpose of this thesis, Genetic Programming and Evolution Strategy are relevant and need to be introduced.

Genetic Programming

Genetic Programming (GP) [42, 43, 44] has been developed by John R. Koza to allow automatic programming and program induction. GP does not distinguish between the representation space and the solution space.

Programs (genotypes) are represented either as tree structures (Tree-based GP) or in a linear form (Linear GP) using machine-language instructions [78]. A *crossover* is considered as a major genetic operator for Genetic Programming. Figure 3.4 shows that the crossover interchanges randomly chosen subtrees of parents' trees without the disruption of the syntax. A *mutation*, another genetic operator, picks a random subtree and replaces it with a randomly generated one. A *selection* is typically implemented as a probabilistic operator, using the relative fitness, which determines the selection probability of an individual. The most important application of GP is symbolic regression.

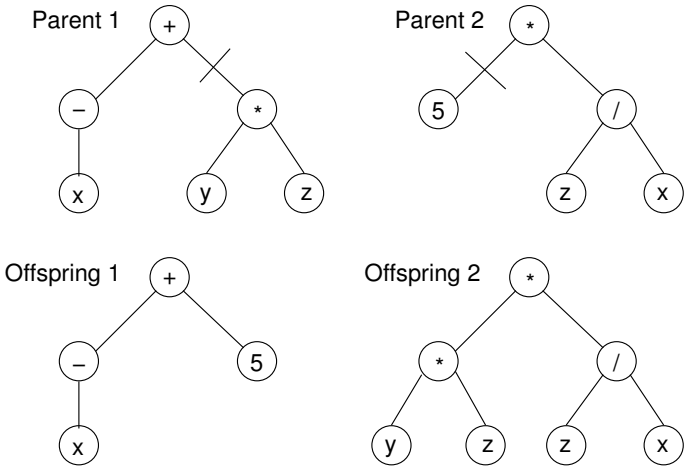


Figure 3.4: A single-point crossover in Genetic Programming

An evolved program may contain segments which do not alter the result of the program execution when they are removed from it, that is, semantically redundant segments (e.g., $i = i + 0$). Such segments are referred to as *introns* [48]. The program size can also grow uncontrollably until it reaches the tree-depth maximum, while the fitness remains unchanged. This effect is called a *bloat* [58]. Problems and benefits of introns, bloat and their relationship are discussed in [48, 58].

Evolution Strategy

Evolution Strategy (ES) [76] was developed for optimization purposes in industrial application by Bienert, Rechenberg and Schwefel in the 1960s. Like Genetic Programming, Evolution Strategy does not distinguish between a genotype and phenotype. Each individual is represented as a real-valued vector. A mutation is regarded as the primary genetic

operator. It aggregates a normal-distributed random variable and a preselected standard deviation value which are applied on every gene of a candidate vector.

Evolution Strategy usually selects the parents deterministically. The algorithm provides two major selection scenarios: $(\mu + \lambda)$-ES and (μ, λ)-ES. The $(\mu + \lambda)$-ES selects the best μ individuals from both offspring and parent populations (of size $\mu + \lambda$). If an offspring gets a better fitness value than its parent, the offspring becomes the new parent. Otherwise, the parent is mutated to create a new offspring. The (μ, λ)-ES selects the best μ individuals from only the offspring population (of size λ).

Evolutionary design and optimization

Evolutionary algorithms can be utilized in various applications. Bentley's classification [14] defines four main categories of an evolutionary-algorithm application: creative evolutionary design, evolutionary design optimization, evolutionary art and evolutionary artificial-life forms. From the perspective of the thesis, creative evolutionary design and evolutionary design optimization are relevant for the following chapters.

In the *creative evolutionary design*, the evolutionary algorithms have the ability to generate entirely new designs when they start from little or nothing and are guided purely by functional performance criteria [14]. An emphasis is placed on novelty and originality. Creative evolutionary design is able to introduce new genes (variables) in a genotype and thus to define new search spaces.

In the *evolutionary design optimization*, an optimization process requires finding a set of free parameters under consideration, so that a certain quality criterion (the fitness value) is minimized (or maximized) [12]. Designers usually start the process with an existing design and parametrize its parts that need improvement. The parameters are encoded into a genotype and then the evolutionary algorithm is involved in finding a sufficient solution. Optimization places an emphasis on finding a solution which is as close to the global optimum as possible.

3.2 Evolvable Hardware

Evolvable hardware (EHW) [25, 30, 78, 100] involves evolutionary algorithms to create specialized hardware automatically. It brings together artificial intelligence methods and reconfigurable hardware. In some cases, EHW is capable of changing a hardware architecture and behavior dynamically and autonomously.

EHW problems fall into two categories [25]: evolutionary hardware design and adaptive systems. The *evolutionary hardware design* uses evolutionary algorithms to evolve a system that meets a predefined specification. The *adaptive systems* reconfigure an existing design to counteract faults or a changed operational environment. Figure 3.5 illustrates relations between an evolutionary algorithm (Evolution Engine and Fitness unit) and reconfigurable hardware.

Two scenarios have been developed for hardware evolution: extrinsic evolution and intrinsic evolution. In *extrinsic evolution*, all candidate circuits are simulated to obtain a fitness score and only the final best solution in the final population is usually physically implemented. In *intrinsic evolution*, every individual in every generation is evaluated in real hardware. The importance of intrinsic evolution was discovered by Thompson [100].

Most reconfigurable devices consist of configurable blocks where functionality and interconnections are controlled by a configuration bitstream. Many types of reconfigurable

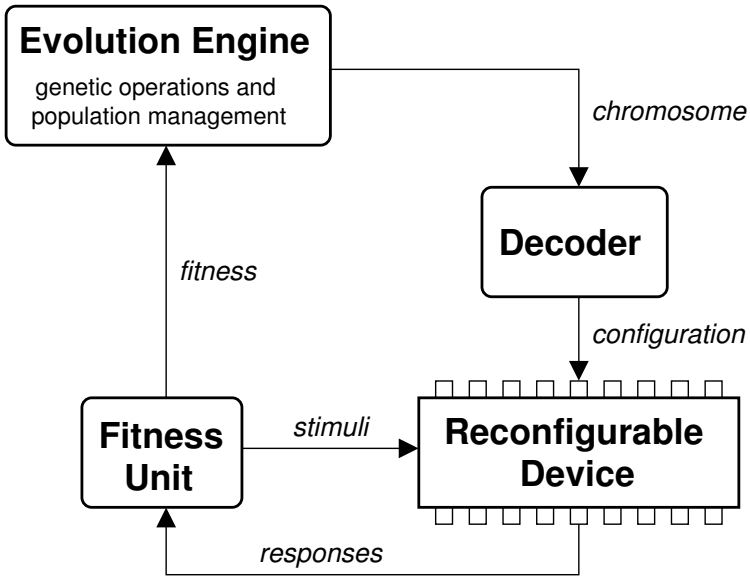


Figure 3.5: High-level description of an evolvable hardware approach

devices have been utilized in EHW. Typical reconfigurable devices are Programmable Logic Arrays (PLAs), Field-Programmable Gate Arrays (FPGAs) for digital designs, Field-Programmable Analog Arrays (FPAAs) for analog designs and Field-Programmable Transistor Arrays (FPTAs) which have been applied to implement either digital or analog designs. Special devices have also been utilized such as a Reconfigurable Polymorphic Module (REPOMO) [73] (see Section 4.2.1) or a Multi-Logic-unit Processor (MLP) [108]. More exotic reconfigurable devices include reconfigurable liquid crystals (Evolution in materio) [26], reconfigurable molecular array (NanoCell) [32], reconfigurable antenna array [51] and reconfigurable optics (deformable mirrors) [52].

Evolutionary hardware design was employed for the creation of innovative designs such as small combinational circuits [59], digital filters [31], image operators [78], classifiers [21] or diagnostic benchmark circuits [65].

Adaptive hardware was involved in several applications, for instance, in self-configurable integrated circuits [92], adaptive image compression methods [75], myoelectric prosthetic hand controller [33] or post-fabrication tuning of filters [64].

3.3 Cartesian Genetic Programming

Cartesian Genetic Programming (CGP) is a sub-domain of GP [62]. CGP is a widely-used method for evolution of programs and digital circuits [62, 59, 104, 35, 17]. In its basic version which is important for the thesis, candidate circuits are directly represented in the chromosome. Extensions of CGP have been proposed in recent years, for example, self-modifying CGP [27], modular CGP [110, 37] or multi-chromosome CGP [111].

3.3.1 Basic CGP

CGP was introduced by Miller and Thompson a decade ago [62]. It resembles the concept of Genetic Programming but introduces some important modifications: (i) a candidate circuit is modeled using a directed acyclic graph; (ii) the graph is encoded as a fixed-size string

of integers; and (iii) a search is performed using a mutation-based Evolution Strategy (no crossover is employed). The main advantage of CGP is that it generates very compact solutions, that is, it can effectively reduce the total number of gates in the case of circuit evolution [104].

In the basic version of CGP, a candidate circuit is modeled in a matrix of $n_c \times n_r$ of programmable n_a-input elements where n_c is the number of columns (the horizontal size) and n_r is the number of rows (the vertical size). The number of inputs, n_i, and outputs, n_o, of the circuit is fixed. Each gate input can be connected either to the output of a gate placed in the previous l columns or to one of the circuit inputs. The l-back (level-back) parameter, in fact, defines the level of connectivity, thus reducing/extending the search space. For example, if $l = 1$ then only neighboring columns may be connected (useful in the design of pipeline architecture); if $n_r = 1$ and $l = n_c$ then full connectivity is allowed. Since only combinational circuits will be evolved (in this thesis), no feedback loops are allowed in candidate circuits. Each gate can be programmed to perform one of n_a-input logic operations defined in the Γ gate set. Each element is encoded as a list of $n_a + 1$ (unsigned) integers where the first n_a components are elements indexes (connections of element inputs and elements outputs) and the last component is the index of a logic operation (from the Γ set). An element input can be connected to either the output of the preceding element or to a primary circuit input. Every individual is encoded as a list of elements using $n_c.n_r.(n_a + 1) + n_o$ integers, where the last n_o components are the elements indexes which represent the primary circuit outputs. As Figure 3.6 shows, while the size of chromosome is fixed, the size of phenotype is variable (i.e. some nodes are not used).

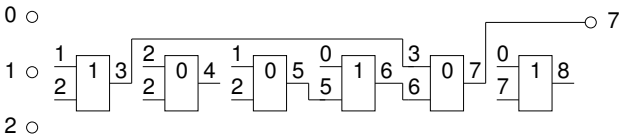


Figure 3.6: An example of a circuit in CGP with parameters: $n_i = 3$, $n_o = 1$, $l = 6$, $n_c = 6$, $n_r = 1$, $\Gamma = \{0 : AND, 1 : OR\}$. Elements 4 and 8 are not utilized. Chromosome is: 1,2,**1**, 2,2,**0**, 1,2,**0**, 0,5,**1**, 3,6,**0**, 0,7,**1**, 7. Logic operations of elements are typed in bold. The last integer indicates the output of the circuit.

3.3.2 Circuit evolution using CGP

The goal of the evolutionary circuit design, when it is applied to logic synthesis, is to obtain a perfectly working circuit (all assignments to the circuit inputs have to be tested). When the perfectly working circuit (correct circuit) is obtained, then it should be "optimized", that is, the number of utilized gates is minimized. Other optimization criteria can be included; however, this thesis mostly considers the number of gates only (except Chapter 8). In the case of the combinational circuit evolution, the fitness function to evaluate a candidate circuit is typically defined in CGP as [35]:

$$fitness = \begin{cases} b & when \quad b < n_o.2^{n_i}, \\ b + (n_c.n_r - z) & otherwise, \end{cases} \tag{3.1}$$

where b is the number of correct output bits obtained as the response for all possible assignments to the inputs; z denotes the number of gates utilized in a particular candidate circuit; and $n_c.n_r$ is the total number of available elements. It can be seen that the term

$n_c.n_r - z$ is considered only if the circuit behavior is perfect, that is, if $b = n_o.2^{n_i}$. We can observe that the evolution is forced (by the fitness function) to first discover a perfect working solution while the size of circuit is not important. After discovering the correct solution, the evolution is forced to minimize the number of gates of the correct circuit.

CGP operates with the population of $1 + \lambda$ individuals (λ is from 4 to 15 typically). The initial population is either randomly generated or created by a heuristic procedure. The initial population can be created using conventional methods or synthesis tools (such as ABC System, SIS) in the case of the heuristic procedure.

Every new population consists of the best individual of the previous population (previous generation) which acts as the parental individual. The offspring individuals are created by a point mutation operator which modifies h randomly selected genes of the parental individual where h is a user-defined value. The implementation of the mutation operator has to ensure that the modification of gene is legal. The algorithm is usually terminated when the maximum number of generations is reached or a sufficient working solution is obtained.

There is an important rule for the selection of the parental individual. In the case when two or more individuals are able to become the parent, the individual which has not been a parent in the previous population will be selected as the new parent. In the case when there are more candidates then the parent is chosen randomly. This strategy is important because it ensures diversity in the population. The strategy has been proven to be very useful in [59, 106].

CGP has been successfully utilized in many applications [59, 17, 77, 41, 39]. It has been investigated experimentally [61] and extended to support modularity, self-modification and other features [110, 28]. In particular, it was shown that multipliers evolved by CGP [104] are smaller than the best known designs of that time (when the number of 2-inputs gates is the decision criterion).

Redundancy and neutrality

The encoding used in CGP is redundant since there may be genes that are entirely inactive. These genes do not influence the phenotype and hence the fitness score. This phenomenon is often referred to as *neutrality*. The role of neutrality has been investigated in detail [115, 105, 61]. For example, it was discovered that the most evolvable representations occur when the genotype is extremely large and in which over 95% of the genes are inactive [61]. But in another example, Collins has shown that for some specific problems, the neutrality-based search is not the best solution [11]. Miller has also identified that the problem of bloat is insignificant for CGP [58].

Parallel simulation

Parallel simulation is the technique that can be used to accelerate the circuit evaluation [59]. The idea of the parallel simulation is to utilize bitwise operators which operate on multiple bits at a high-level language (such as the C language). The technique performs more than one evaluation of a gate in a single step. For example, when a combinational circuit under simulation has three inputs and it is possible to concurrently perform bitwise operations over $2^3 = 8$ bits in the simulator, then the circuit can be completely simulated by applying a single 8-bit test vector at each circuit input (see the encoding in Figure 3.7). In contrast, when it is impossible then the eight three-bit test vectors must be applied sequentially. Practically, current processors allow us to operate with 64 bit operands, that

means, it is possible to evaluate the truth table of a six-input circuit by applying a single 64-bit test vector at each input. Therefore, the obtained speedup is 64 against the sequential simulation. In the case that a circuit has more than 6 inputs, then the speedup is constant (i.e. 64). This technique has been applied in all evolutionary design and optimization experiments reported in the thesis.

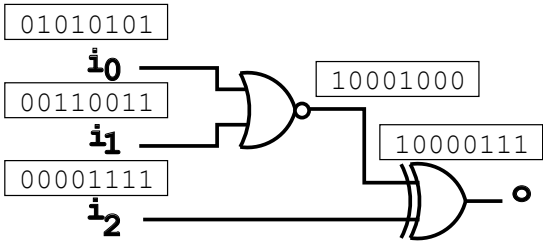


Figure 3.7: Parallel evaluation of a candidate circuit

3.4 Scalability Problem

In the context of the evolutionary circuit design, the *scalability problem* is often mentioned in the literature [106, 114, 23]. The implication of the scalability problem is that only relatively simple circuits have been evolved so far.

The main problem of the evolutionary combinational-circuit design is that the evaluation time of a candidate circuit grows exponentially with the increasing number of inputs (assuming that all possible input combinations are evaluated by the fitness function) [106]. Hence, the evaluation time becomes the main bottleneck of the evolutionary approach when complex circuits with many inputs are evolved. Another problem is that the genotype size grows with the increasing complexity of target circuits. Long chromosomes imply large search spaces that are usually difficult for EA.

We will briefly introduce some techniques that were developed to suppress the influence of the scalability problem of the evolutionary circuit design. The most common techniques are: incremental evolution, the multi-chromosome approach and development.

Incremental evolution and the multi-chromosome approach are based on the decomposition of a complex problem into sub-problems. The sub-problems can be evolved in a reasonable time. The technique uses a *training set and training vector partitioning* [102]. For solving the sub-problems, CGP can be used. The *bidirectional incremental evolution* [34] incorporates two approaches: divide-and-conquer [101] and incremental evolution [22]. Bidirectional incremental evolution is based on gradual decomposition of a complex problem into sub-problems of decreasing complexity and afterwards, on gradual evolution of sub-problems of increasing complexity into a complex problem. This approach utilizes the training set and training vector partitioning. The *multi-chromosome approach* [111] is based on a genotype which consists of several chromosomes. Each chromosome represents a solution for one output of a target circuit. The output solutions (chromosomes) are evolved simultaneously. An individual is represented by the genotype (not by a single chromosome). This approach utilizes training vector partitioning.

Computational development [3, 45, 82] is usually considered as a non-trivial and indirect mapping from genotypes to phenotypes in an evolutionary algorithm. In such a case, the genotype has to contain a prescription for construction of the target object. While the

genetic operators work with genotypes, the evaluation of candidate solutions (the fitness evaluation) is applied on phenotypes created by means of development. The principles and selected application of development are summarized in [46, 4]. Lindenmayer systems (L-systems), cellular automata, graph-generation grammars and an instruction-based development are major approaches to development. These models were applied to design digital circuits. The main problem of current development is that resulting circuits are not competitive with conventional synthesis [4].

Chapter 4

Polymorphic Electronics

Polymorphic electronics is a new unconventional sub-area of electronics. It was introduced by A. Stoica's group at the NASA Jet Propulsion Laboratory as a new class of electronic devices that exhibit a new style of (re)configuration [97]. Polymorphic electronics[1] has a superimposed built-in functionality. A change of function does not need reconfiguration/switches as in traditional approaches. The changes of functionality come from an environment, such as temperature, light, power supply voltage and control signal. Polymorphic electronics can be understood as a new reconfigurable technology capable of integrating logic functions with sensing in a single compact structure. In fact, the fundamental building components, polymorphic logic gates, merge the capability of performing logic operations with sensing. Hence polymorphic gates would also be very useful in building the *embodied intelligence*—intelligent devices, whose functionality emerges in interaction with a physical environment [6]. Although polymorphic gates can be implemented relatively effectively using current CMOS technology, we can expect an expansion of polymorphic devices with further development of nanoelectronics and molecular electronics.

Current research in the area of polymorphic electronics can be split into three fields: design of reliable polymorphic gates, development of synthesis algorithms and development of applications. We will briefly survey them in the following sections.

4.1 Polymorphic Gates

Polymorphic gates play a central role in polymorphic electronics. The *polymorphic gate* is capable of switching among two or more logic operations (functions). However, the selection of the operation is performed unconventionally. The logic operation of a polymorphic gate depends on some external factors, for example, on the level of the power supply voltage (V_{dd}), temperature, light or some other external signals [74, 93, 94, 95, 96, 97, 98, 118, 119].

A polymorphic gate implements several logic operations according to a control signal which can hold different values. For purposes of this thesis, we denote a bi-functional polymorphic gate as X_1/X_2, where X_1 is the logic operation of the first mode (*mode 1*) and X_2 of the logic operation of the second mode (*mode 2*). For example, Stoica's polymorphic bi-functional NAND/NOR gate [93] controlled by V_{dd} operates as the NOR gate for $V_{dd} =$ 1.8 V (*mode 1*) and the NAND gate for $V_{dd} = 3.3$ V (*mode 2*).

Table 4.1 surveys the polymorphic gates reported in the literature. The gates were designed mostly through an evolutionary approach [94, 98, 116]. Only three of them have

[1] Polymorphic electronics is also known as polytronics [97]

been fabricated so far; the remaining polymorphic gates were either simulated or tested in a field programmable transistor array (FPTA-2). The six-transistor NAND/NOR gate [93] controlled by the power-supply voltage, V_{dd}, was fabricated in the 0.5-micron HP technology. The ten-transistor NAND/NOR gate [95] controlled by V_{dd} was fabricated in the 0.5-micron HP technology. Another eight-transistor NAND/NOR gate [74] controlled by V_{dd} was fabricated in the 0.7-micron AMIS technology.

Table 4.1: Existing polymorphic gates. The 'Trans.' label denotes utilized transistors and the 'Ref.' label denotes references in the literature.

Gate	Control Values	Control	Trans.	Ref.
AND/OR	27/125°C	temperature	6	[97]
NAND/NOR	80/120°C	temperature	12	[96]
AND/OR/XOR	3.3/0.0/1.5 V	control signal	10	[97]
AND/OR	3.3/0.0 V	control signal	6	[97]
NAND/NOR/NXOR/AND	0/0.9/1.1/1.8 V	control signal	11	[118]
NAND/WIRE/AND	0.0/1.0/-1.8 V	control signal	9	[119]
WIRE/OR/XOR/AND2b/NAND/AND1b	0.0/0.2/0.4–0.8/1.0/1.2–1.6/1.8 V	control signal	19	[94]
NAND/NOR	0/2.5 V	control signal	10	[89]
NOR/NAND	0/2.5 V	control signal	8	[72]
NAND/XOR	0/2.5 V	control signal	9	[70]
AND/OR	1.2/3.3 V	supply voltage	8	[98]
NAND/NOR	3.3/1.8 V	supply voltage	6	[93]
NAND/NOR	3.3/1.8 V	supply voltage	10	[95]
NAND/NOR	5/3.3 V	supply voltage	8	[74]

Figure 4.1 shows one of possible designs of the polymorphic NAND/NOR gate which was published in [74]. Logic operations of this circuit are controlled by the power supply voltage. Circuit responses (the Z label) are demonstrated for all possible input combinations (the A, B labels) in Figure 4.2. The waveforms come from measurements of the gate implementation in the 0.7-micron AMIS technology.

Figure 4.3 shows a circuit symbol of a polymorphic gate. If logic operations of a gate are not obvious from the context, the logic operations are given as a part of the gate symbol. Because the circuit symbols for any polymorphic gates are not yet under any international (even national) standard, thus the symbol given in Figure 4.3 is presented for local usage in circuit schemes of this thesis.

However, some gates, which are introduced in Table 4.1, do not exhibit electronic properties as good as standard CMOS gates. For example, very high power consumption was reported for the NAND/NOR gate [74]. Due to this fact, the spectrum of applications is limited for polymorphic electronics nowadays.

4.2 Polymorphic Circuits

Having polymorphic gates, researchers have begun to develop new methods for synthesis of digital circuits that contain polymorphic gates [18, 53, 79, 87]. The main motivation is to obtain reconfigurable (and thus potentially adaptive) circuits for a very low cost and without the necessity to implement a reconfiguration infrastructure, such as switches, multiplexers or configuration registers. Figure 4.4a shows an example of a *polymorphic digital circuit* and

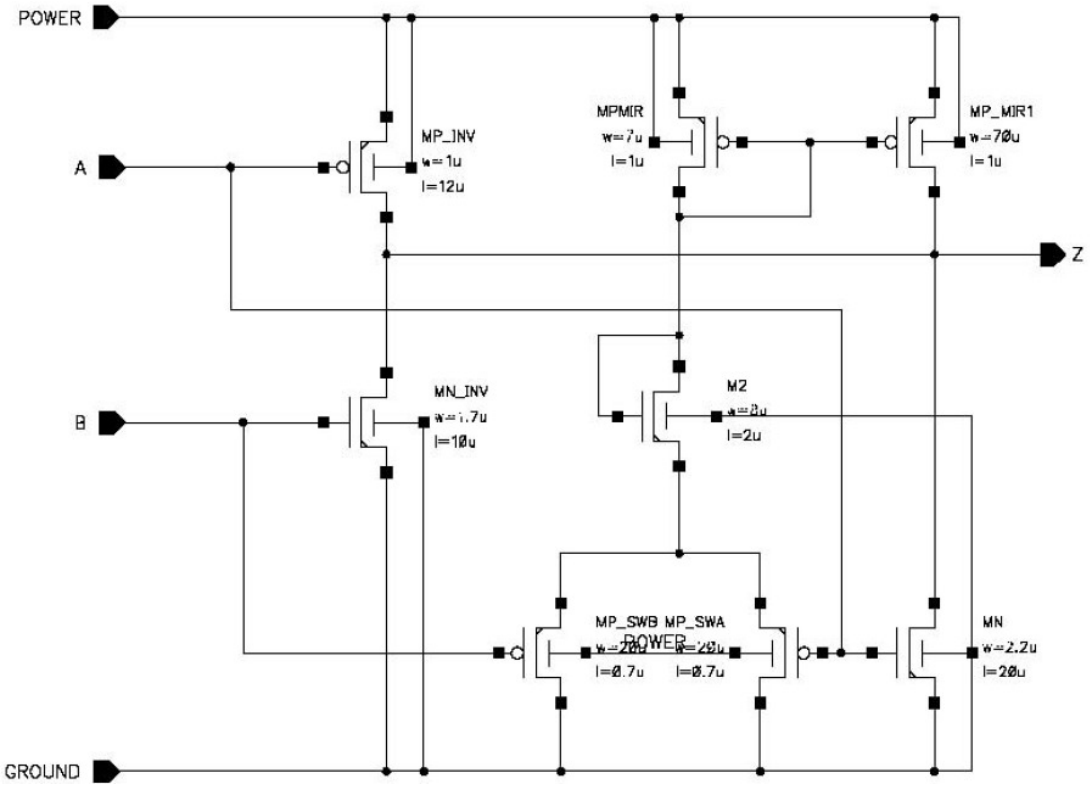


Figure 4.1: Transistor-level implementation of the NAND/NOR gate published in [74]. The gate realizes the NAND operation for V_{dd} from 3.9 to 5.0 V and the NOR operation for V_{dd} from 3.0 to 3.7 V. The A,B labels denote the inputs and the Z label denotes the output.

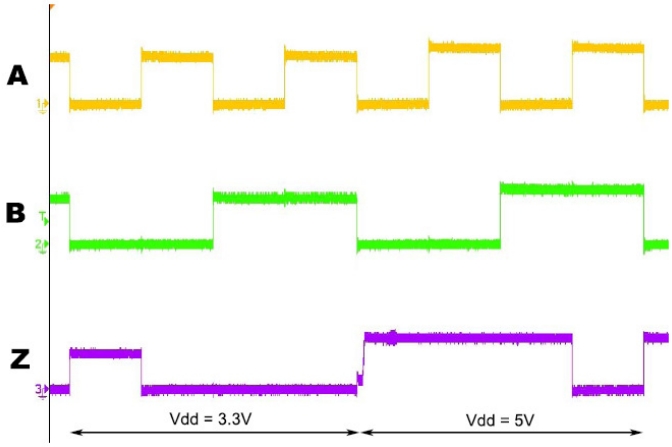


Figure 4.2: Behavior of the NAND/NOR polymorphic gate [74] measured at 5 kHz. The A, B labels denote the inputs and the Z label denotes the output.

Figures 4.4b,c show its equivalent behavior in both modes of the polymorphic NAND/NOR gate. The equivalent functions are $f_1 = \overline{i_0 \wedge i_1} \oplus i_2$ in *mode 1* and $f_2 = \overline{i_0 \vee i_1} \oplus i_2$ in *mode 2*.

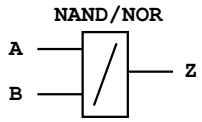


Figure 4.3: A circuit symbol of a polymorphic NAND/NOR gate.

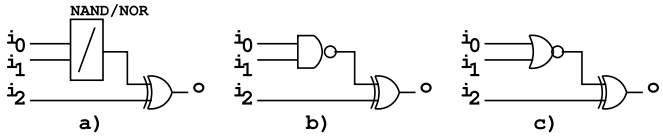


Figure 4.4: a) Example of a polymorphic circuit; b) Equivalent circuit in *mode 1*; c) Equivalent circuit in *mode 2*

4.2.1 REPOMO32 platform

A *Reconfigurable Polymorphic Module* (REPOMO) [73, 85, 103] has been developed in order to investigate electrical properties of polymorphic circuits and to demonstrate the applications of polymorphic electronics. The REPOMO32 platform contains an array of 32 configurable logic elements; each of them can perform the AND, OR, XOR and polymorphic NAND/NOR logic operation. Since the set of logic operations is complete, the platform can perform any logic function (of course, functionality is limited by the chip size). This chip utilizes the NAND/NOR gates [74] controlled by power supply voltage, V_{dd}. When $V_{dd} = 5.0\ V$ the polymorphic gate exhibits the NAND operation and when $V_{dd} = 3.3\ V$ the gate exhibits the NOR operation. The remaining ordinary (conventional) gates do not change their logic operations with a shift of V_{dd} (from 3.0 V to 5.0 V). Figure 4.5 shows the block structure of the REPOMO32 platform.

4.3 Applications of Polymorphic Electronics

Papers [97, 98] suggest various areas in which polymorphic gates can be utilized. Applications of polymorphic electronics, which were reported or proposed so far, are given as references in the following summary:

(i) Automatic change of circuit behavior when a power supply is not sufficient [84].

(ii) Implementation of low-cost reconfigurable/adaptive systems that are able to adjust their behavior inherently in response to certain control signals, such as multi-functional counters [70, 118].

(iii) Implementation of novel concepts for testing and diagnosing of electronic circuits, such as self-checking adders [54, 74] utilization or reduction of test vector volume [90].

(iv) Implementation of a hidden function (invisible for the user) which can be activated in a specific environment [97, 98].

(v) Intelligent sensors for biometrics, robotics and industrial measurement [97, 98].

(vi) Reverse engineering protection [97, 98].

In the following paragraphs we briefly introduce some applications that have been developed at FIT BUT.

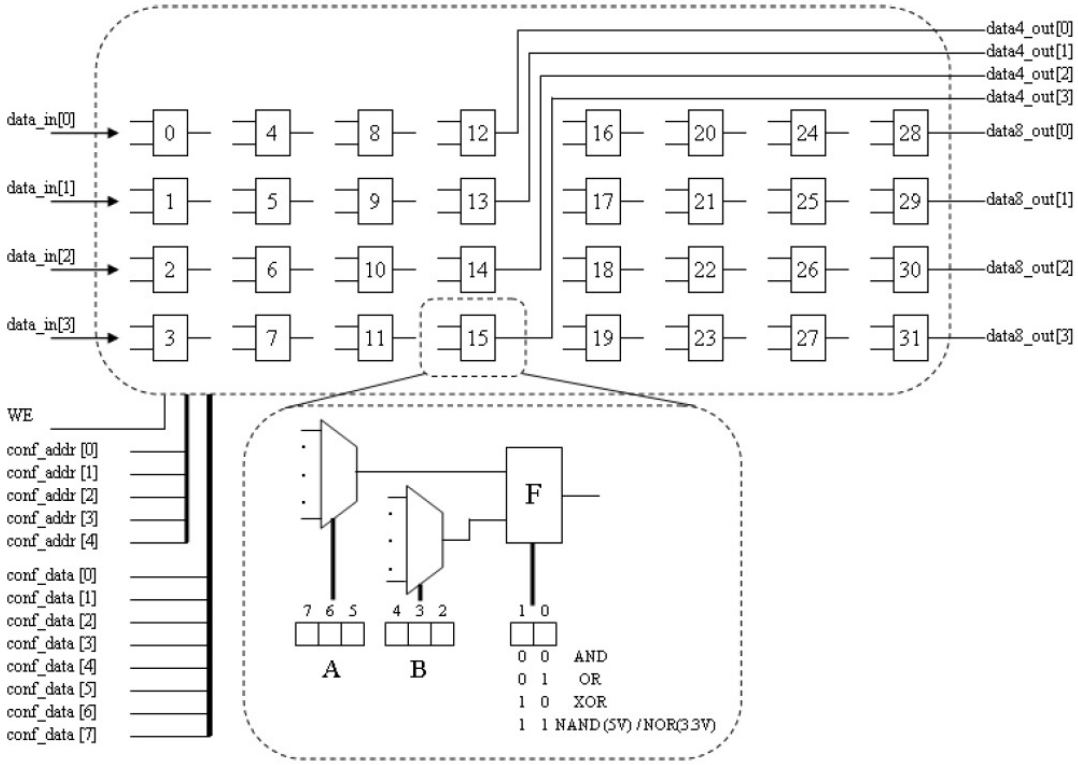


Figure 4.5: The REPOMO32 block structure published in [85].

4.3.1 Polymorphic FIR filter

The *polymorphic FIR filter* [84] with backup mode enabling power saving (see Figure 4.6) can operate in two modes. In the first mode (standard mode), the filter performs a normal function. In the second mode (backup mode), the filter operates with a reduced power supply voltage. In this mode, some filter coefficients are reconfigured and the rest of the filter is disconnected. Experiments have indicated that power consumption can significantly be reduced in the backup mode while the quality of filtering remains reasonable. One should note that the reconfigurable constant-coefficient multipliers of the FIR filter are implemented using the NAND/NOR polymorphic gates.

4.3.2 Polymorphic controller

Let us imagine that an intelligent controller is needed in some application. A feature of that controller is that it can skip less important states under some circumstances, such as low power or high temperature. An example of such a controller is the three-bit *polymorphic controller* published in [69, 70, 71] (see Figure 4.7). This controller has seven states in the normal mode and five states in the backup mode. Note that in this design, the next-state logic is implemented utilizing the NAND/XOR polymorphic gates.

4.3.3 Self-checking adder

The *polymorphic self-checking adder* [74, 80, 81, 85] does not utilize any additional signals to indicate the fault. The adder is able to detect a reasonable number of stuck-at-faults

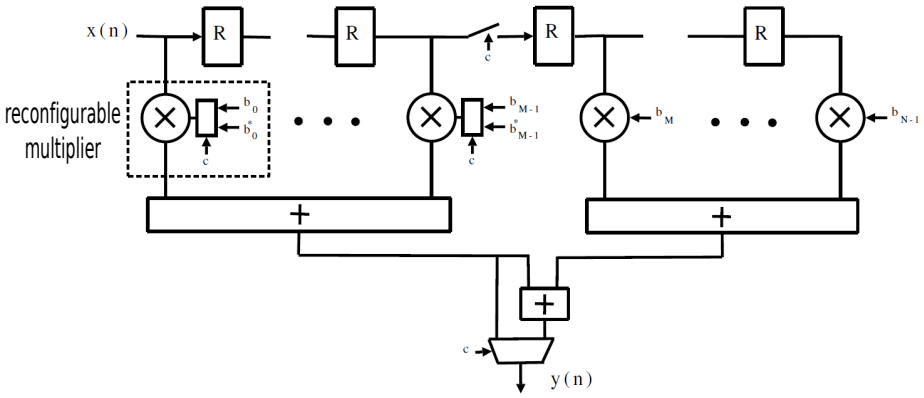


Figure 4.6: FIR filter with backup mode enabling power savings published in [84].

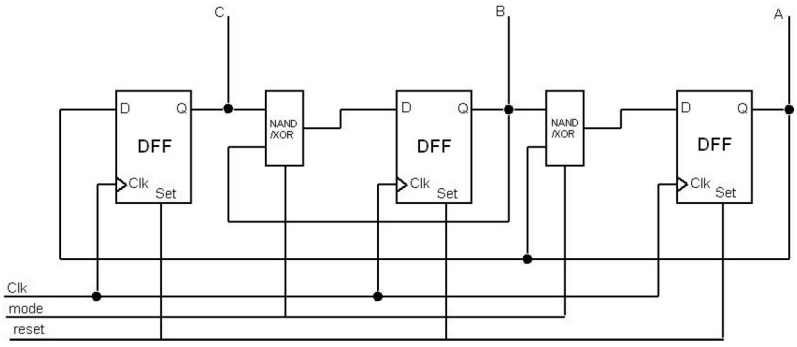


Figure 4.7: Multi-functional circuit which operates as a controller [69] with different operational modes.

by oscillations at the C_{out} output while the control signal of polymorphic gates oscillates. The logic circuitry of the self-checking full-adder can be seen in Figure 4.8. Note that this adder utilizes the NAND/NOR gates and has reasonable area overhead.

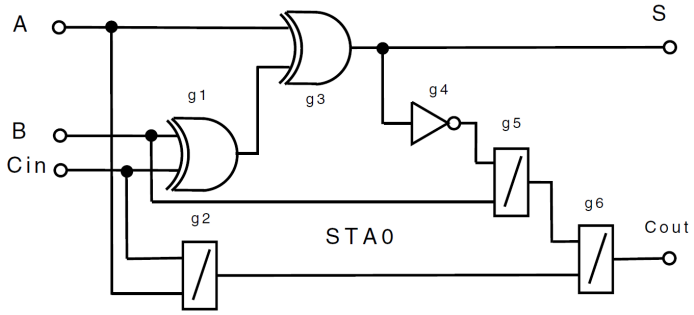


Figure 4.8: Full adder with the self-checking feature taken from [81]. It utilizes NAND/NOR polymorphic gates.

Chapter 5

Goals

Previous chapters have shown that Cartesian Genetic Programming is applicable to design and optimization of the digital circuits. However, we would like to improve some parts of the algorithm in order to get better solutions.

At the beginning, we need to put several questions to ourselves in the field of circuit design and optimization that will lead our research. Is it possible to optimize current implementations of application-specific circuits with the evolutionary algorithm? And what algorithm or its extension is suitable to be involved in the optimization process? How shall we modify conventional circuit synthesis methods in order to allow them to operate with polymorphic gates? Can we minimize the number of gates in polymorphic circuits that we have designed by "adapted" conventional methods? What benchmarks should be utilized to compare the results?

For answering these questions, we have defined the following goals for the doctoral thesis:

- To employ Cartesian Genetic Programing and its modifications as an evolutionary algorithm in the ordinary-circuit optimization with the goal of improving existing results of combinational circuit synthesis.
- To utilize specific polymorphic gates in ordinary circuits to minimize the circuit size in terms of the number of utilized gates. Cartesian Genetic Programing will be used with the modified fitness function.
- To propose and investigate new approaches to polymorphic-circuit design based on polymorphic multiplexing and polymorphic Binary Decision Diagrams. We will compare the solutions obtained by these two new approaches with solutions obtained from Cartesian Genetic Programing. Since there is not available any benchmark set of polymorphic circuits, we will introduce a new set of benchmark circuits to evaluate and compare the synthesis algorithms.
- There is a possibility that the polymorphic multiplexing and polymorphic Binary Decision Diagrams will not be able to produce area-inefficient solutions. Thus, the goal is to employ the extended Cartesian Genetic Programing to optimize a polymorphic circuit at the gate level.

In order to fairly compare the results of various methods, the solutions will mainly be sought in the form of circuits composed of the two-input gates.

Chapter 6

Extensions of Standard Cartesian Genetic Programming

For purposes of this thesis, it is useful to extend Cartesian Genetic Programming (CGP). The extension is necessary for improving the search-space-exploration capabilities and reducing the computation overhead. The extension is derived from the standard CGP introduced in Section 3.3. It should be noted that the definition of a chromosome and its phenotype remains unchanged.

6.1 Design Phase and Optimization Phase

For our experiments, it is suitable to divide the CGP runtime into two phases. I call them as *the design phase* and *the optimization phase*. The design phase is usually performed before the optimization phase in CGP (see Figure 6.1). *The design phase* is typically performed for searching of a brand-new correct circuit. The search is usually conducted from an initial population of random individuals. It employs a fitness function which evaluates the circuit behavior only. When a candidate circuit conforms to the behavioral specification, then the number of gates becomes important. *The optimization phase* is typically performed to optimize (e.g., the number of gates) a fully functional circuit.

The division into two phases offers us better conditions for setting the CGP. Each phase can utilize different parameters, such as the size of a population, mutation operator, selection method or fitness function. Note that the design phase and optimization phase can be executed independently. In other words, the optimization phase does not have to follow the design phase and also the design phase does not have to precede the optimization phase. The design phase can be substituted by conventional design tools, such as SIS or the ABC System.

6.2 Modified Fitness Functions

In order to avoid a conditionally defined fitness function (see the when-otherwise condition in Equation 3.1), it is useful to define a new fitness function where priorities are set for various objectives. The intended approach allows us to aggregate various objectives into a single fitness value. Since the goal is to minimize, the new definition also has a new global extreme, the minimum, which is the value of 1. Note that the problem is not considered as multi-objective in this thesis.

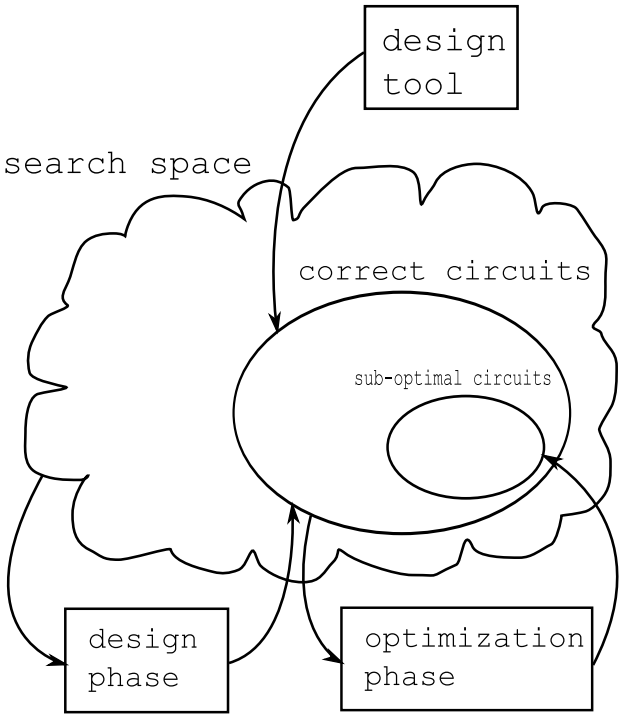


Figure 6.1: Relations among the search space, design phase and optimization phase.

The new fitness function of one objective is defined as follows:

$$\begin{aligned} fitnessp_1 &= 1 + q_1.g_1; \\ g_1 &= 1, \end{aligned} \tag{6.1}$$

where q_1 is the objective (the goal is to minimize q_1 here) and g_1 is generic constant.

The new fitness function of two objectives (priorities) is defined as follows:

$$\begin{aligned} fitnessp_2 &= 1 + q_2.g_2 + q_1.g_1; \\ g_1 &= 1, \\ g_2 &= (f_{max}(q_1) + 1).g_1, \end{aligned} \tag{6.2}$$

where q_2 is the more preferred objective over q_1; g_1, g_2 are generic constants and $f_{max}(q_1)$ gives the maximum value of the q_1 objective.

The resulting generic fitness function for n objectives (priorities) is defined as follows:

$$\begin{aligned} fitnessp_n &= 1 + q_n.g_n + .. + q_1.g_1; \\ g_1 &= 1, \\ g_i &= (f_{max}(q_{i-1}) + 1).g_{i-1}, i = 2..n, \end{aligned} \tag{6.3}$$

where q_j is j-th objective, g_j is a generic constant and $f_{max}(q_j)$ is the maximum possible value of q_j for $j = 1..n$. Furthermore, it holds that q_j is preferred objective over q_{j-1}. One can observe that the objectives are mutually disjunctive.

The standard fitness function for circuit evolution (Equation 3.1) can be redefined ac-

cording to Equation 6.2 as follows:

$$\begin{aligned} fitnessp'_2 &= 1 + \bar{b}.g'_2 + z.g_1 = 1 + \bar{b}.(n_c.n_r + 1) + z, \\ &= 1 + \bar{b}.n_c.n_r + \bar{b} + z; \\ g_1 &= 1, \\ g'_2 &= (f_{max}(z) + 1).g'_1 = n_c.n_r + 1, \end{aligned} \tag{6.4}$$

where $\bar{b}$ is the number of incorrect bits obtained for all possible input combinations; z is the number of utilized gates; and $n_c.n_r$ is the total number of programmable elements. The number of incorrect bits, $\bar{b}$, can be understood as Hamming distance between the truth table of the candidate circuit and the truth table of the desired circuit.

The $fitnessp'_2$ function (Equation 6.4) is applicable in both the design phase and optimization phase. However, as we will show later, it is useful (for some cases) to define the fitness function as follows:

$$\begin{aligned} fitnessp'_1 &= 1 + \bar{b}.g_1 = 1 + \bar{b}; \\ g_1 &= 1. \end{aligned} \tag{6.5}$$

6.3 Selection Strategies

From the perspective of the thesis, the fitness function and selection strategy are the most interesting features of the standard CGP. The *selection strategy* defines which candidate acts as a parent in the new population. Because the $(1 + \lambda)$ strategy is often used in CGP, the lowest-scored individual is always preserved. The result of evolution is then just the lowest-scored individual of the last generation.

Consider a situation in which a fully working circuit has already been obtained ($\bar{b} = 0$) in the design phase and the number of gates is being optimized now. If the mutation operator creates an individual, x, with fitness value f_x from a parental individual, ρ, and $f_x \leq f_\rho$ then x will become a new parental solution ρ' (assuming that there is no better result of mutation in the population). However, if the mutation operator creates individual y with the fitness value f_y and $(f_y > f_\rho) \wedge (f_y \leq 1 + n_c.n_r)$ then ρ will be selected as the parent for the new population and y will be discarded (assuming that the fitness values of other solutions are higher than f_y). In this way, many new fully functional solutions are lost.

The new selection strategy and fitness function is proposed *only* for the situation when the number of gates is optimized, that is, the fitness value of the best individual is lower than $1 + (n_c.n_r + 1)$. Otherwise, the algorithm works as the standard CGP. Since the best individual found so far will not be copied to the new population automatically, it is necessary to store it in an auxiliary variable. Let β denote the best discovered solution and f_β be its fitness value. In the first population, β is initialized using ρ.

Assume that $x_1 \ldots x_\lambda$ are individuals (with fitness values $f_{x_1} \ldots f_{x_\lambda}$) created from the parental solution ρ using the mutation operator and $f_\beta \leq 1 + n_c.n_r$ (that is, we are in the gate reduction phase now). Because the best individual β and parental individual ρ are not always identical, we have to determine their new instances β' and ρ' separately. The best-discovered solution is defined as:

$$\beta' = \begin{cases} \beta & when \quad f_\beta \leq f_{x_i}, i = 1 \ldots \lambda, \\ x_j & otherwise, \end{cases} \tag{6.6}$$

where x_j is the lowest-scored individual for which $f_{x_j} < f_\beta$ holds. If multiple individuals exist that have lower fitness than f_β in $\{x_1 \dots x_\lambda\}$, we will choose the best one of them. The new parental individual is defined as:

$$\rho' = \begin{cases} \rho & when \quad \forall i, i = 1 \dots \lambda : f_{x_i} \geq 1 + (n_c.n_r + 1) \\ x_j & otherwise, \end{cases} \tag{6.7}$$

where x_j is a randomly selected individual from those in $\{x_1 \dots x_\lambda\}$ which obtained the fitness score lower than $1 + (n_c.n_r + 1)$. In other words, the new parent must be a fully functional solution; however, the number of gates is not important for its selection. Note that the result of evolution is no longer ρ but β. The proposed strategy will be denoted as the *SeS-2 strategy* and the traditional strategy as the *SeS-1 strategy*.

6.4 Short-Circuit Evaluation

The fitness evaluation procedure which probes every assignment to the inputs (i.e., $0..2^n-1$) is the main time bottleneck. We will show that when a correct circuit is finally found ($\bar{b} = 0$ in Equation 6.4) and the goal is to minimize the number of gates, it is possible to reduce the evaluation time. The accurate evaluation is important when a correct circuit has not been found yet. In that case, we have to know the fitness score as precisely as possible (that is, the exact number of bits has to be calculated) in order to obtain a relatively smooth fitness landscape.

Short-circuit evaluation is an evaluation enhancement proposed in this thesis. In order to make the evaluation of a candidate circuit as quickly as possible (assuming that the parent circuit is correct), it is only tested whether a candidate circuit is working correctly or incorrectly. In the case that a candidate circuit does not produce a correct output value for the p-th input vector during the evaluation, the remaining $2^n - p - 1$ vectors are not evaluated and the circuit gets the worst possible score ($\bar{b} = f_{max}(\bar{b})$). Experimental results (in Section 7.3.1) will show that this technique reduces the computational overhead significantly.

6.5 Training-Set Reorganization

One can observe that ordered test vectors $(0..2^n - 1)$ could lead to a situation when some parts of a circuit are not tested in the earlier phases of evaluation. Hence the *training-set reorganization* is proposed to reorder the input vectors (and corresponding output vectors) so that more circuit parts are tested from the beginning of the evaluation. This strategy could be beneficial if the candidate circuit would have many inputs and the short-circuit evaluation would be involved. The reorganization itself is done by changing the order of test vectors randomly. Figure 6.2 illustrates the reorganization of vectors when the parallel simulation (see Section 3.3.2) is involved.

The advantage of this approach is that the reorganization proceeds only in the initialization phase of an evolutionary algorithm and thus it has no additional overhead on an evaluation process. The influence of the reorganization on the evaluation time will be covered in Section 7.3.1.

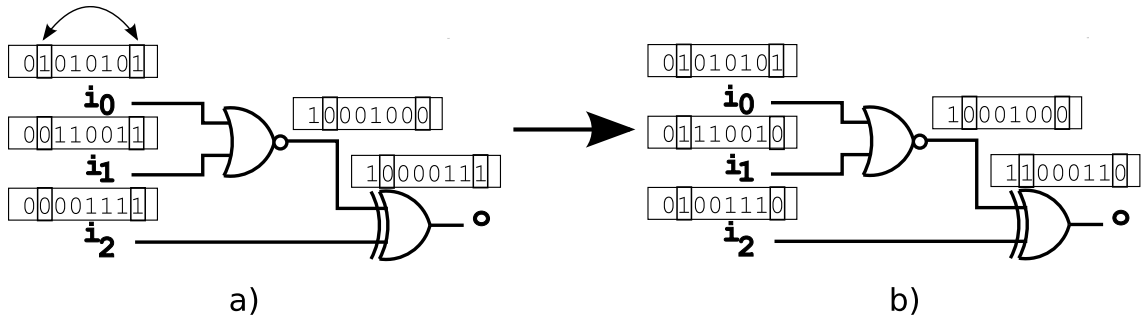


Figure 6.2: Reorganization of bit pairs when the parallel simulation is involved: a) before the reorganization, b) after the reorganization

Chapter 7

Evolutionary Optimization of Circuit Designs

There are many applications of Cartesian Genetic Programming (CGP) in which the objective is not only to evolve a fully functional solution. One or several additional parameters have to be optimized in order to obtain a competitive solution. Evolutionary design of digital circuits is a typical example [30]. In order to compete with conventional synthesis algorithms, evolved circuits should be smaller, faster or less power consuming in comparison to circuits which have been designed conventionally. Because the requirement of correct functionality is more important than the additional requirements, the design problem is not usually treated as multi-objective. Hence CGP is used to find a fully functional solution first, and then the obtained solution is optimized to meet other criteria. The subsequent optimization can be formulated as a multi-objective problem, for example, when we are looking for a suitable trade off between the circuit size and delay. In practical design, the fitness function is explicitly modified after achieving a fully functional solution to reflect the additional criteria. For the rest of the chapter, let us assume that there is only one additional criterion – to find as compact a phenotype as possible, that is, to minimize the number of gates.

In this chapter, we will show that Cartesian Genetic Programming can lead to area-efficient digital circuits even if the requirement on the gate reduction *is not* specified explicitly. It is shown that the specific selection strategy (SeS-2) can provide more compact circuits than the standard CGP. It is interesting that this strategy does not prefer explicitly smaller phenotypes in a parent selection. The strategy is especially well-performed when it is applied in the post-synthesis phase to optimize the circuits which have been synthesized using conventional methods. This phenomenon has been examined on multiplier circuits, majority circuits and the LGSynth91 benchmark circuits. The compared strategies are the selection strategies, SeS-1 and SeS-2, which were introduced in Section 6.3. We will also answer an obvious question on how it is possible that the size of phenotype can be minimized implicitly in the SeS-2 strategy.

7.1 Benchmark Problems

Selection strategies will be evaluated in the task of combinational circuits synthesis. Small multipliers, majority circuits and the LGSynth91 benchmark circuits will be used as test circuits.

7.1.1 Small binary multipliers

Design of small binary multipliers is the most popular benchmark problem for gate level circuit evolution. Because the direct CGP approach is not scalable, it works only for the 4bx4b multipliers (i.e. 8-input/8-output circuits) and smaller. Table 7.1 summarizes the best known results for various multipliers according to [59, 104]. In those experiments, CGP has been used with two-input AND, XOR gates and the AND gate with one input inverted ($\bar{x} \wedge y$). Parameters of CGP were set as $l = n_c$, $\lambda = 4$ and the mutation operator has modified three genes of chromosome; the remaining parameters were given in Table 7.1. CGP was initialized by conventional designs. The fitness function was constructed according to Equation 3.1 or its equivalent representation in the SeS-1 selection strategy.

For this class of circuits, CGP is capable of creating innovative designs, that is, circuits containing fewer gates than the best conventional implementations. However, it is important to carefully initialize CGP parameters. For example, in order to reduce the search space the gate set should contain just the logic gates that are important for multipliers (the solutions denoted as CGP in Table 7.1 were obtained using $\Gamma' = \{x.y,\ x \oplus y,\ \bar{x}.y\}$). However, the gate realizing $\bar{x}.y$ is not usually considered as a single gate in digital design. Its implementation can be constructed using two gates, the AND and NOT gates. Hence, we also included CGP* to Table 7.1 which is the result obtained when one considers the realizing $\bar{x}.y$ as two gates in the multipliers shown in [104].

Table 7.1: The number of two-input gates in the best implementations of multipliers according to [59, 104].

Multiplier	Conventional	CGP	CGP*	$n_r \times n_c$	Generation Limit
2bx2b	8	7	9	1×7	10,000
3bx2b	17	13	14	1×17	200,000
3bx3b	30	23	25	1×35	20,000,000
4bx3b	47	37	44	1×56	200,000,000
4bx4b	64	57	67	1×67	700,000,000

7.1.2 Binary majority circuits

Binary majority circuits are relatively small circuits that are useful for testing the short-circuit evaluation method (see Section 6.4). Since the number of inputs are high and search space is relatively small, the evaluation can be terminated early for almost all candidate solutions.

7.1.3 The LGSynth91 benchmarks

For further comparison of the selection strategies, we have selected 16 circuits from the LGSynth91 benchmark suite [113] (see Table 7.6). As selected circuits have up to 14 inputs and 16 outputs, they are suitable for evolutionary optimization, that is, the scalability problem is not significant. In this case, we have utilized CGP in the post-synthesis phase, that is, CGP is employed to reduce the number of gates in already synthesized circuits. In this experiment, we have used the ABC-System synthesis tool [2] to perform the (conventional) synthesis as a replacement of the design phase in the evolutionary algorithm. Each circuit is represented as a netlist of gates in the BLIF format (Berkeley Logic Interchange Format). This format is transparent because it is a list of all interconnected single-output

combinational gates in the human-readable form. And so, it can be adopted by other synthesis methods such as CGP. In addition, the ABC-System tool can utilize the user-defined set of available gates for the synthesis process.

7.2 Experimental Setup

CGP is used according to its definition in Section 3.3. The initial population is generated either randomly or by using fully functional solutions obtained from conventional synthesis methods or from the best known conventional designs. When CGP is applied as a post-synthesis optimizer, then the number of gates in the conventional synthesis result will be denoted m (it is assumed that each gate possesses up to 2 inputs). In cases of the random initialization, the parameter n_c is set according to the expected circuit size. Experience shows that some redundancy in genotype size is beneficial to the search process(see Section 3.3.2) [61].

CGP will operate with parameters $n_c = m, n_r = 1, l = n_c, n_a = 2$. This setting imposes no structural restrictions on generated combinational circuits. In all experiments, the population size is $1 + 14$. For the design phase, the mutation operator modifies three genes (integers) of the chromosome. For the optimization phase, the mutation operator modifies from 1 to 14 genes with the uniform distribution (it modifies seven genes on average). We have used the gate set $\Gamma = \{x.y,\ x+y,\ \bar{x},\ \overline{x.y},\ \overline{x+y},\ x \oplus y,\ x,\ 0,\ 1\} \equiv \{$AND, OR, NOT, NAND, NOR, XOR, $identity$, $const_0$, $const_1\}$ where NOT and $identity$ are unary operations (taking the first input of an element) and $const_k$ is constant generator with the value k (taking also the first input of an element). If it is not mentioned explicitly, then each experiment is repeated 10 times with a 100 million generation limit. In all experiments, we compare the SeS-1 and SeS-2 selection strategies (see Section 6.3). We also examine the short-circuit evaluation (Section 6.4) on suitable benchmarks and then examine the training-set reorganization, as well.

7.3 Experimental Results

We have performed three classes of experiments. In the first class of experiments, we have investigated the benefits of the short-circuit evaluation and test-vector reorganization. In the second class, we have focused on the post-evolutionary-design optimization. In the third one, we have compared the selection strategies using various benchmarks.

7.3.1 Evaluation time reduction

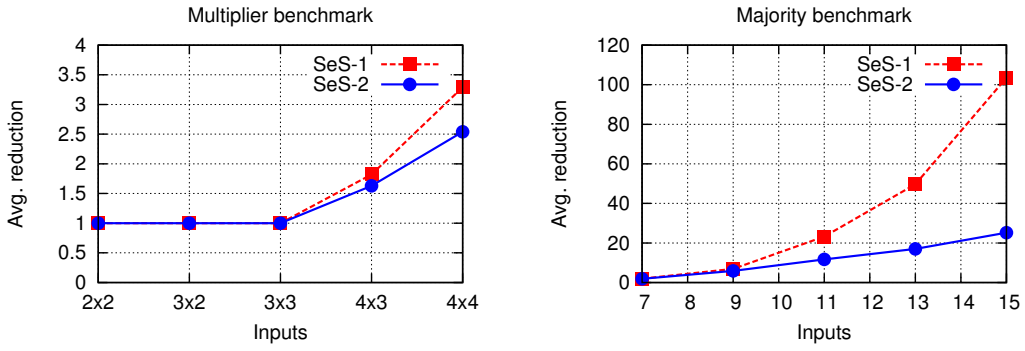
The first class of experiments is focused on the benefits of short-circuit evaluation and training-set reorganization. Table 7.2 shows the results of the experiments. We can see that the effect of the short-circuit evaluation becomes observable when a benchmark has more than 6 inputs. It is due to the 64-bit parallel simulation ($64 = 2^6$) of candidate circuit performed on the 64-bit processor. From the results, it can be seen that a computation is faster when the short-circuit-evaluation enhancement is involved. Figure 7.1 shows a computation time reduction for the selected benchmarks. The initial population was initialized by circuits designed by the ABC-System tool. Since the running overhead of the short circuit evaluation and test-vector reorganization is insignificant, we consider these methods as beneficial.

Table 7.2: The mean number of evaluations of the benchmark circuits (the 'Evals.' column). Meaning of the labels: 'Ins.' and 'Outs.' denote the number of circuit inputs and outputs; 'Reorg?' denotes "Has the training-set reorganization been involved?"; 'Tot. Evals.' denotes the total number of evaluations when the short-circuit evaluation is not involved; 'Reduct.' denotes a reduction of a computational overhead; and 'Corrects' denotes non-destructive mutations.

Benchmark	**Ins.**	**Outs.**	n_c	**Strategy**	**Reorg?**	**Tot. Evals.** [.10^6]	**Evals.** [.10^6]	**Reduct.**	**Corrects** [.10^6]
2x2 Mult.	4	4	17	SeS-1	n/a	1,400	1,400	1	284.5
				SeS-2					226.5
3x2 Mul.	5	5	16	SeS-1	n/a	1,400	1,400	1	23.0
				SeS-2					19.1
3x3 Mult.	6	6	57	SeS-1	n/a	1,400	1,400	1	152.8
				SeS-2					208.1
4x3 Mult.	7	7	125	SeS-1	no	2,800	1,677	1.67	121.2
					yes		1,541	1.82	136.5
				SeS-2	no		1,806	1.55	293.7
					yes		1,716	1.63	310.3
4x4 Mult.	8	8	269	SeS-1	no	5,600	2,060	2.72	98.3
					yes		1,704	3.29	93.2
				SeS-2	no		2,555	2.19	290.2
					yes		2,204	2.54	262.6
7b Maj.	7	1	27	SeS-1	no	2,800	1,457	1.92	40.6
					yes		1,442	1.94	39.7
				SeS-2	no		1,490	1.88	79.5
					yes		1,476	1.9	75.0
9b Maj.	9	1	47	SeS-1	no	11,200	1,792	6.25	35.1
					yes		1,624	6.9	28.8
				SeS-2	no		2,104	5.32	80.2
					yes		1,906	5.88	70.4
11b Maj.	11	1	67	SeS-1	no	44,800	2,810	15.94	19.3
					yes		1,926	23.26	14.9
				SeS-2	no		4,673	9.59	85.3
					yes		3,844	11.66	77.8
13b Maj.	13	1	97	SeS-1	no	179,200	5,876	30.5	12.5
					yes		3,608	49.66	16.0
				SeS-2	no		13,373	13.4	71.1
					yes		10,521	17.03	70.5
15b Maj.	15	1	127	SeS-1	no	716,800	18,620	38.5	14.1
					yes		6,926	103.49	10.0
				SeS-2	no		42,036	17.05	54.4
					yes		28,398	25.24	52.0

If we look at the results of the 4bx3b Multiplier or 13-bit Majority benchmarks in Table 7.2, we will see that the number of correct individuals (see the 'Corrects' column) is higher for enabled reorganization than for a disabled one. We can also observe that the proportion of evaluations and correct circuits is different in this case. Let us suppose that direct proportion exists between the amount of correct individuals and the amount of evaluations, thus the benefit of the reorganization is evident.

The evaluation enhancements do not influence the quality of resulting circuits but they



a) *b)*

Figure 7.1: Average reduction of the computational time in comparison with the complete truth table evaluation for: a) the Multiplier benchmark; and b) the Majority benchmark.

are capable of accelerating the evolutionary process significantly.

7.3.2 Evolution from a random population

In the second class of experiments, we have evolved multipliers with up to four-input operands from the initial population generated randomly. According to recommendations of [61], we intentionally allowed relatively large topologies to be used by CGP. The topology is always $1 \times n_c$ where the n_c values were set on the basis of the ABC-System synthesis results (see Table 7.4).

Table 7.3 summarizes the number of gates (the best and mean values), and the mean number of generations to reach fully functional solutions and the success rate. As design of the 2bx2b and 3bx2b multipliers is easy for CGP, we have mainly analyzed the results for larger problem instances. It can be seen in Table 7.3 that the SeS-2 strategy gives better results in terms of the best and mean number of gates than the SeS-1 strategy. However, the mean number of generations is higher for the SeS-2 strategy. We have obtained almost identical minimal number of gates when compared with [104] (also in Table 7.1, the 'CGP' label), even when CGP is randomly initialized and the non-specific set of gates is utilized.

7.3.3 Post-synthesis optimization

The third class of experiments compares the selection strategies when CGP is applied to reduce the number of gates in already functional circuits.

We compared three approaches of seeding the initial population in the case of multipliers. The resulting multipliers from the ABC-System tool (denoted as 'seed:ABC' in Table 7.4) are taken as seeds in the first group of experiments. The second group of experiments is initialized by the best multipliers (denoted as 'seed:Table 7.1' in Table 7.4) reported in paper [104]. The seeds of the third group of experiments are created generically as the combinational Carry-Save multipliers (denoted as 'seed: Comb. Mult.' in Table 7.4) according to [112]. Table 7.4 shows that the SeS-2 strategy can produce more compact designs (see the column labeled by 'Best') than the SeS-1 strategy. The mean numbers of gates are given in generations of 1, 2, 5, 10, 20, 50 and 100 million. If the higher number of generations is used then the smaller circuit can be obtained.

Table 7.3: The best-obtained and mean number of gates for the Multiplier benchmark when CGP starts from a randomly generated initial population. The 'Mean Gens.' label denotes the mean number of processed generations to find a correct individual.

Circuit	Strategy	n_c	Best	Mean	Mean Gens.	Success
2bx2b	SeS-1	7	7	7	2,738	100%
	SeS-2		7	7	2,777	100%
3bx2b	SeS-1	16	13	13	651,297	100%
	SeS-2		13	13	741,758	100%
3bx3b	SeS-1	57	25	27.7	476,812	100%
	SeS-2		23	23.4	625,682	100%
4bx3b	SeS-1	125	46	52.7	2,714,891	100%
	SeS-2		37	43.1	4,271,179	100%
4bx4b	SeS-1	269	110	128.3	29,673,418	90%
	SeS-2		60	109.4	37,573,311	70%

The best-evolved 4bx4b multiplier is shown in Figure 7.2. It is composed of 56 gates (taken from Γ, which does not consider the AND gate with one input inverted as a single gate). The best circuit presented in [104] consists of 57 gates taken from Γ', or 67 gates if the Γ gate set is used. We can also express the implementation cost in terms of transistors usage. While the multiplier shown in Figure 7.2 is composed of 400 transistors, the multiplier reported in [104] is composed of 438 transistors. It is assumed that the number of transistors to create a particular gate [112] is the following: 4 for the NAND gate; 4 for the NOR gate; 6 for the OR gate; 6 for the AND gate; 2 for the NOT gate; and 10 for the XOR gate.

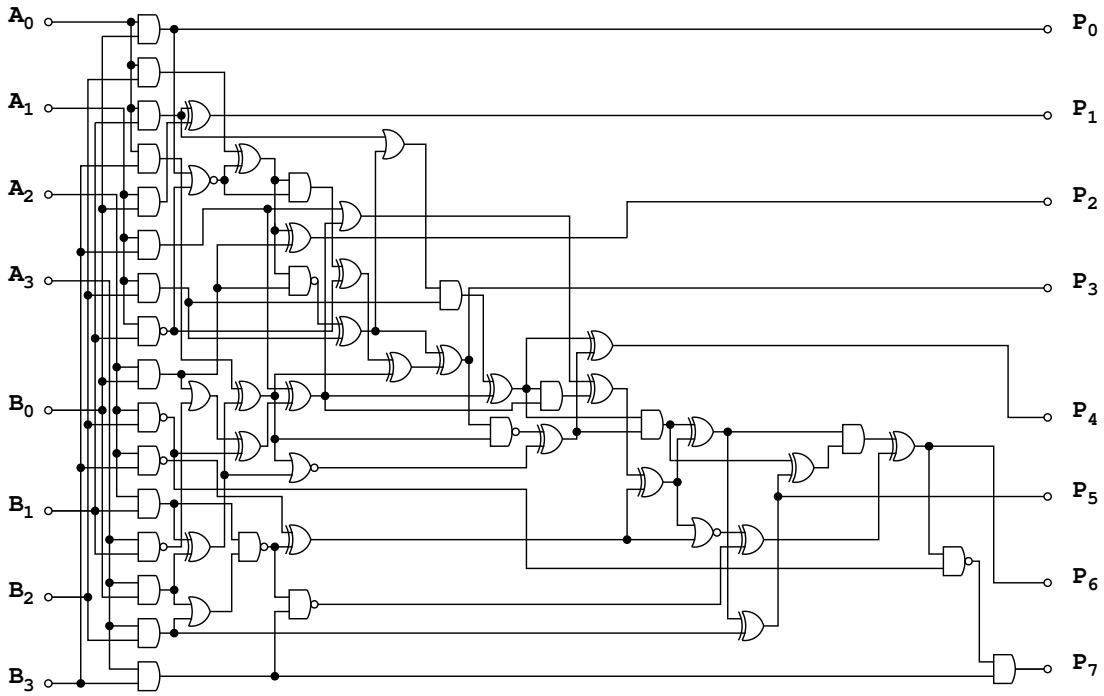


Figure 7.2: Circuit scheme of the best evolved 4bx4b multiplier

Figure 7.3a compares the number of gates of the best-obtained individual, β, during

Table 7.4: The best-obtained and the mean numbers of gates in generations (the 'Gens.' label) from 1 to 100 million for the Multiplier benchmarks when the evolutionary algorithm is initialized by functional solutions of different origin. The 'Strg.' label denotes the selection strategy.

seed: ABC	**Strg.**	n_c	**Best**	1.10^6 **Gens.**	2.10^6 **Gens.**	5.10^6 **Gens.**	10.10^6 **Gens.**	20.10^6 **Gens.**	50.10^6 **Gens.**	100.10^6 **Gens.**
2bx2b	SeS-1	17	7	7	7	7	7	7	7	7
	SeS-2		7	7	7	7	7	7	7	7
3bx2b	SeS-1	16	13	13	13	13	13	13	13	13
	SeS-2		13	13	13	13	13	13	13	13
3bx3b	SeS-1	57	26	38.2	36.1	34.3	32.6	31	29.8	28.7
	SeS-2		23	31.5	28.8	27.2	25	24.5	24.2	23.5
4bx3b	SeS-1	125	54	93.2	88.3	79.3	75.6	71.6	66.6	64.4
	SeS-2		37	80	68	55.9	49.9	46.9	44.1	41.1
4bx4b	SeS-1	269	140	212.4	190.6	178.9	170.9	165.2	158.5	152.4
	SeS-2		68	218.2	182.2	151.3	136.5	121.2	107	93.3
seed: Table 7.1	**Strg.**	n_c	**Best**	1.10^6 **Gens.**	2.10^6 **Gens.**	5.10^6 **Gens.**	10.10^6 **Gens.**	20.10^6 **Gens.**	50.10^6 **Gens.**	100.10^6 **Gens.**
2bx2b	SeS-1	9	7	7	7	7	7	7	7	7
	SeS-2		7	7	7	7	7	7	7	7
3bx2b	SeS-1	14	13	13	13	13	13	13	13	13
	SeS-2		13	13	13	13	13	13	13	13
3bx3b	SeS-1	25	23	25	25	24.7	23.9	23.5	23.2	23.1
	SeS-2		23	25	25	24.7	24.4	24.2	23.5	23.1
4bx3b	SeS-1	44	36	38.5	37.8	37.1	36.8	36.8	36.4	36.3
	SeS-2		35	37.9	37.1	36.5	36.4	36.2	36.2	36.1
4bx4b	SeS-1	67	57	59.6	58.8	58	57.8	57.5	57.3	57.1
	SeS-2		56	59.5	59.2	58.7	58.3	57.2	56.8	56.8
seed: Comb.Mult.	**Strg.**	n_c	**Best**	1.10^6 **Gens.**	2.10^6 **Gens.**	5.10^6 **Gens.**	10.10^6 **Gens.**	20.10^6 **Gens.**	50.10^6 **Gens.**	100.10^6 **Gens.**
2bx2b	SeS-1	8	7	7	7	7	7	7	7	7
	SeS-2		7	7	7	7	7	7	7	7
3bx2b	SeS-1	17	13	13	13	13	13	13	13	13
	SeS-2		13	13	13	13	13	13	13	13
3bx3b	SeS-1	30	23	28	28	28	27.8	27.6	26.5	25.8
	SeS-2		23	28	28	27.6	26.8	25	24.4	23.4
4bx3b	SeS-1	45	37	43	43	43	42.4	41.9	40.6	39.2
	SeS-2		37	43	43	42.6	42.2	41.5	39.9	38.4
4bx4b	SeS-1	64	59	62.9	62.6	62.6	62.3	61.5	60.6	60.2
	SeS-2		59	62.9	62.9	62.8	62.4	62	61.3	60.8

the progress of evolution of the 4bx4b multipliers for the SeS-1 and SeS-2 strategies (the values were taken from the best run; CGP was initialized by the ABC-System-designed individual). Figure 7.3b shows the number of gates of the parent individual, ρ, during the same experiment. Figures 7.4a,b also show the number of gates of the best-obtained-individual, β, and the parent-individual, ρ, but they represent the mean values of 10 runs for each strategy. It can be seen that the parent is different from the best-obtained solution for the SeS-2 strategy (the curve is not monotonic). We can also observe that the SeS-1 strategy provides better results than the SeS-2 strategy in the earlier stages of the evolution. However, the SeS-2 strategy outperforms the SeS-1 strategy when more generations are used

for the evolution.

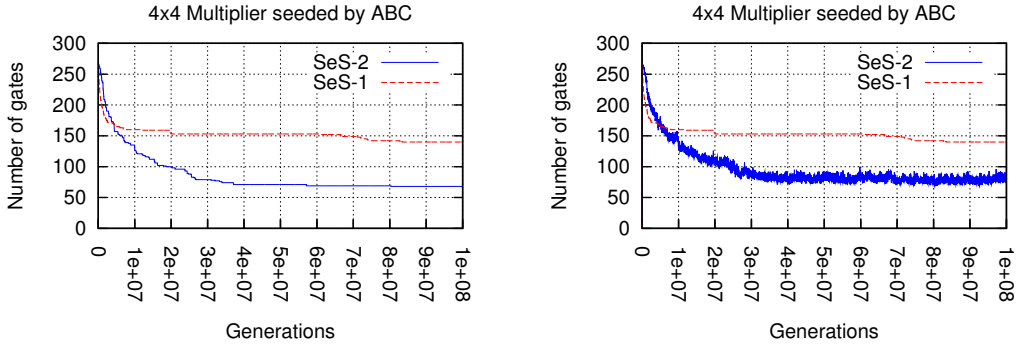


Figure 7.3: The number of gates during the evolution run: a) the best-obtained individual, β; b) parent individual, ρ. Measurements of the best runs of the 4bx4b Multiplier benchmarks which have been initialized by the ABC-System design.

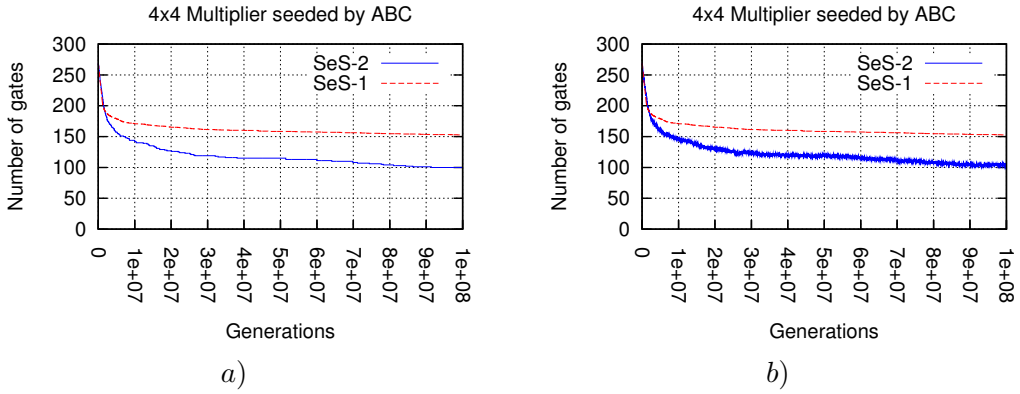


Figure 7.4: The mean number of gates during ten independent runs of the 4bx4b Multiplier benchmark: a) the best-obtained individual, β; b) parent individual, ρ.

Table 7.5 shows what has happened when more generations are used. For each benchmark circuit and strategy (SeS-1 and SeS-2), two groups of runs with a different generation limit were performed (30 runs per 100 million generations and 3 runs per 1 billion generations). The total number of evaluated individuals is the same for both groups. It can be seen that the SeS-2 strategy achieves better minimization results than the SeS-1 strategy when more generations are used.

Table 7.6 gives the best-obtained and mean number of gates for the LGSynth91 benchmark circuits when the evolutionary algorithm starts from already working circuits. The initial circuits, seeds (of the size given by n_c), were obtained from the original LGSynth91 library of circuit netlists (in the BLIF format). They were mapped using the ABC-System tool on two-input gates of the Γ set. The 'Expect. Gates' is the estimated number of gates (after a synthesis) given in [113]. It can be seen that the SeS-2 strategy is more successful than the SeS-1 strategy in most cases. In general, CGP gives better results than

Table 7.5: The best-obtained, worst and mean number of gates for the Multiplier and Majority benchmarks when evolved with a different generation limit. CGP was initialized by the ABC designs. The 'Ins.' and 'Outs.' labels denote number of circuit inputs and outputs. The 'G.p.R.' label denotes the number of generations per one run.

Benchmark	**Ins.**	**Outs.**	**Strategy**	**Runs**	**G.p.R.** [$.10^6$]	**Best**	**Worst**	**Mean**
4bx3b Multiplier	7	7	SeS-1	30	100	54	78	67.73
			SeS-2	30	100	36	51	41.50
			SeS-1	3	1,000	48	74	63.33
			SeS-2	3	1,000	35	36	35.67
4bx4b Multiplier	8	8	SeS-1	30	100	140	163	152.87
			SeS-2	30	100	65	125	90.53
			SeS-1	3	1,000	128	143	137.33
			SeS-2	3	1,000	56	58	57.00
9b Majority	9	1	SeS-1	30	100	31	38	34.40
			SeS-2	30	100	24	32	28.77
			SeS-1	3	1,000	28	31	29.33
			SeS-2	3	1,000	24	30	27.67
11b Majority	11	1	SeS-1	30	100	48	61	54.97
			SeS-2	30	100	32	53	39.20
			SeS-1	3	1,000	44	53	48.33
			SeS-2	3	1,000	31	42	38.33
13b Majority	13	1	SeS-1	30	100	69	85	79.27
			SeS-2	30	100	46	78	60.77
			SeS-1	3	1,000	75	82	78.00
			SeS-2	3	1,000	47	48	47.33

'Expect. Gates' because it does not employ any deterministic synthesis algorithm; all the optimizations are being done implicitly without any structural biases.

7.4 Summary

We have seen, so far, that the selection of a parent individual on the basis of its bare functionality (and so neglecting the number of gates) provides slightly better results at the end of the evolution (when the goal is to reduce the phenotype size) than the standard approach of CGP. We have experimentally confirmed that CGP can implicitly find a very compact phenotype even if it is not required explicitly. This strategy is especially useful for the optimization of larger circuits where the first fully functional solution discovered by CGP (or its initializing circuit, seed, created by a conventional method) is far from optimum in terms of the phenotype size.

How is it possible that the SeS-2 selection strategy really works? It is important to recall that the fitness-function landscape is rugged and neutral in the case of the digital circuit evolution using CGP [105, 60]. Hence relatively simple mutation-based search algorithms are more successful than sophisticated search algorithms and genetic operators such as those developed in the field of genetic algorithms and estimation of distribution algorithms. In the standard CGP, generating the offspring individuals is biased to the best individual that has been discovered so far. The best individual is changed only if a better or equally-scored solution is found. In the SeS-2 strategy, the changes of the parent individual are more

Table 7.6: The best-obtained and mean number of gates of the LGSynth91 benchmark when CGP starts from the initial solutions (of size n_c) which have been synthesized using the ABC-System tool. Note that the *alu4* benchmark was optimized only three times due to the time-consuming evaluations. 'Expect. Gates' denotes the number of utilized gates which are expected after an optimization [113].

Circuit	**Inputs** n_i	**Outputs** n_o	**Expect. Gates**	**Seed Size** n_c	**Best SeS-1**	**Best SeS-2**	**Mean SeS-1**	**Mean SeS-2**
9symml	9	1	43	216	53	23	68.5	25.5
C17	5	2	6	6	6	6	6	6
alu2	10	6	335	422	134	73	149	89.4
alu4	14	8	681	764	329	274	358	279
b1	3	4	13	11	4	4	4	4
cm138a	6	8	17	19	16	16	16	16
cm151a	12	2	33	34	24	23	24	23
cm152a	11	1	n/a	24	22	21	22.1	21.8
cm42a	4	10	17	20	17	17	17	17
cm82a	5	3	27	12	10	10	10	10
cm85a	11	3	38	41	23	22	24.1	22
decod	5	16	22	34	30	26	30	26.1
f51m	8	8	43	146	29	26	32.9	27.3
majority	5	1	9	10	8	8	8	8
x2	10	7	42	60	27	27	29.6	27.4
z4ml	7	4	20	40	15	15	15	15

frequent because the only requirement for a candidate individual is to be fully functional. Hence we consider that the SeS-2 strategy is more "explorative" than the used one in the standard CGP (the SeS-1 strategy).

Our hypothesis is based on two crucial features of CGP – neutrality of search and redundancy of encoding [105, 61, 60] (see Section 3.3.2). We argue that a consideration of fully functional but not necessarily the smallest-discovered individuals as parents leads to a smaller number of harmful mutations and, therefore, it leads to a better search space exploration in comparison with the standard SeS-1 strategy. The search algorithm then can easily escape from the local extremes in the optimization phase of the SeS-2 strategy.

Our hypothesis is that if a high degree of redundancy is present in the genotype, the SeS-2 selection strategy will generate more functionally correct individuals than the SeS-1 strategy (the standard CGP). And because the fitness landscape is rugged and neutral, then the SeS-2 strategy will be more efficient in finding compact circuit implementations than the SeS-1 strategy (the standard CGP). In order to verify this hypothesis, we have measured the number of mutations that lead to the functionally correct circuits. When CGP is initialized by a working circuit, we have in fact measured the number of neutral or useful mutations. Figure 7.5 compares the results for the SeS-1 and SeS-2 strategies in the experiments that are reported in Tables 7.2, 7.3 and 7.4. The y-axis is labeled as MNM which stands for 'Millions of Non-destructive Mutations'. For small multipliers (2bx2b, 3bx2b), the SeS-1 strategy always yields higher MNM which contradicts our hypothesis. However, these really small multipliers are not interesting because the problem is easy and an optimal solution can be discovered very quickly. In cases of more difficult circuits, the SeS-2 strategy provides higher MNM in most cases, especially when a sufficient redundancy

is available (see Figures 7.5a, 7.5d). When the best resulting multipliers of paper [104] are used to seed the initial population, the SeS-1 strategy has always greater non-destructive mutations than the SeS-2 strategy (see Figure 7.5b). It corresponds with the theory that the evolutionary algorithm (with almost zero redundancy in the genotype) has got stuck at the local extreme and the SeS-2 strategy does not have any space to work.

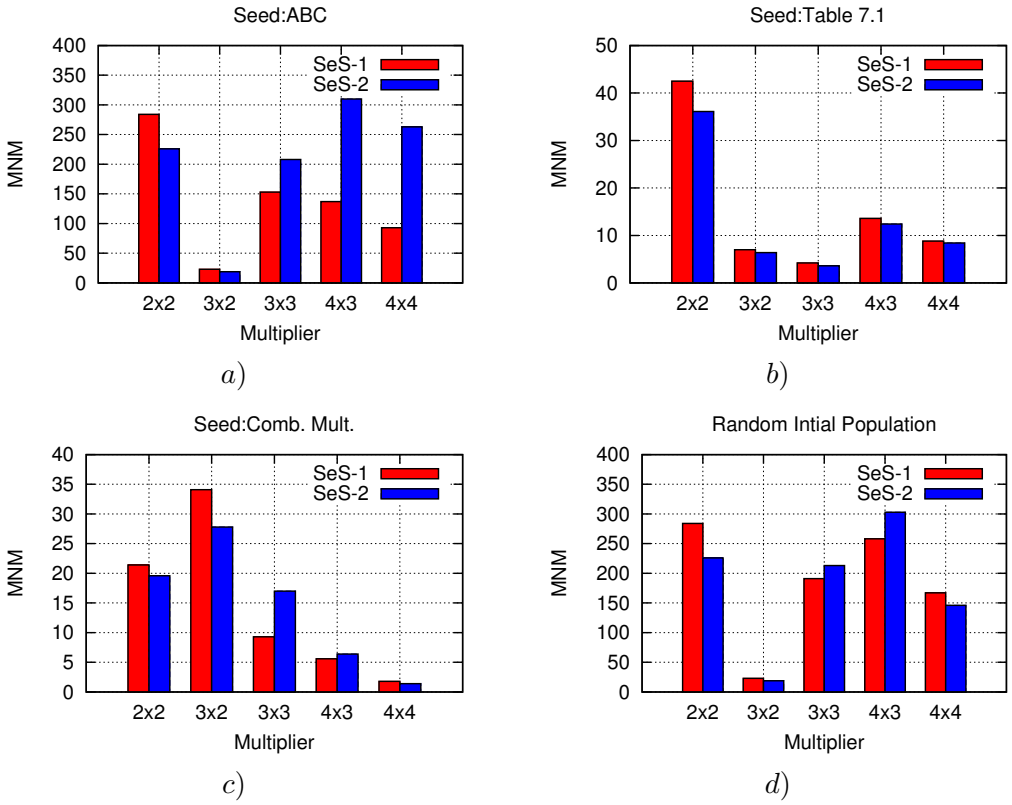
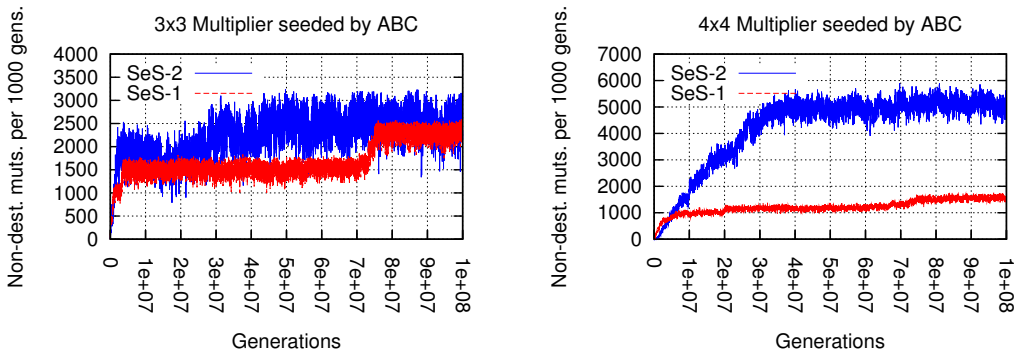


Figure 7.5: Millions of Non-destructive Mutations (MNM) for different experiments (mean values given)

The number of non-destructive mutations was counted every 1,000 generations and the resulting value was plotted as a single dot to Figure 7.6a (the 3bx3b multiplier) and Figure 7.6b (the 4bx4b multiplier). The best runs initialized by the ABC-System designs are shown in both cases. It is evident that significantly more correct individuals on average have been generated for the SeS-2 strategy. It can be seen that while the SeS-1 strategy tends to create a relatively stable amount of correct individuals in time (the dispersion is approximately 200 individuals for the 4bx4b multiplier), great differences are observable in the amount of correct individuals for the SeS-2 strategy (the dispersion is approximately 1,000 individuals for the 4bx4b multiplier). That also supports the idea of the biased search of the SeS-1 strategy.



a) *b)*

Figure 7.6: The number of non-destructive mutations per 1,000 generations of: a) the 3bx3b Multiplier benchmark; b) the 4bx4b Multiplier benchmark

Chapter 8

Utilization of Polymorphic Gates in Evolutionary Design

This chapter shows that the evolutionary design of digital circuits, which is conducted at the gate-level, is able to produce human-competitive circuits at the transistor-level. In addition to the standard gates, we utilize polymorphic gates (such as the NAND/NOR and NOR/NAND gates; see Table 4.1 for gates which are controlled by an external control signal) which consist of a few transistors, but exhibit non-trivial three-input logic functions. The proposed implementations of adders and majority circuits, which utilize these gates, can contain fewer transistors than conventional implementations of these circuits. Moreover, the utilization of these unconventional gates can significantly improve the success rate of a search process. It should be noted that we will not utilize polymorphic gates controlled by the power-supply voltage in this chapter.

Stoica et al. noted [91]: "From the evolvable hardware perspective, it is interesting to have programmable granularity, allowing the sampling of novel architectures together with the possibility of implementing standard ones. The optimal choice of elementary block type and granularity is task dependent." Therefore, its designer's challenge to define the representational bias in order to obtain a suitable search space within the entire space of possible solutions.

While the evolutionary design conducted at a certain level is able to provide optimized solutions for a particular level (such as the transistor level), it is very difficult to evolve at such a particular level more complicated circuits that are typically designed at a higher level (such as the gate level). On the other hand, when the evolution is conducted at a higher level, then a particular complex solution might be found; however, it is not usually optimal from the perspective of lower levels which have to be taken into account when the evolved circuit has to be fabricated.

This fact is also visible in the standard design process of digital circuits. Complex digital circuits are designed at the RT level (register-transfer level) or at the gate level. Various optimization techniques are utilized to obtain an optimized gate-level circuit. Optimized solution is then implemented using well-known transistor-level implementations of standard logic gates which are available in the so-called *cell library*. However, the final transistor-level solution might not be optimal at all.

For example, the logic expression $Y = \overline{A.B + C.D}$ (a corresponding circuit is shown in Figure 8.1a) seems to be optimized at the gate level. It requires two AND gates, along with a single OR gate and inverter; it costs 20 transistors (see Figure 8.1c). However, the function

Y can be implemented using eight transistors only when the special AND-OR-Invert circuit is employed (see Figure 8.1b). The reason is that the transistor level allows implementing this particular circuit much easier than a standard gate-level optimization suggests. As circuit designers know this fact, they do map RT-level designs to the elementary components of the cell library that are optimized for a particular fabrication process.

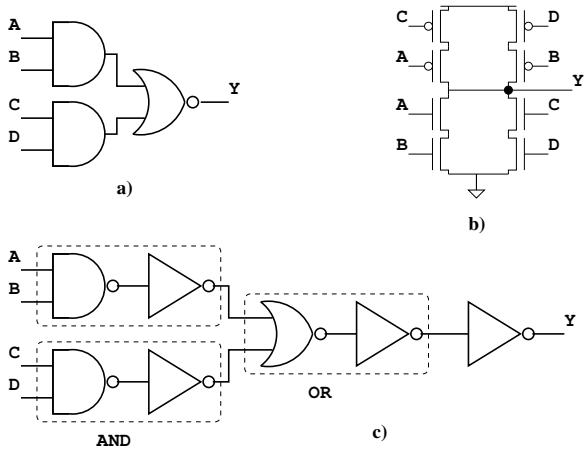


Figure 8.1: And-Or-Invert Circuit: a) the gate-level implementation, b) the transistor-level (CMOS) implementation, c) a naive gate-level implementation

Unfortunately, in the field of evolutionary design of digital circuits, designers usually do optimize the number of standard gates. They ignore the real circuit cost [59, 106, 9, 121] on a chip (perhaps with the exception of [1]). Although the evolutionary circuit design has proven that it is able to generate the gate-level circuits which lie beyond the scope of human designs [59, 44, 38], these benefits have not been demonstrated so far for nontrivial digital circuits considered at the transistor level. Note that some implementations of simple gates were evolved at the transistor level [38, 49, 117]; however, as mentioned above, the evolutionary design of more complicated digital circuits directly at the transistor level is currently outside the capabilities of commonly available computers. In particular, Langeheine argues that the evolutionary design of a single XOR gate is quite difficult [49].

We will utilize Cartesian Genetic Programming (CGP) to evolve gate-level combinational circuits with the aim of minimizing the number of transistors in target designs. The goal is to show that the utilization of special elementary circuit components (polymorphic gates), which are optimized for the transistor level, can reduce the number of transistors in some digital circuits. The adders and majority circuits are examined as benchmark circuits.

8.1 Gate-Level and Transistor-Level Designs

Let us consider a one-bit full adder. This circuit has two operands, A and B, and the input carry, C_{in}. It generates the sum:

$$S = A \oplus B \oplus C_{in}; \tag{8.1}$$

and the output carry:

$$C_{out} = AB + BC + AC \tag{8.2}$$

These equations, 8.1 and 8.2, have been derived from the truth table of this circuit and optimized using standard operations of Boolean algebra [109]. Since the circuit contains a

three-input XOR gate, three two-input AND gates and a three-input OR gate, it utilizes 38 transistors. In order to calculate the number of transistors, we have to look at Table 8.1 which gives the *typical* number of transistors for the implementation of digital gates.

Table 8.1: Logic gates and the typical number of transistors to implement them.

Gate	**Inputs**	**Notation**	**Transistors**
AND	2	AND2	6
	3	AND3	8
OR	2	OR2	6
	3	OR3	8
NOT	1	NOT	2
XOR	2	XOR2	8/10
	3	XOR3	12/16
NAND	2	NAND2	4
	3	NAND3	6
NOR	2	NOR2	4
	3	NOR3	6
MUX	2+1	MX	6
NAND/NOR	2+1	NAND/NOR	10
NOR/NAND	2+1	NOR/NAND	8

However, a standard static CMOS VLSI implementation of the one-bit full adder utilizes only 24 transistors [112] (see Figure 8.2). The number of transistors can even be reduced to 22 when so-called transmission gates are utilized [122]. Zimmermann and Gupta compared different implementations of full adders in terms of area, delay and power consumption [123]. The implementation cost of adders is strongly connected with the cost of the XOR gate. While the standard static CMOS two-input XOR gate is implemented using 10 transistors, only 8 transistors are sufficient if transmission gates can be utilized. Cheng and Hsieh compared various implementations of the three-input XOR gate [10]. Their paper also shows that the three-input XOR gate can be implemented using 12 transistors.

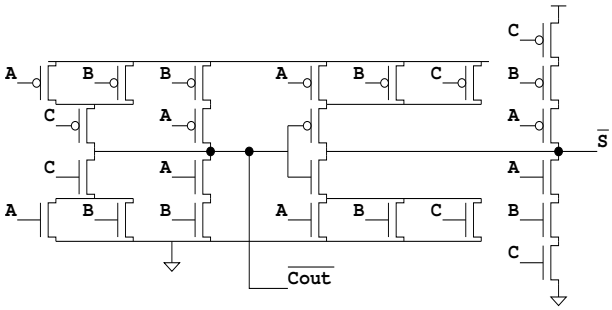


Figure 8.2: Standard 24-transistor implementation of the static one-bit full adder

Miller et al. evolved a very unconventional implementation of the one-bit full adder [59]. Although they have not optimized the number of transistors, Figure 8.6a shows that the evolved circuit can be implemented using 22 transistors only as it contains two two-input XOR gates and a single multiplexer.

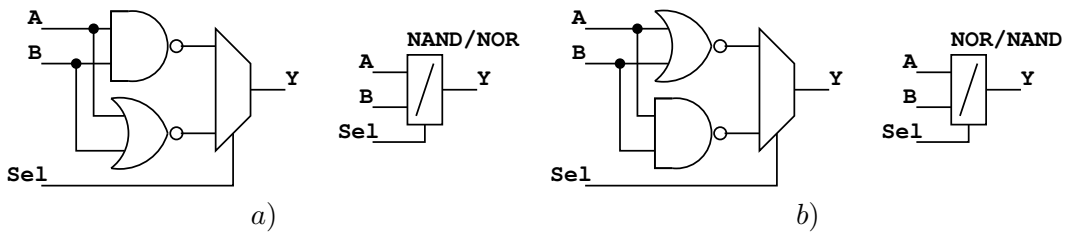
8.2 The NAND/NOR and NOR/NAND Polymorphic Gates

We intend to utilize two polymorphic (unconventional) three-input gates [89] that are not usually considered by circuit designers. It will be shown in Section 8.4 that the evolutionary design approach is able to effectively utilize these uncommon components as building blocks for more complex circuits.

The NAND/NOR gate as well as the NOR/NAND gate has three inputs, A, B and Sel, and operates according to Table 8.2. As Figure 8.3 shows, these gates perform multiplexing of the NAND logic operation and the NOR logic operation according to a selection input, Sel. Table 8.1 suggests that their implementation should utilize 14 transistors (6 transistors for the multiplexer, 4 transistors for the NAND gate and 4 transistors for the NOR gate). However, the NAND/NOR gate can be implemented using 10 transistors only (see Figure 8.4a), while the implementation of the NOR/NAND gate utilizes only 8 transistors (see Figure 8.4b).

Table 8.2: Truth tables of the NAND/NOR and NOR/NAND gates

Sel	A	B	NAND/NOR	NOR/NAND
0	0	0	1	1
0	0	1	1	0
0	1	0	1	0
0	1	1	0	0
1	0	0	1	1
1	0	1	0	1
1	1	0	0	1
1	1	1	0	0



a) *b)*

Figure 8.3: Gate-level implementation of the NAND/NOR gate (a) and the NOR/NAND gate (b).

8.3 Experimental Setup

Extended Cartesian Genetic Programming (see Chapter 6) was employed as an evolutionary algorithm for the design of circuits. This extension utilizes the polymorphic gates, which are controlled by a control signal together with ordinary gates. A candidate digital circuit is represented using a one-dimensional array of programmable nodes, $n_c \times 1$. Each programmable element has three inputs and a single output. It can be programmed to realize one of logic operations specified in the Γ_u gate set (see Table 8.1).

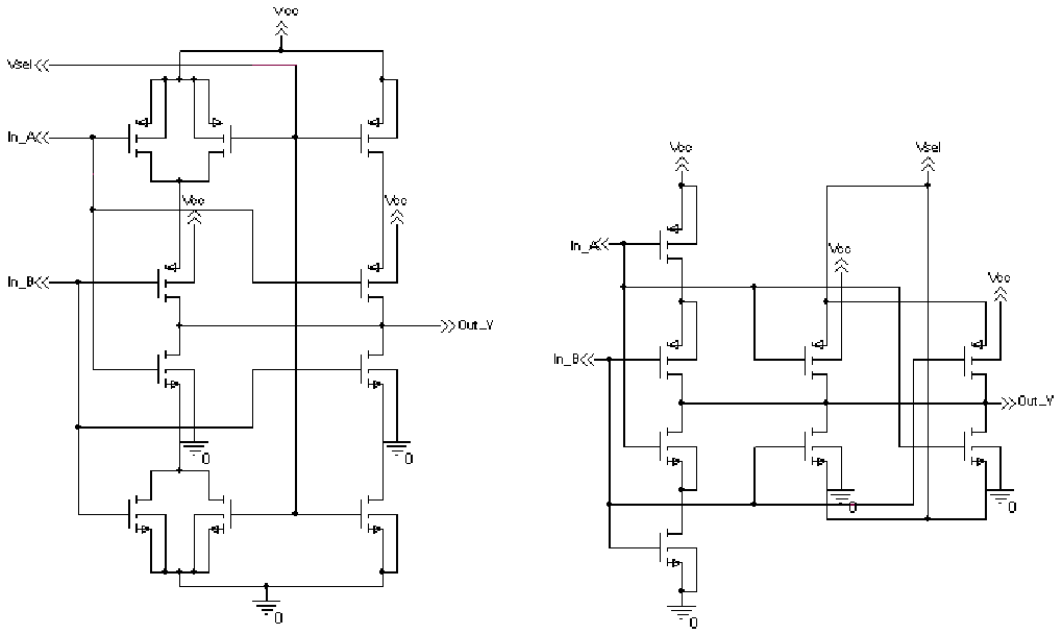


Figure 8.4: CMOS implementation of the NAND/NOR gate (a) and the NOR/NAND gate (b) gate proposed in [89].

Digital circuits are encoded as arrays of integers of the size $4n_c + n_o$, where n_c is the number of elements and n_o is the number of circuit outputs. Figure 8.5 shows a circuit consisting of three inputs, two outputs and five programmable elements. The encoding of circuits corresponds to the encoding which was introduced in Section 3.3.

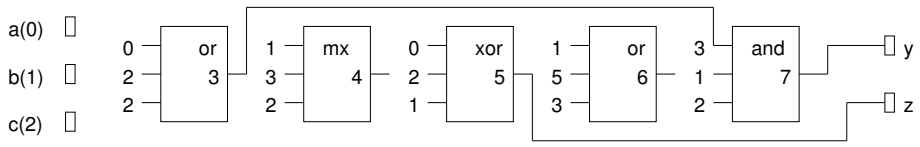


Figure 8.5: Example of encoding of a three-input/two-output circuit. Chromosome is encoded as: 0,2,2,0; 1,3,2,3; 0,2,1,2; 1,5,3,0; 3,1,2,1; 7,5. Logic operations are encoded as: 0 for OR, 1 for AND, 2 for XOR, 3 for MX. Note that only the elements labeled by 3, 5 and 7 are utilized in the phenotype.

We do change the mutation rate after achieving a perfect functionality, that is, before a transistor utilization is minimized. For the design phase, the algorithm modifies $h_d = 3$ genes of the chromosome. For the optimization phase, the algorithm modifies from 1 to 14 genes (h_o) of the chromosomes (that is seven genes on average). The algorithm operates with a population of $1 + \lambda$ individuals (in this case $\lambda = 14$). The initial population is randomly generated.

The fitness function for the design phase reflects Equation 6.5. In addition to maximizing the functionality, we minimize the number of transistors, the number of gates and delay (at the gate level) in the optimization phase. This is achieved by using a prioritized fitness

function which is derived for four objectives from Equation 6.3:

$$fitnessp_3 = 1 + q_4.g_4 + q_3.g_3 + q_2.g_2 + q_1.g_1, \tag{8.3}$$

where q_4 denotes the number of incorrect bits computed by the candidate circuit (which possess the highest priority), q_3 is the total number of utilized transistors, q_2 is the number of utilized gates and q_1 denotes delay of the circuit (which possess the lowest priority). Generic constants, g_0, g_1, g_2 and g_3, are generated as in Equation 6.3. The number of transistors is calculated for the gates utilized in the phenotype according to Table 8.1 (XOR2 utilizes 10 transistors and XOR3 utilizes 12 transistors). Other parameters remain the same as in Chapter 6 (or in Section 3.3) or are specified in the following section.

8.4 Experimental Results

This section reports the experimental results which were obtained for four target benchmarks: one-bit full adder, two-bit adder, five-input majority circuit and seven-input majority circuit.

8.4.1 The one-bit full adder

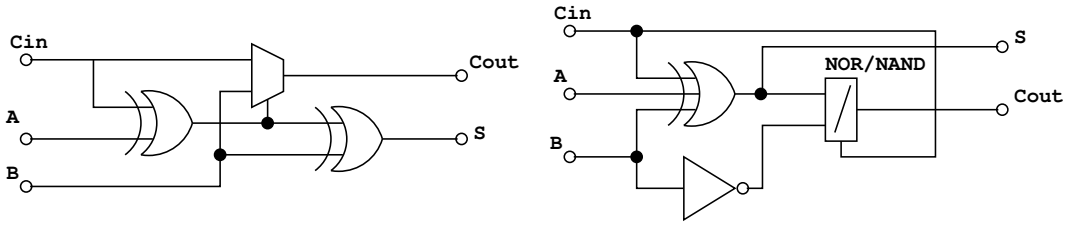
In order to evolve a one-bit full adder, we set the size of genotype to 20×1 and the generation limit up to 1,000,000 generations. We tested different combinations of gates in the Γ set and different transistor utilizations for those gates. We have run 50 independent experiments for each set of gates. Table 8.3 lists: the gates included into Γ; the average number of generations (AvgG) needed to find a fully functional solution (not necessarily optimized for transistor utilization); and examples of best-evolved implementations. We can observe that the 22-transistor implementation was found for all proposed combinations of gates. Figure 8.6 shows examples of the best implementations.

Table 8.3: Resulting implementations of the 1-bit full adder for different sets of gates, $\Gamma_1..\Gamma_6$. The number of utilized transistors is given for each gate.

Gates	AvgG	XOR2	XOR3	MX	NOT	NAND/NOR	NOR/NAND	Best Implementations
Γ_1	386.9	10	12	6	2	-	-	3.MX+2.NOT = 22 tr. (C1)
Γ_2	334.9	10	12	6	2	10	8	NOR/NAND+XOR3+NOT = 22 tr. (C2)
Γ_3	367.4	10	16	6	2	-	-	C1
Γ_4	259.3	10	16	6	2	10	8	C1 and C2
Γ_5	329.2	8	12	6	2	-	-	2.XOR2+MX = 22 tr.
Γ_6	394.9	8	12	6	2	10	8	C2

8.4.2 The two-bit adder

The two-bit adder calculates the two-bit sum and output carry for two two-bit operands and the input carry, that is, the circuit has five inputs and three outputs. In order to investigate which set of gates is suitable for this problem, we compared 10 different gate sets (denoted as #1..#10 in the following tables). We set a genotype topology to 60×1 and a generation limit of up to 1,000,000 generations. We performed 100 independent experiments for each gate set.



a) *b)*

Figure 8.6: Evolved 1-bit full adders

The first part of Table 8.4 provides the mean number of utilized gates in the best-obtained circuits of all runs. Thus, we can observe those gates that are useful for this particular problem. An empty space means that a gate was not included in the set of gates. The second part of Table 8.4 gives the mean values for some parameters calculated from the best-obtained circuits of the 100 independent runs: the number of utilized transistors, the number of utilized gates, circuit delay (measured at the gate level), the fitness value and the number of generations. 'Succ. Rate' denotes the number of runs (out of 100 runs) in which a correctly working circuit was evolved.

Table 8.4: The two-bit adder benchmark. The mean values of some parameters are calculated from 100 independent runs for ten different sets of gates.

Gate Set	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10
AND2s	3.27	0.18	0.03	11	7.93					
OR2s	2.13	0.13	0.1		7.6					-
XOR2s	3.95	1.55	0.44							
NOTs	0.69	0.99	1.22	17	2.59			2.9	2.52	-
NAND2s	0.86	0.22	0.11			17.71		15.69		
NOR2s	0.88	0.2	0.07				17.26		15.95	
AND3s		0	0	4	0.12					
OR3s		0.01	0		0.21					-
XOR3s		0.93	1.53							
NAND3s		0	0			1.13		1.45		
NOR3s		0.01	0.01				1.44		1.25	
MXs		3.55	0.88							
NAND/NORs			0.55							
NOR/NANDs			1.51							
Transistors	80.24	53.62	49.62	132	101	77.68	77.82	77.24	76.36	
Gates	11.78	7.77	6.45	32	18.45	18.84	18.74	20.03	19.73	
Delay	5.56	4.05	3.56	10	8.03	7.9	7.88	7.45	7.57	
Fitness	2,535,619	2,882,840	2,867,875	22,408,827	2,166,756	5,737,087	5,452,684	5,879,844	4,878,742	
Generations	38,686.8	32,793.2	21,386.8	997,862	156,150	863,785	855,925	848,595	755,830	
Succ. Rate	100%	100%	100%	1%	100%	31%	34%	29%	44%	0%

Figure 8.7 shows the best implementation of the two-bit adder that we have evolved. By connecting two conventional one-bit full adders or two Miller's one-bit full adders [59] into the Carry-Ripple adder, we can obtain a solution with 44 transistors. Evolved solution utilizes two NOR/NAND gates, two three-input XOR gates and a NOT gate, so it utilizes 42 transistors.

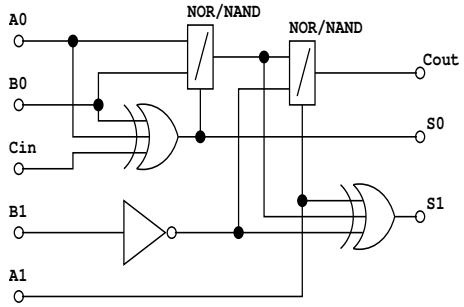


Figure 8.7: Evolved two-bit adder

8.4.3 Majority circuits

If only the two-input AND and two-input OR gates can be utilized, then according to a sorting network-based implementation [40], the five-input majority circuit consists of 10 such gates (that is 60 transistors) and the seven-input majority circuit consists of 20 such gates (that is 120 transistors).

In order to evolve a five-input majority circuit and seven-input majority circuit, we have set the genotype topology to 60×1 and the generation limit up to 10,000,000 generations. Tables 8.5 and 8.6 summarize the mean values of parameters calculated from 100 independent runs for ten different sets of gates. Both tables illustrate that the utilization of uncommon gates significantly improves the quality of evolved solutions (see the #3 column).

Table 8.5: The five-input majority benchmark. The mean values of some parameters are calculated from 100 independent runs for ten different sets of gates.

Gate Set	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10
AND2s	4.59	0.55	0.06	8	3.28					
OR2s	4.23	0.75	0.12		2.99					8.67
XOR2s	0.38	0.32	0.01							
NOTs	0.11	0.42	0.1	12	0			2.97	2.76	11.33
NAND2s	1.17	1.47	0.27			11.36		9.23		
NOR2s	1.03	0.99	0.19				11.8		9.45	
AND3s		0.3	0.09	2	1.3					
OR3s		0.27	0.05		1.45					2.67
XOR3s		0.33	0.01							
NAND3s		0.15	0			2.25		2		
NOR3s		0.08	0.03				2.01		2.09	
MXs		3.25	0.36							
NAND/NORs			0.95							
NOR/NANDs			2.48							
Transistors	65.74	51.08	36.14	88	59.62	58.94	59.26	54.86	55.86	96
Gates	11.51	8.88	4.72	22	9.02	13.61	13.81	14.2	14.31	22.67
Delay	5.49	4.51	3.18	7	4.61	5.12	5.23	5.59	5.49	7.33
Fitness	2,481,648	2,873,457	2,817,610	5,445,075	2,012,203	1,563,433	1,564,636	1,548,288	1,552,015	5,388,845
Generations	40,677.9	12,130.5	9,348.8	996,910	7,461.89	37,914.7	35,905.9	37,436.9	49,098.4	990,743
Succ. Rate	100%	100%	100%	1%	100%	100%	100%	100%	100%	3%

Table 8.7 shows basic properties of selected five-input majority circuits that we have evolved. It can be seen in Figure 8.8 that the best circuit has only 32 utilized transistors, that is, three NOR/NAND gates, one NOR gate and one NAND gate. Without the

Table 8.6: The seven-input majority benchmark. The mean values of some parameters are calculated from 100 independent runs for ten different sets of gates.

Gate Set	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10
AND2s	9.67	1.54	0.59	-	-					
OR2s	8.67	1.87	1.21		-					-
XOR2s	1.08	1.29	0.16							
NOTs	0.14	0.79	0.65	-	-			5	6.12	-
NAND2s	0.83	1.68	0.61			17		19.6		
NOR2s	1.72	1.18	0.63				20.14		15.75	
AND3s		0.48	0.28	-	-					-
OR3s		0.63	0.32		-					-
XOR3s		1.33	0.21							
NAND3s		0.6	0.13			7.5		4.2		
NOR3s		0.56	0.1				5.14		5.12	
MXs		7.73	1.84							
NAND/ORs			3.21							
NOR/NANDs			3.36							
Transistors	131.44	124.6	97.49			113	111.43	113.6	106	
Gates	22.11	19.68	13.32			24.5	25.29	28.8	27	
Delay	8.44	7.55	5.68			6.5	7.86	9	8.62	
Fitness	5,399,959	3,392,049	3,156,677			9,046,069	8,728,553	7,006,954	7,309,033	
Generations	849,817	370,296	135,659			973,505	971,021	979,242	979,381	
Succ. Rate	36%	94%	98%	0%	0%	4%	7%	5%	8%	0%

utilization of the uncommon gates, the algorithm is able to find the implementation which utilizes 40 transistors. Note that a conventional implementation of this benchmark contains 60 transistors. The transistor-optimized conventional solution, which was proposed in [67], utilizes 38 transistors.

Table 8.7: Some properties of the best evolved five-input majority circuits.

Circuit	#1	#2	#3	#4	#5	#6
Ordinary Gates	8	2	7	0	8	7
NAND/NORs	–	0	–	1	–	–
NOR/NANDs	–	3	–	3	–	–
Transistors	40	32	42	34	46	48
Gates	8	5	7	4	8	7
Delay	4	3	4	3	3	3

Table 8.8 shows basic properties of selected seven-input majority circuits that we have evolved. It can be seen in Figure 8.9 that the best circuit has only 60 utilized transistors, that is, two NOR/NAND gates, two NAND/NOR gates and three three-input XOR gates. The best evolved circuit, which utilizes standard gates, utilizes 86 transistors.

8.5 Summary

Our task was to design non-trivial digital circuits (more complicated than a single gate) and, simultaneously, to create such implementations of those circuits that are optimized for the target platform, that is, rather for the transistor level than for the gate level. As we

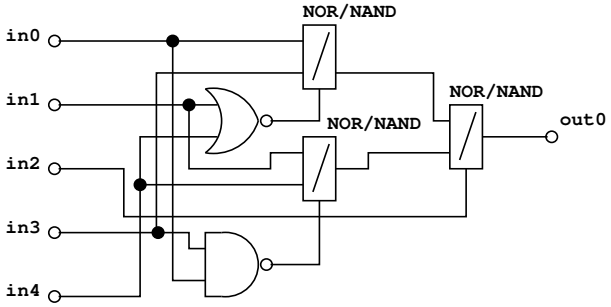


Figure 8.8: Evolved five-input majority circuit

Table 8.8: Some properties of the best evolved seven-input majority circuits.

Circuit	#1	#2	#3
Ordinary Gates	15	2	13
NAND/NORs	–	2	–
NOR/NANDs	–	2	–
Transistors	86	60	96
Gates	15	6	13
Delay	9	3	6

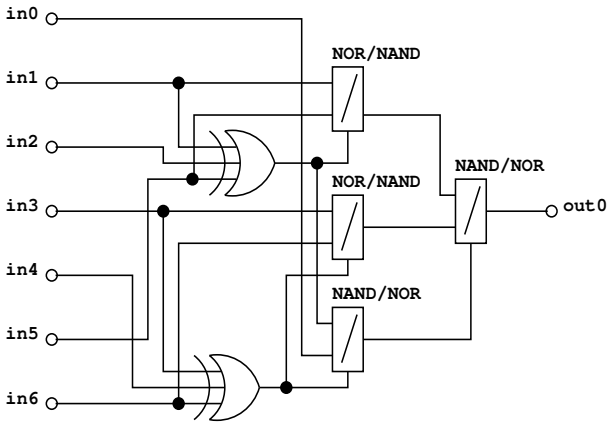


Figure 8.9: Evolved seven-input majority circuit

are not able to evolve these circuits directly at the transistor level[1], the evolutionary design and optimization were performed at the gate level. However, the implementation cost of candidate circuits was evaluated at the transistor level.

Except for the standard gates, we utilized two polymorphic gates which were well-suited as basic components for the circuits that we wanted to evolve. In most cases, evolved solutions utilize fewer transistors than well-optimized transistor-level conventional implementations as well as existing evolved implementations. Experimental results show that the utilization of the uncommon gates significantly helps the evolution to find good solutions

[1]As far as we know, no similar results have been reported in the available literature. Probably because available computing resources are not sufficient.

(see the averages in the #3 column of Tables 8.4, 8.5, 8.6). Table 8.9 summarizes the results. Therefore, the identification of the suitable transistor-level components (that is, the gates such as the NAND/NOR, NOR/NAND or AND-OR-Invert gates) and the utilization of them as building components at the gate level represents a promising method for the design of non-trivial digital circuits which are optimized at the transistor level. As Table 8.3 shows, the method is really able to minimize the number of transistors in target circuits when different implementation costs are assigned to the gates.

Table 8.9: The number of transistors in the best implementations of test circuits. 'EA' denotes the evolutionary algorithm; 'EA+PG' denotes the evolutionary algorithm with polymorphic gates.

	Method		
Circuit	**Conventional**	**EA**	**EA+PG**
1-bit adder	22	22	22
2-bit adder	44	44	42
5-input majority	38	40	32
7-input majority	-	86	60

Another advantage of the proposed method is that when the algorithm operates with gates, and those gates are correctly implemented using transistors, then the final solution will exhibit a desired electrical behavior. It often happens during evolutionary digital circuit design which is conducted at the transistor-level that some transistors are not used as switches (i.e., they operate in the active region). This leads to circuits which exhibit undesired properties such as high power consumption.

On the other hand, we have not considered some important aspects in these experiments. Although we have been optimizing the delay at the gate level, the delay of evolved solutions is not necessarily optimized for the transistor level. We also did not deal with the power consumption of evolved circuits. We did not consider the placement and routing aspects of evolved circuits on a chip. These issues should be investigated in a future paper.

In comparison with Koza's highly computationally-demanding genetic-programming method [44], our approach allowed us to design target circuits in a relatively short time. A single run of the algorithm which uses 60 programmable elements and produces 1,000,000 generations requires 30 seconds for the one-bit full adder, 154 seconds for the two-bit adder (or the five-input majority) and 198 seconds for the seven-input majority circuit on average when they are computed at a common PC which is equipped with an Athlon64 X2 4800+ processor.

Chapter 9

Polymorphic Circuit Design

In the area of polymorphic circuit design, we have focused on the design of bi-functional polymorphic circuits in this thesis. The design space is situated on a gate level. Target circuits consist of polymorphic gates[1] together with ordinary[2] gates.

9.1 Polymorphic Circuit Synthesis Problem

The following problem formulation resembles the definition proposed in [79, 87]. The goal of the bi-functional polymorphic-circuit synthesis can be formulated as the problem of finding such a circuit which performs its required functions [79]: f_1 in the first mode (*mode 1*) and f_2 in the second mode (*mode 2*) of polymorphic gates (see Section 4.1). We will denote a bi-functional polymorphic circuit as f_1/f_2, where f_1 is the logic function of the first mode and f_2 is the logic function of the second mode, *mode 2*.

One must remember that a polymorphic gate is denoted as X_1/X_2, where X_1 is the logic operation of the first mode and X_2 is the logic operation of the second mode. The method used to physically control the mode of polymorphic gates is not important in the proposed synthesis problem formulation. The polymorphic gate implements two operations according to a control signal which can hold two different values. The ordinary gates performs only one operation, however, their functionality must fully be defined for each mode. For example, the conventional NAND gate considered for polymorphic circuits must perform the NAND operation in both modes (denoted as NAND/NAND). Let $\Gamma^{(1)}$ denote a set of ordinary gates. Let $\Gamma^{(2)}$ denote a set of polymorphic gates. Let Γ denote a set of all gates, $\Gamma = \Gamma^{(1)} \cup \Gamma^{(2)}$. The requirements on the Γ set with respect to the logic completeness have been investigated in [50].

Any polymorphic circuit can formally be represented by an oriented graph $G = (V, E, \varphi)$, where V is the set of vertexes; E is the set of edges between the vertexes, $E = \{(a, b)|a, b \in V\}$; and φ is the mapping of logic operation (gate) to each vertex, $\varphi : V \rightarrow \Gamma$. Note that V models the gates and E models the connections of the gates. The circuit (and also its graph) is in the k-th mode if all gates are in the k-th mode.

Given that Γ and logic functions f_1 and f_2 are required in *modes 1 and 2*, the problem of the bi-functional circuit synthesis at the gate level is formulated as follows: find a graph ,G, representing the digital circuit which performs the f_1 logic function in the first mode and

[1] Here, polymorphic gates do not utilize any special control logic signal. Their logic function is controlled using the power-supply-voltage levels, temperature, etc.

[2] We mean standard two gates.

the f_2 logic function in the second mode. Various additional requirements can be specified, such as minimization of delay or area.

Unfortunately, this problem cannot be approached by conventional synthesis methods directly since they do not allow representing polymorphic logic functions and manipulating with them.

The problem formulation can naturally be extended for k different functions according to k modes of polymorphic gates; however, for simplicity, we will deal with bi-functional circuits in this thesis.

Note that paper [53] also deals with the polymorphic-circuits synthesis problem; however, the goal of synthesis is different: a target circuit is constructed to perform a single function independently on the mode of polymorphic gates.

9.2 Proposed Methods

We propose three methods to approach the polymorphic-circuit-synthesis problem in this section. The first one is based on the evolutionary design using Cartesian Genetic Programming. The remaining approaches extend the conventional synthesis methods to be applicable for polymorphic circuits. While Polymorphic Binary Decision Diagrams enable inserting the polymorphic gates at the input part of the target circuit, Polymorphic Multiplexing allows the polymorphic gates to be included at the output part of the target circuit.

9.2.1 Evolutionary Design

The polymorphic circuit synthesis can be approached using extended Cartesian Genetic Programming as it is introduced in Section 3.3. The main extensions of the standard CGP are the extensions of a gate set and the fitness function.

The set of gates, Γ, is extended by polymorphic gates. It should be noted that ordinary gates must be able to work in both polymorphic modes correctly.

The evaluation of the fitness function is performed for each mode of gates separately. The following fitness-function definition is capable of evaluating a candidate circuit in both phases, the design phase and the optimization phase:

$$\begin{aligned} fitnessp_2'' &= 1 + (\bar{b_1} + \bar{b_2}).g_2 + z.g_1 = 1 + (\bar{b_1} + \bar{b_2}).(n_c.n_r + 1) + z, \\ &= 1 + \bar{b_1}.n_c.n_r + \bar{b_2}.n_c.n_r + \bar{b_1} + \bar{b_2} + z; \\ g_1 &= 1, \\ g_2 &= (f_{max}(z) + 1).g_1 = n_c.n_r + 1, \end{aligned} \tag{9.1}$$

where $\bar{b_1}$ (respectively $\bar{b_2}$) is the number of incorrect bits obtained for all possible combinations in the first mode (respectively in the second mode), z is the number of utilized gates and $n_c.n_r$ is the total number of programmable elements. The number of incorrect bits, $(\bar{b_1} + \bar{b_2})$, can be understood as Hamming distance between the obtained truth table and the target truth table. The fitness function (Equation 9.1) is derived from Equations 6.2 and 6.4.

Acceleration of evaluation

For acceleration of the evaluation process, the parallel simulation (see Section 3.3.2) and short-circuit evaluation (see Section 6.4) are utilized. The parallel simulation takes place

in the design and optimization phase, while the short-circuit evaluation takes place only in the optimization phase. The parallel simulation has to be performed for each mode of gates separately (see Figure 9.1).

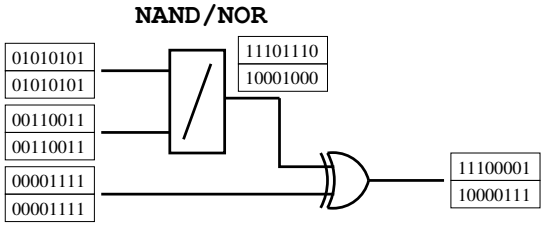


Figure 9.1: Parallel simulation of a candidate polymorphic circuit

The short-circuit evaluation for polymorphic circuits is described and examined in detail in Chapter 10 which also focuses on the optimization phase where the short-circuit evaluation is very useful.

9.2.2 Polymorphic BDD-based Synthesis

We propose *Polymorphic Binary Decision Diagrams* (Polymorphic BDD) to extend the standard Decision Diagrams. We will show how we construct Polymorphic BDD and transform this BDD to the corresponding polymorphic circuits.

Binary Decision Diagrams

Decision Diagram (DD) [13] over a set of Boolean variables, $X_n = \{x_1, .., x_n\}$, and a non-empty terminal set, T, is defined as a directed acyclic graph, $G = (V, E)$, with exactly one root node and the following properties:

- A node in V is a non-terminal or terminal node.
- A non-terminal node is labeled by variable x_i and has two successors $low(x_i)$ and $high(x_i)$ in V.
- A terminal node is labeled with a value from T.

The size of DD is given by the number of its nodes. The level i is the set of nodes labeled by x_i. A DD is ordered, if each variable is encountered at most once on each path from the root to a terminal node and the variables are encountered in the same order on each path.

Binary Decision Diagram (BDD) is defined as DD over X_n [13]; however, its terminal set is $T = \{0, 1\}$. If the BDD has root node v, then BDD represents a Boolean function f_v defined as follows: If v is a terminal node of value 0 (or 1), then $f_v = 0$ (or $f_v = 1$); and if v is a non-terminal node labeled with index x_i then f_v is the function $f_v(x_1, .., x_n) = \bar{x_i}.f_{low(v)}(x_1, .., x_n) + x_i.f_{high(v)}(x_1, .., x_n)$, where $f_{low(v)}$ (or $f_{high(v)}$) denotes the function represented by $low(v)$ (or $high(v)$). We can also call the non-terminal nodes as if-then-else nodes, that is, if x_i is true, then $f_v = f_{high}$ else $f_v = f_{low}$.

Multi-Terminal BDD (MTBDD) [16] is the extension of BDD which allows integers to be placed in terminal nodes. Decision variables still remains Boolean.

We define Polymorphic BDD as MTBDD in which terminal integers represent elementary polymorphic functions.

Design of Polymorphic BDD

Let f_1 denote a target function in the first mode and f_2 denote a target function in the second mode (according to Section 9.1). Let R_1 and R_2 be the truth tables for f_1 and f_2. Let us assume that R_1 and R_2 are fully defined and ordered. Then the design of Polymorphic BDD, which represents f_1 and f_2, is a three-step procedure:

1. Create the truth table, R, of the polymorphic circuit as a composition of R_1 and R_2. The truth table, R, has 2^n rows, n columns with all the input variable assignments and two columns with logic output values for the f_1 and f_2 logic functions.

2. Choose a decision variable, x_c (where c is from 0 to $n-1$), and divide R into $2^n/2$ segments (rows) in such a way that the input assignment differs only in x_c in each segment (see Figure 9.2a). From each segment, extract a signature, $S = 2^3.s_{21} + 2^2.s_{20} + 2^1.s_{11} + 2^0.s_{10} \equiv (s_{21}s_{20}s_{11}s_{10})_2$, that is the least significant bits come from f_1 and the most significant bits come from f_2; the values of variables $s_{21}, s_{20}, s_{11}, s_{10}$ are determined according to a conversion matrix (see Figure 9.2b).

3. Process the resulting $2^n/2$-row truth table R' (see Figure 9.2c) using a chosen algorithm for the MTBDD processing to create an optimized DD structure.

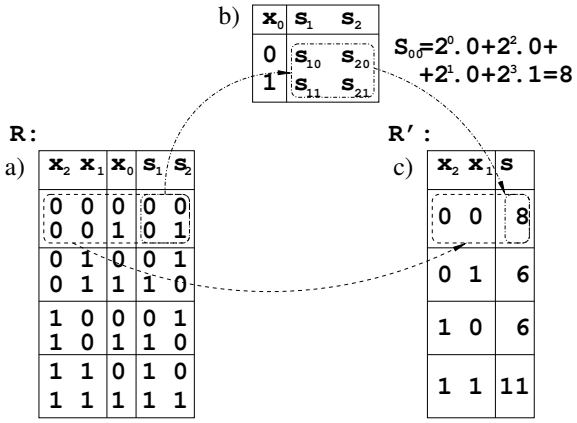


Figure 9.2: Transformation process of the three-bit Majority/Parity truth table from R to R' (s_1 denotes the majority; s_2 denotes the even parity): a) truth table R before transformation; b) conversion matrix of the S signature ; and c) transformed truth table R'.

Figure 9.2 shows an example of processing the R-truth-table (which represents Majority/Parity functionality) into the transformed table, R'. The values of the s column represent simple polymorphic functions (see Table 9.1).

Polymorphic BDD which is constructed using the proposed algorithm can further be optimized by using standard techniques that are applicable for MTBDD. The optimization techniques include:

- Reducing identical (redundant) branches (type I [13]).
- Reducing (redundant) non-terminal (if-then-else) node with the same branches (type S [13]).
- Reordering the input variables.

Table 9.1: Transformation of signatures to elementary polymorphic functions: 'id' denotes the identity function, and 'neg' denotes the logic inversion.

S	s_1 / s_2	S	s_1 / s_2	S	s_1 / s_2	S	s_1 / s_2
0	0 / 0	4	0 / neg	8	0 / id	12	0 / 1
1	neg / 0	5	neg / neg	9	neg / id	13	neg / 1
2	id / 0	6	id / neg	10	id / id	14	id / 1
3	1 / 0	7	1 / neg	11	1 / id	15	1 / 1

An example of a reduced BDD can be seen in Figure 9.4a. This BDD represents the three-bit Majority/Parity function.

Note that for modeling a multi-output problem, BDD can be extended to multi-output BDD simply by the creation of BDD for each output of a modeled problem (circuit). Resulting BDDs can be merged into an extended BDD in order to reduce (redundant) equivalent branches which can be shared.

Synthesis of Polymorphic BDD into polymorphic circuit

Transformation of Polymorphic BDD to the polymorphic circuit is a straightforward procedure. Firstly, the decision non-terminal (if-then-else) nodes are directly mapped onto two-input ordinary multiplexers. The edges of BDD are transformed as signal lines that interconnect the multiplexers. This creates a network of multiplexers. Afterwards, terminal nodes are implemented as elementary polymorphic circuits according to the conversion table shown in Figure 9.3. The elementary polymorphic circuits utilize the ordinary and NAND/NOR polymorphic gates. If, for some reason, another set of gates has to be used, a different conversion table has to be created. Finally, the elementary polymorphic circuits are connected to the multiplexer network corresponding to BDD.

Figure 9.4a shows the reduced Polymorphic BDD for the three-bit Majority/Parity benchmark and Figure 9.4b shows its implementation which utilizes the polymorphic gates together with multiplexers and ordinary gates.

Elementary polymorphic circuits (see Table 9.3) have been created by using the extended CGP for the design of polymorphic circuits. So, it is not a big challenge to create another table for a different set of (ordinary and polymorphic) gates.

9.2.3 Polymorphic Multiplexing

A straightforward approach to the implementation of polymorphic circuits is the utilization of a *polymorphic multiplexer*, *pmux*. This multiplexer propagates the A signal in the first mode and the B signal in the second mode of polymorphic gates. The gate-level implementation of the polymorphic multiplexer, *pmux*, shown in Figure 9.5, is based on the NAND/NOR and ordinary gates. This implementation utilizes five gates. The multiplexer was created using the elementary polymorphic circuits, *id/0*, *0/id* (proposed in Figure 9.3), and the OR gate. We will use this implementation for comparisons of various synthesis methods which are proposed in this thesis. However, it is expected that a more compact and efficient transistor-level solution of the polymorphic multiplexer will be available in the future.

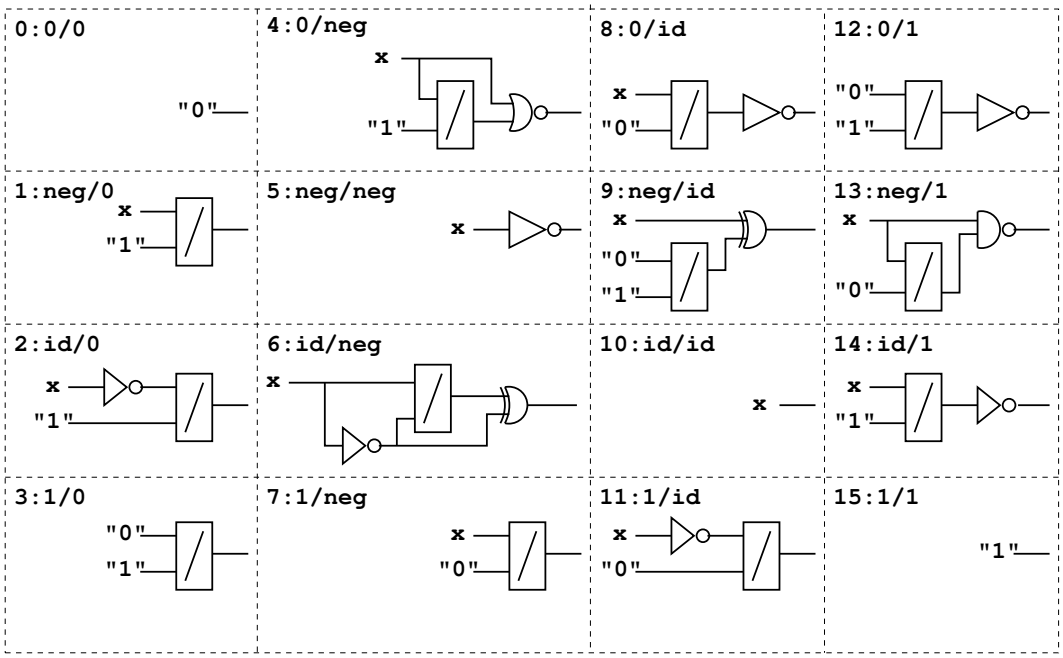


Figure 9.3: Conversion table for the transformation procedure of Polymorphic BDD into a polymorphic circuit

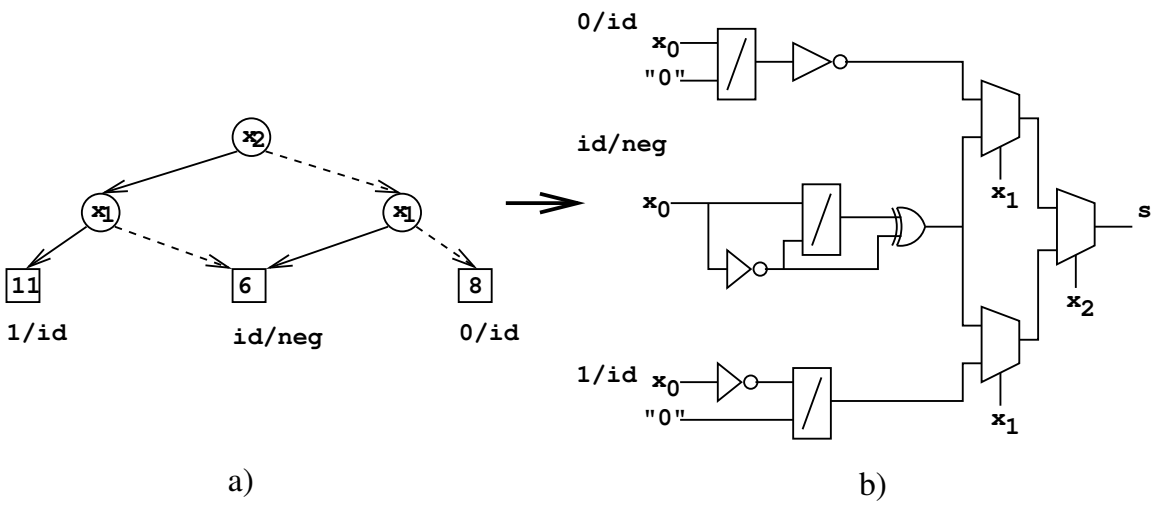


Figure 9.4: Reduced Polymorphic BDD (a) and the corresponding polymorphic circuit (b) for the three-bit Majority/Parity problem.

Polymorphic Multiplexing works as follows. Let us consider that a target polymorphic circuit has to implement the f_1 function together with the f_2 function. A conventional approach is used to synthesize the M_1 module implementing f_1 and another circuit, while the M_2 module is implementing f_2 independently. The output of the synthesized circuits are then multiplexed using polymorphic multiplexers as shown in Figure 9.6a. This approach will be denoted as Independent Modules (IM). Note that for especially smaller circuits it is also possible to use evolutionary circuit design instead of conventional methods to create the modules. Larger modules are usually designed by conventional design methods.

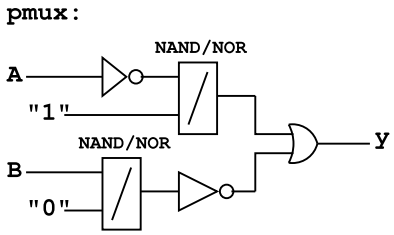
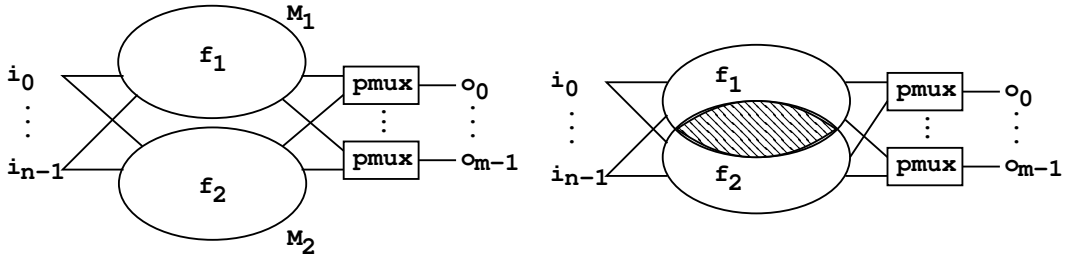


Figure 9.5: Polymorphic multiplexer at the gate level



a) *b)*

Figure 9.6: Multiplexing of conventional circuits by polymorphic multiplexers: a) independent modules, and b) sharing gates between modules.

In order to reduce the number of gates, the goal of synthesis can be to maximize the amount of gates that are shared by both circuits (see the intersection in Figure 9.6b). Espresso [5] and ABC-System[2] are conventional circuit synthesis methods that we have chosen to synthesize particular modules. We applied them with the aim of minimizing the number of gates in both modules and sharing as many gates as possible between the modules.

9.3 Benchmark Circuits

As no commonly-used benchmark set is available to evaluate proposed synthesis algorithms, we will introduce a new one. The proposed benchmark set consists of 14 circuits (see Table 9.2). This benchmark set can be divided into three subsets of problems: the Multiplier/Sorter circuits of the variable size; the Majority/Parity circuits of the variable size; and the polymorphic constant-coefficient multipliers (a subset denoted as xConstant/xConstant). An example of the constant-coefficient benchmark is the x67/x127 circuit which multiplies an input value by 67 in the first mode and by 127 in the second mode of polymorphic gates.

Combining a multiplier and binary sorter in the Multiplier/Sorter benchmark was chosen for several reasons. The multiplier benchmark is very popular in the (evolutionary) circuit design community and its synthesis is not trivial. The sorter accompanies the multiplier because it has also the same number of inputs and outputs. The Multiplier/Sorter circuits have been used as benchmarks in literature; however, this is only for up to eight inputs/eight outputs [79, 87, 18]. It should be noted that the area-optimal multipliers as well as sorters have significantly different structures for a given numbers of inputs.

Table 9.2: Proposed benchmarks

Benchmark Subset	Shortcut	Inputs	Outputs	*Mode 1*	*Mode 2*
Multiplier/Sorter	M/S4	4	4	2x2-bit multiplier	4-bit sorter
	M/S5	5	5	3x2-bit multiplier	5-bit sorter
	M/S6	6	6	3x3-bit multiplier	6-bit sorter
	M/S7	7	7	4x3-bit multiplier	7-bit sorter
	M/S8	8	8	4x4-bit multiplier	8-bit sorter
	M/S9	9	9	5x4-bit multiplier	9-bit sorter
Majority/Parity	M/P7	7	1	7-bit majority	7-bit parity
	M/P9	9	1	9-bit majority	9-bit parity
	M/P11	11	1	11-bit majority	11-bit parity
	M/P13	13	1	13-bit majority	13-bit parity
xConstant/xConstant	x67/x127	7	14	multiply by 67	multiply by 127
	x131/x251	8	16	multiply by 131	multiply by 251
	x257/x509	9	18	multiply by 257	multiply by 509
	x521/x1021	10	20	multiply by 521	multiply by 1021

Similarly, majority and parity circuits have the same number of inputs and completely different structures (utilized in the Majority/Parity benchmarks). The reason for including the xConstant/xConstant benchmarks was motivated by application [84].

9.4 Experimental Results

For all proposed benchmarks, we have designed polymorphic circuits using all three design methods: direct evolutionary design, Polymorphic-Binary-Decision-Diagram-based synthesis; and Polymorphic-Multiplexing-based synthesis.

9.4.1 Direct evolutionary design using CGP

For benchmark problems, 10 runs of CGP were performed in each experiment; the population size was 15, three genes were mutated in the design phase (if not specified elsewhere), seven genes were mutated on average in the optimization phase and up to 100 million generations were produced in each run.

The results for the Multiplier/Sorter benchmark are given in Table 9.3. The results are reported for the best setting of CGP parameters that have been found. Table 9.4 summarizes the results for the Majority/Parity benchmark. In this case, CGP can evolve Majority/Parity benchmark circuits with up to 13 inputs. The results for the xConstant/xConstant benchmark are summarized in Table 9.5.

9.4.2 Results of Polymorphic BDDs

Table 9.6 shows the results of the Polymorphic-BDD method for various benchmark problems. By 'Gates' we mean common two-input gates (the AND, OR, XOR, NAND, NOR gates), polymorphic two-input NAND/NOR gates, two-input multiplexers and one-input inverters. In Polymorphic-BDD design procedure, the reduction of redundant branches, redundant nonterminal (if-then-else) nodes with the same branches and reordering the inputs were applied.

Table 9.3: Parameters and results of CGP for the Multiplier/Sorter benchmark. Gates in the 'Gate set' row are numbered as: 1 – NAND/NOR, 2 – AND, 3 – OR, 4 – XOR, 5 – NAND, 6 – NOR, 7 – NOT A, 8 – NOT B, 9 – MOV A and 10 – MOV B, where MOV denotes the identity operation. The 'Mutated genes' label denotes the number of genes mutated in the design phase. The 'Generations avg.' label denotes the average number of generations when a correct circuit has been found.

Multiplier/Sorter	**2x2b/4b**	**3x2b/5b**	**3x3b/6b**	**4x3b/7b**
Input/outputs	4	5	6	7
$u \times v$	10×12	100×1	120×1	16×16
L-back	1	100	120	16
Mutated genes	1	2	4	4
Gate set	1, 2, 9, 10	1–4, 9, 10	1–10	1, 2, 9, 10
Successful runs	100%	100%	90%	30%
Generations (avg.)	52,580	854,900	26,972,648	62,617,151
Min. # of gates	23	30	52	113

Table 9.4: Parameters and results of CGP for the Majority/Parity benchmark. The gate set includes the NAND/NOR, AND, OR, XOR, NAND, NOR, NOT gates and identity operation.

Majority/Parity	**7b**	**9b**	**11b**	**13b**
Inputs	7	9	11	13
$u \times v$	80×1	120×1	120×1	160×1
L-back	80	120	120	160
Successful runs	100%	90%	50%	10%
Generations (avg.)	766,362	4,762,745	8,145,890	9,712,501
Min. # of gates	25	42	61	80

Table 9.5: Parameters and results of CGP for the xConstant/xConstant benchmark. The gate set includes the NAND/NOR, AND, OR, XOR, NAND, NOR, NOT gates and identity operation.

xConstant/xConstant	**x67/x127**	**x131/x251**	**x257/x509**
Inputs	7	8	9
Outputs	14	16	18
$u \times v$	320×1	640×1	320×1
L-back	320	640	320
Successful runs	60%	10%	30%
Generations (avg.)	7,192,359	46,833,855	40,002,719
Min. # of gates	94	239	116

9.4.3 Results of Polymorphic Multiplexing

We have used the Espresso and ABC-System synthesis tools to synthesize the modules of benchmark circuits with the aim of minimizing the number of gates and sharing as much gates as possible between the modules. Table 9.7 shows the results of the Espresso and ABC-System tools for the benchmark problems (see the rows labeled 'Espresso' and 'ABC'). One should note that multi-input gates produced by the Espresso method were

Table 9.6: Results of the Polymorphic-BDD design method for the Multiplier/Sorter, Majority/Parity and xConstant/xConstant benchmarks.

Multiplier/Sorter	**2x2/4b**	**3x2/5b**	**3x3/6b**	**4x3/7b**	**4x4/8b**	**5x4/9b**
Input/outputs	4	5	6	7	8	9
Nodes	19	37	79	135	253	412
Terminals	9	10	11	11	12	12
Gates	31	50	94	150	269	428
Majority/Parity	**7b**	**9b**	**11b**	**13b**		
Inputs	7	9	11	13		
Nodes	21	34	49	66		
Terminals	5	5	5	5		
Gates	31	41	59	73		
xConstant/xConstant	**x67/x127**	**x131/x251**	**x257/x509**	**x521/x1021**		
Inputs	7	8	9	10		
Outputs	14	16	18	20		
Nodes	205	407	326	882		
Terminals	16	16	15	16		
Gates	228	430	348	905		

converted to equivalent circuits composed of two-input gates.

We have also used the best-known optimized implementations of independent modules that we have interconnected by polymorphic multiplexers, that is, no sharing of gates between the modules were enabled. Results (denoted as 'IM') are given in Table 9.7 too. It can be seen that the strategy of independent modules leads to fewer gates; however, the circuits synthesized by the IM method are still large and have to be further optimized.

Table 9.7: Polymorphic-Multiplexing design results of the Multiplier/Sorter, Majority/Parity and xConstant/xConstant benchmarks.

Multiplier/Sorter	**2x2/4b**	**3x2/5b**	**3x3/6b**	**4x3/7b**	**4x4/8b**	**5x4/9b**
Inputs/outputs	4	5	6	7	8	9
Espresso	77	168	419	960	2309	—
ABC	37	61	119	198	359	679
IM	39	57	79	111	145	180
Majority/Parity	**7b**	**9b**	**11b**	**13b**		
Inputs	7	9	11	13		
Espresso	752	—	—	—		
ABC	39	58	79	112		
IM	33	49	63	83		
xConstant/xConstant	**x67/x127**	**x131/x251**	**x257/x509**	**x521/x1021**		
Inputs	7	8	9	10		
Outputs	14	16	18	20		
Espresso	—	—	—	—		
ABC	244	513	364	983		
IM	308	352	396	540		

9.5 Summary

The best implementations (in terms of the number of gates) obtained by the proposed methods are summarized for all benchmark circuits in Table 9.8. The best-achieved results are typed in bold. Although the direct polymorphic circuit evolution using CGP is not scalable, it can be considered as a very successful method for small problem instances. We applied only a basic set of operations to optimize Polymorphic BDDs. We expect that some improvements will be obtained by applying more advanced optimization techniques over BDDs. Results of Espresso were calculated only for some of the benchmark circuits because resulting circuits are too large and thus they are not competitive with other methods. The highest number of gates produced by Espresso is mainly due to the fact that many-input gates have to be transformed to circuits composed of two-input gates. Hence Espresso is considered as the weakest method for the polymorphic synthesis problem.

Table 9.8: Summary of the number of gates obtained by all methods for the benchmark problems. The set of gates includes the NAND/NOR polymorphic gate.

Circuit	**Ins.**	**Outs.**	**CGP**	**BDD**	**Esp**	**ABC**	**IM**
M/S4	4	4	**23**	31	77	37	39
M/S5	5	5	**30**	50	168	61	57
M/S6	6	6	**52**	94	419	119	79
M/S7	7	7	113	150	960	198	**111**
M/S8	8	8	—	269	2,309	359	**145**
M/S9	9	9	—	428	—	679	**180**
M/P7	7	1	**25**	31	752	39	33
M/P9	9	1	42	**41**	—	58	49
M/P11	11	1	61	**59**	—	79	63
M/P12	13	1	80	**73**	—	112	83
x67/x127	7	14	**94**	228	—	244	308
x131/x251	8	16	**239**	430	—	513	352
x257/x509	9	18	**116**	348	—	364	396
x521/x1021	10	20	—	905	—	983	**540**

Although we have considered only the NAND/NOR gate and polymorphic circuits with two modes, the proposed methods can easily be extended to utilize other polymorphic gates. In order to compare various settings, the methods have been used with other sets of gates: the ordinary gates with the NAND/XOR gate (see results in Table 9.9); and the ordinary gates with the NAND/NOR and NOR/NAND gate (see results in Table 9.10). Espresso was excluded from subsequent experiments due to its inefficiency in comparison to other synthesis methods.

It can be noticed that the results for Polymorphic Multiplexing (the 'ABC' and 'IM' designs) are the same in Tables 9.8 and 9.10. It is because the number of utilized gates of the polymorphic multiplexer is the same for both gate sets: ordinary gates with the NAND/NOR gate; and ordinary with the NAND/NOR and NOR/NAND gates. The NAND/NOR gate accompanied with the NOR/NAND gate does not give us an advantage in the lower number of utilized gates of the polymorphic multiplexer. On the other hand, the utilization of both polymorphic gates gives us smaller circuit designs in other

Table 9.9: Summary of the number of gates obtained by the methods for the benchmark problems. The set of gates includes the NAND/XOR polymorphic gate.

Circuit	Ins.	Outs.	CGP	BDD	ABC	IM
M/S4	4	4	**22**	31	33	35
M/S5	5	5	**35**	50	56	52
M/S6	6	6	**60**	94	113	73
M/S7	7	7	**94**	150	191	104
M/S8	8	8	—	269	351	**137**
M/S9	9	9	—	428	670	**171**
M/P7	7	1	**23**	32	38	32
M/P9	9	1	**34**	42	57	48
M/P11	11	1	**56**	60	78	62
M/P13	13	1	—	**74**	111	82
x67/x127	7	14	**109**	229	230	294
x131/x251	8	16	**214**	431	497	336
x257/x509	9	18	**98**	349	346	378
x521/x1021	10	20	—	906	963	**520**

Table 9.10: Summary of the number of gates obtained by the methods for the benchmark problems. The set of gates includes the NAND/NOR and NOR/NAND polymorphic gates.

Circuit	Ins.	Outs.	CGP	BDD	ABC	IM
M/S4	4	4	**17**	28	37	39
M/S5	5	5	**33**	47	61	57
M/S6	6	6	**57**	91	119	79
M/S7	7	7	**101**	147	198	111
M/S8	8	8	—	226	359	**145**
M/S9	9	9	—	425	679	**180**
M/P7	7	1	**22**	27	39	33
M/P9	9	1	**38**	39	58	49
M/P11	11	1	**55**	**55**	79	63
M/P13	13	1	—	**71**	112	83
x67/x127	7	7	**100**	222	244	308
x131/x251	8	8	**215**	424	513	352
x257/x509	9	9	**109**	342	366	396
x521/x1021	10	10	—	899	983	**540**

cases.

We can see from the summaries (Tables 9.8, 9.9 and 9.10) that there is no single method which outperforms the other methods.

The circuits synthesized using the aid of conventional methods are still large and they have to be further optimized.

Chapter 10

Evolutionary Optimization of Polymorphic Circuits

Polymorphic circuits created using the methods based on Polymorphic Multiplexing or Polymorphic Binary Decision Diagrams are quite large in many cases and thus are inefficient. In the case of Polymorphic Multiplexing, polymorphic gates are solely located at the outputs of a circuit. In the case of Polymorphic BDDs, the polymorphic gates are located close to the inputs of a circuit. In order to minimize the total number of gates, it is desirable to integrate polymorphic gates deeply into the circuit structure and also increase the ratio of polymorphic gates to ordinary gates. We will show later in our experiments that increasing the ratio of polymorphic gates usually leads to decreasing the total number of gates.

10.1 Evolutionary Optimization Algorithm

The evolutionary algorithm introduced in Section 9.2.1 was taken as the basis for evolutionary optimization of polymorphic circuits. For purposes of optimization, polymorphic circuits are converted into the CGP representation. Since presented CGP uses two-input nodes, all components have to utilize up to two inputs. Therefore, the three-input multiplexer is converted into four two-input gates (in the BDD-based design), the three-input AND-gate is converted into two two-input gates (in the Espresso-based designs), etc.

As we know from Chapter 7 the modified selection strategy is beneficial in the optimization phase. As the strategies are defined in Section 6.3, they do not have to be further modified to handle the polymorphic ability. However, we will denote these strategies utilized for the polymorphic circuit optimization as follows: the traditional strategy, as the *SeS-1' strategy*, and the modified strategy as the *SeS-2' strategy*.

Acceleration of optimization

For acceleration of optimization, the parallel simulation and short-circuit evaluation are utilized. The parallel evaluation was described earlier in Sections 6.4 and 9.2.1. One must remember that it has to be run for each mode of polymorphic gates separately. The short-circuit evaluation (see Section 6.4) extended for polymorphic circuits takes place even for small circuits. A candidate circuit is evaluated for both modes of gates separately as mentioned earlier. When a candidate circuit is recognized to be incorrect for the first mode of polymorphic gates, then the evaluation of the circuit is terminated.

10.2 Experimental Setup

We have focused on two areas in experiments: gate-level minimization using proposed selection strategies and reduction of computational overhead using short-circuit evaluation.

The effect of the short-circuit evaluation will be measured using the proposed evolutionary algorithm (Section 10.1). The algorithm is used with a population size of 15 individuals and the $n_c \times 1$ topology, where n_c is the number of elements in the seed which comes from the polymorphic-multiplexing design method based on the ABC-System implementations. The mutation operator modifies seven genes on average. Each experiment is performed ten times. The number of generations depends on the complexity of a benchmark problem and is given for particular benchmarks in Table 10.2. Results will be given for the parallel simulation on the 64-bit architecture.

CGP is, in fact, seeded with a fully functional but non-optimal design with respect to the number of gates. The goal of optimization is to reduce the number of gates. We will compare the traditional SeS-1' selection strategy which explicitly counts the number of gates with the unconventional SeS-2' selection strategy.

We will show in Section 10.3 that the unconventional SeS-2' selection strategy significantly improves the results of optimization, although it does not take into account the number of gates explicitly.

10.3 Experimental Results

10.3.1 Short-circuit evaluation

Table 10.1 gives the computational effort of optimization for CGP seeded using ABC-based designs. The 'Reduction' column of Table 10.1 shows the reduction of the computational time which is achievable when candidate circuits are evaluated with the short-circuit evaluation (see Sections 6.4 and 10.1).

Figure 10.1 shows graphically the computational reduction from Table 10.1. We can see that the curves of the selection strategies are close to each other in comparison to the curves shown in Figure 7.1. The reason is probably that the evolutionary-optimization algorithm of polymorphic circuit designs has not utilized the training-set reorganization (see Section 6.5) in this case. On the other hand, this feature does not influence the quality of optimized circuits, but instead speeds up the evaluation.

10.3.2 Circuit optimization

Table 10.2 shows the results obtained by applying evolutionary optimization on the benchmark circuits created by the polymorphic-BDD-based synthesis. For both selection strategies, we compared the number of gates (best, worst, mean) and the average number of polymorphic gates. We can observe that SeS-1' and SeS-2' give us very different results.

Similarly, Table 10.3 shows the results when CGP was seeded by the polymorphic-multiplexing synthesis with support of the ABC synthesis tool. We can observe that the best solutions are significantly different too.

In a similar way, Table 10.4 shows the results when CGP was initialized by the Espresso-based synthesis. Not all benchmarks were optimized (even designed) because we reached the limits of the Espresso algorithm implementation. Differences between the selection strategies are significant.

Table 10.1: The mean number of evaluations. 'Tot. Evals.' denotes the total number of evaluations when the short-circuit evaluation is not involved.

Benchmark	**Ins.**	**Outs.**	n_c	**Strategy**	**Tot. Evals.** $[10^6]$	**Evaluations** $[10^6]$	**Reduction**
M/S4	4	4	45	SeS-1'	30	17.6	1.70
				SeS-2'		17.5	1.71
M/S5	5	5	71	SeS-1'	300	172.7	1.74
				SeS-2'		172.1	1.74
M/S6	6	6	131	SeS-1'	300	170.5	1.76
				SeS-2'		170.8	1.76
M/S7	7	7	212	SeS-1'	6,000	2,128.8	2.82
				SeS-2'		2,216.7	2.71
M/S8	8	8	375	SeS-1'	12,000	2,896.0	4.14
				SeS-2'		3,040.2	3.95
M/S9	9	9	697	SeS-1'	24,000	4,740.5	5.06
				SeS-2'		4,872.0	4.93
M/P7	7	1	41	SeS-1'	60	19.6	3.06
				SeS-2'		19.6	3.06
M/P9	9	1	60	SeS-1'	240	36.3	6.61
				SeS-2'		36.8	6.52
M/P11	11	1	81	SeS-1'	9,600	1,037.3	9.25
				SeS-2'		1,024.1	9.37
M/P13	13	1	114	SeS-1'	38,400	3,729.2	10.30
				SeS-2'		3,851.2	9.97
x67/x127	7	14	274	SeS-1'	600	214.5	2.80
				SeS-2'		223.7	2.68
x131/x251	8	16	547	SeS-1'	1,200	340.2	3.53
				SeS-2'		341.2	3.52
x257/x509	9	18	410	SeS-1'	2,400	672.1	3.57
				SeS-2'		647.3	3.71
x521/x1021	10	20	1028	SeS-1'	4,800	906.5	5.29
				SeS-2'		895.6	5.36

Finally, Table 10.5 gives the results of evolutionary optimization initialized with the polymorphic-multiplexing-based synthesis which have been composed of independent modules.

Presented results indicate that the SeS-2' strategy clearly outperforms the SeS-1' strategy. This phenomenon was also observed during the optimization of conventional circuits (see Section 7.4). For example, see Figure 10.2, which demonstrates a typical run of CGP, and Figure 10.3, which demonstrates the average behavior of CGP on the particular benchmark.

10.4 Summary

In comparison with conventional synthesis, proposed methods require significantly more computational time. The reason is that the synthesis problem of a polymorphic circuit

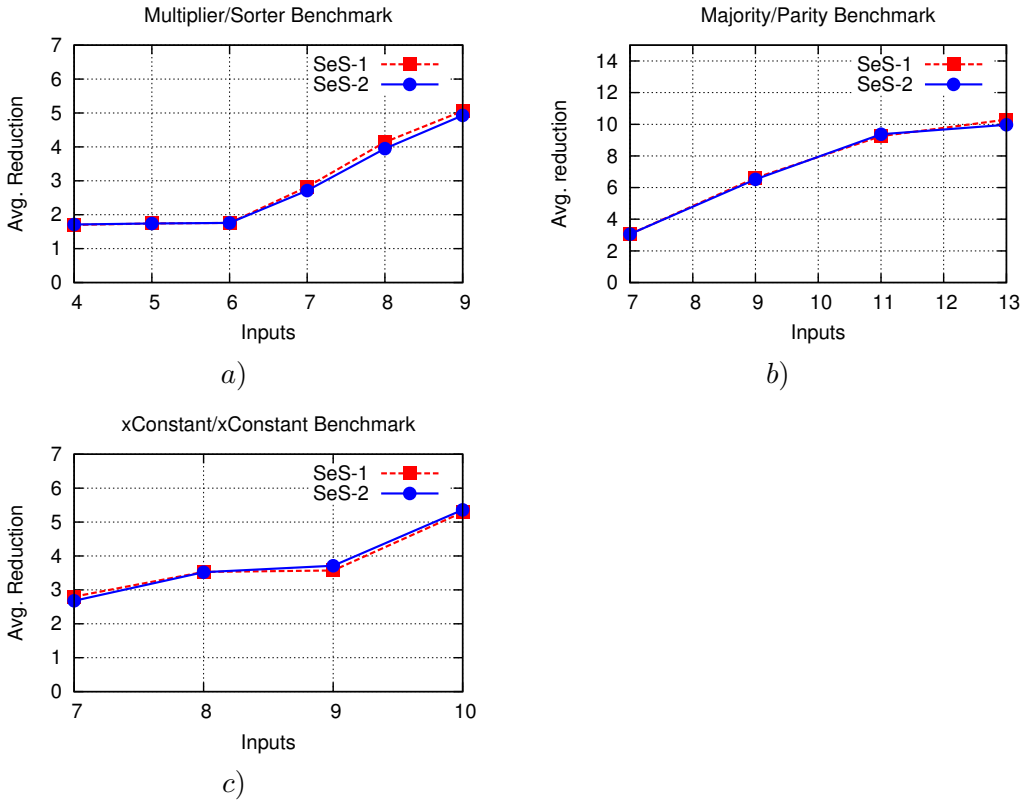


Figure 10.1: Average reduction of the computational time in comparison with the complete truth table evaluation for: a) Multiplier/Sorter benchmark; b) Majority/Parity benchmark; and c) xConstant/xConstant benchmark.

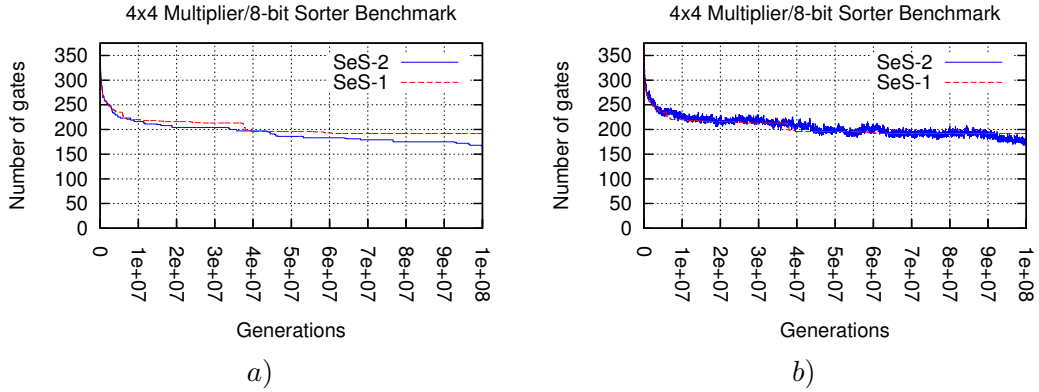


Figure 10.2: The number of gates of: a) the best-obtained individual, β, and b) parent individual, ρ (taken from the best runs of the 4x4-bit Multiplier/8-bit Sorter benchmark which were initialized by the ABC designs).

is more difficult than the conventional synthesis and also the main role is played by the evolutionary search which is time consuming. The computation time of the direct CGP evolution can be expressed as follows: it takes 80 seconds to generate one million generations

Table 10.2: Results of the evolutionary optimization initialized by the polymorphic-BDD-based synthesis. The 'GpR' label denotes generations per run. The 'PolyG' label denotes the average ratio of polymorphic gates in optimized circuits.

Benchmark	**Ins.**	**Outs.**	n_c	**GpR** $[10^6]$	**Strategy**	**Best**	**Worst**	**Mean**	**PolyG**
M/S4	4	4	98	1	SeS-1'	27	33	30.0	11%
					SeS-2'	20	24	21.7	28%
M/S5	5	5	173	10	SeS-1'	46	56	52.0	8%
					SeS-2'	35	56	39.4	26%
M/S6	6	6	345	10	SeS-1'	103	129	116.5	4%
					SeS-2'	81	97	87.0	17%
M/S7	7	7	569	100	SeS-1'	174	194	184.7	6%
					SeS-2'	112	137	125.8	19%
M/S8	8	8	1,043	100	SeS-1'	355	407	385.7	8%
					SeS-2'	256	289	274.5	13%
M/S9	9	9	1,867	100	SeS-1'	530	572	553.1	30%
					SeS-2'	471	525	503.0	27%
M/P7	7	1	98	1	SeS-1'	31	44	39.9	15%
					SeS-2'	27	36	32.3	17%
M/P9	9	1	148	1	SeS-1'	49	79	65.4	8%
					SeS-2'	44	64	55.5	12%
M/P11	11	1	210	10	SeS-1'	71	97	86.2	11%
					SeS-2'	49	82	59.2	17%
M/P13	13	1	276	10	SeS-1'	97	138	113.2	7%
					SeS-2'	70	109	88.0	13%
x67/x127	7	14	860	10	SeS-1'	218	262	234.9	3%
					SeS-2'	141	188	176.2	15%
x131/x251	8	16	1,668	20	SeS-1'	560	607	583.1	2%
					SeS-2'	535	622	570.7	9%
x257/x509	9	18	1,343	50	SeS-1'	286	369	331.9	5%
					SeS-2'	194	237	218.3	16%
x521/x1021	10	20	3,568	20	SeS-1'	1,289	1,381	1,334.2	1%
					SeS-2'	1,407	1,554	1,481.5	6%

for the four-input polymorphic circuit on the Athlon64 3200+ processor. For a seven-input polymorphic circuit, 207 seconds are needed for the same processor.

Table 10.6 (see the 'Design time' label) gives the average time required to construct the 4x4-bit multiplier/8-bit sorter by proposed methods on the Athlon64 X2 4800+ processor. Except for the Polymorphic-BDD synthesis (when brute force was used for the optimal ordering of input variables), the design time is quite reasonable. It is almost zero if independent modules are available in advance. The main bottleneck is the optimization time imposed by CGP which is in the order of hours for this circuit. However, this time can be reduced by decreasing the number of generations if a slightly larger solution is acceptable. Figures 10.2 and 10.3 demonstrate the relation between the number of gates and the

Table 10.3: Results of evolutionary optimization initialized by the ABC-based synthesis. The 'GpR' label denotes generations per run. The 'PolyG' label denotes the average ratio of polymorphic gates.

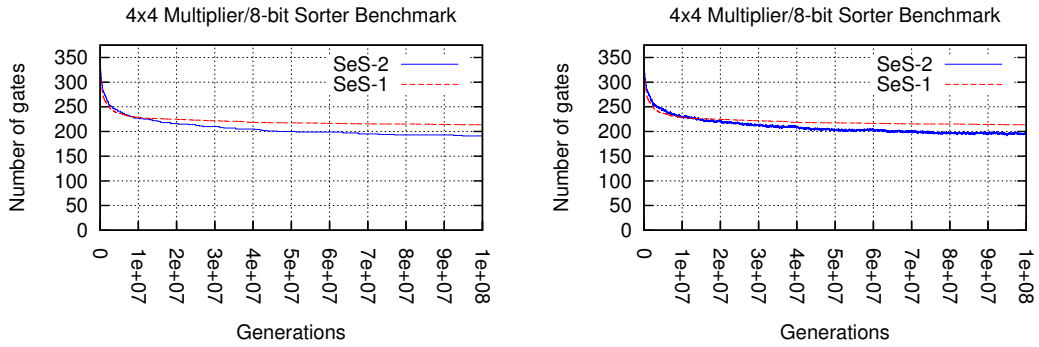
Benchmark	**Ins.**	**Outs.**	n_c	**GpR** [10^6]	**Strategy**	**Best**	**Worst**	**Mean**	**PolyG**
M/S4	4	4	45	1	SeS-1'	19	22	20.2	37%
					SeS-2'	18	21	19.3	36%
M/S5	5	5	71	10	SeS-1'	36	43	39.0	27%
					SeS-2'	32	37	34.8	28%
M/S6	6	6	131	10	SeS-1'	71	85	77.2	21%
					SeS-2'	59	72	66.2	24%
M/S7	7	7	212	100	SeS-1'	110	135	121.4	25%
					SeS-2'	88	107	97.1	24%
M/S8	8	8	375	100	SeS-1'	192	227	213.6	22%
					SeS-2'	166	213	181.7	22%
M/S9	9	9	697	100	SeS-1'	366	411	384.6	23%
					SeS-2'	323	378	348.3	21%
M/P7	7	1	41	1	SeS-1'	29	33	31.3	17%
					SeS-2'	21	33	29.2	20%
M/P9	9	1	60	1	SeS-1'	45	53	48.6	20%
					SeS-2'	42	50	45.5	17%
M/P11	11	1	81	10	SeS-1'	66	72	70.9	17%
					SeS-2'	62	71	68.3	18%
M/P13	13	1	114	10	SeS-1'	88	95	91.7	18%
					SeS-2'	80	95	85.6	21%
x67/x127	7	14	274	10	SeS-1'	111	143	130.2	28%
					SeS-2'	85	114	96.4	25%
x131/x251	8	16	547	20	SeS-1'	209	241	220.7	25%
					SeS-2'	187	222	202.3	22%
x257/x509	9	18	410	50	SeS-1'	133	162	146.3	26%
					SeS-2'	112	130	122.4	25%
x521/x1021	10	20	1,028	20	SeS-1'	346	429	380.7	22%
					SeS-2'	326	380	359.1	20%

generation (i.e. the time spent by optimization).

The best implementations (in terms of the number of gates for the gate set which includes the NAND/NOR gate) obtained by proposed methods and subsequent CGP optimization are summarized for all benchmark circuits in Table 10.7. The best-achieved results are typed in bold. The Polymorphic-Multiplexing-based synthesis using independent modules or the ABC-System tool followed by the CGP optimization provide the best results for larger circuits. The optimization of the Polymorphic-BDD synthesis does not perform as well as the optimization of the Polymorphic-Multiplexing-based synthesis. Table 10.7 also shows that no single method outperforms other methods in all problem instances. By using the proposed methods, we were able to significantly improve the gate utilization of

Table 10.4: Results of evolutionary optimization initialized by the Espresso-based synthesis. The 'GpR' label denotes generations per run. The 'PolyG' label denotes the average ratio of polymorphic gates.

Benchmark	**Ins.**	**Outs.**	n_c	**GpR** $[10^6]$	**Strategy**	**Best**	**Worst**	**Mean**	**PolyG**
M/S4	4	4	92	1	SeS-1'	22	29	25.8	21%
					SeS-2'	20	24	21.6	31%
M/S5	5	5	183	10	SeS-1'	49	59	52.8	17%
					SeS-2'	30	40	36.4	25%
M/S6	6	6	436	10	SeS-1'	123	149	132.0	8%
					SeS-2'	75	104	84.4	24%
M/S7	7	7	979	100	SeS-1'	239	274	258.4	6%
					SeS-2'	116	157	135.6	21%
M/S8	8	8	2,330	100	SeS-1'	616	697	646.3	3%
					SeS-2'	318	404	375.3	14%
M/P7	7	1	755	1	SeS-1'	230	272	246.7	3%
					SeS-2'	44	184	90.3	12%



a) *b)*

Figure 10.3: The mean number of gates of: a) the best-obtained individuals, β, and b) parent individuals, ρ (taken from 10 runs of the 4x4-bit Multiplier/8-bit Sorter benchmark which were initialized by the ABC designs).

almost all benchmark circuits in comparison with the results existing in the present literature ('B.L.' column in Table 10.7). The proposed SeS-2' strategy significantly improves the search efficiency.

In most cases, combining the BDD design with evolutionary optimization reduces the number of gates. However, in some cases, evolutionary optimization can even increase the number of gates. For example, see the M/P9 benchmark in Table 10.7 in which the BDD design (without evolutionary optimization) leads to 41 gates while the subsequent optimization requires 44 gates. The reason for this is that the applied BDD synthesis uses multiplexers, and these multiplexers have to be transformed into the two-input-gates design where a single multiplexer is transformed on four two-input gates for CGP.

Table 10.5: Results of evolutionary optimization initialized with by the IM-based synthesis. The 'GpR' label denotes generations per run. The 'PolyG' label denotes the average ratio of polymorphic gates.

Benchmark	**Ins.**	**Outs.**	n_c	**GpR** $[10^6]$	**Strategy**	**Best**	**Worst**	**Mean**	**PolyG**
M/S4	4	4	47	1	SeS-1'	22	25	23.0	29%
					SeS-2'	19	23	20.6	27%
M/S5	5	5	67	10	SeS-1'	39	41	39.3	26%
					SeS-2'	34	38	35.9	29%
M/S6	6	6	91	10	SeS-1'	59	62	61.1	23%
					SeS-2'	58	60	59.3	25%
M/S7	7	7	125	100	SeS-1'	83	87	84.6	21%
					SeS-2'	80	83	81.4	26%
M/S8	8	8	161	100	SeS-1'	109	112	110.8	23%
					SeS-2'	105	108	106.9	21%
M/S9	9	9	198	100	SeS-1'	150	153	151.6	17%
					SeS-2'	148	152	149.8	17%
M/P7	7	1	35	1	SeS-1'	30	32	31.1	12%
					SeS-2'	27	32	30.5	11%
M/P9	9	1	51	1	SeS-1'	42	47	43.5	10%
					SeS-2'	41	46	42.9	8%
M/P11	11	1	65	10	SeS-1'	54	61	59.9	7%
					SeS-2'	51	61	57.3	9%
M/P13	13	1	85	10	SeS-1'	66	76	71.0	7%
					SeS-2'	65	74	68.1	7%
x67/x127	7	14	350	10	SeS-1'	137	197	160.1	16%
					SeS-2'	114	171	148.5	18%
x131/x251	8	16	400	20	SeS-1'	264	271	267.6	14%
					SeS-2'	259	272	267.2	13%
x257/x509	9	18	450	50	SeS-1'	337	346	342.4	13%
					SeS-2'	340	347	342.7	13%
x521/x1021	10	20	600	20	SeS-1'	484	497	490.7	10%
					SeS-2'	488	495	492.2	10%

Table 10.6: Design time and optimization time for the 4x4-bit multiplier/8-bit sorter initialized by the proposed methods.

Method	**BDD**	**Espresso**	**ABC**	**IM**
Elements (n_c)	1043	2330	375	161
Generations $[10^6]$	100	100	100	100
Design time [s]	345	1	3	0.001
Optimization time [s]	154,015	361,394	21,176	9,261

Table 10.7: The minimum number of gates obtained by all methods for the benchmark problems. The last two columns ('B.L.' and 'Ref.') give the best known results from the present literature. The set of gates includes NAND/NOR polymorphic gate. The 'o' suffix of labels located in the right half of the table means that the CGP-based optimization was applied.

Circuit	Ins.	Outs.	CGP	BDD	Esp	ABC	IM	BDDo	Espo	ABCo	IMo	B.L.	Ref.
M/S4	4	4	23	31	77	37	39	20	20	**18**	19	23	[87]
M/S5	5	5	**30**	50	168	61	57	35	30	32	34	30	[87]
M/S6	6	6	**52**	94	419	119	79	81	75	59	58	52	[87]
M/S7	7	7	113	150	960	198	111	112	116	88	**80**	110	[18]
M/S8	8	8	—	269	2,309	359	145	256	318	166	**105**	205	[18]
M/S9	9	9	—	428	—	679	180	471	—	323	**148**	—	—
M/P7	7	1	25	31	752	39	33	27	44	**21**	27	29	[18]
M/P9	9	1	42	**41**	—	58	49	44	—	42	**41**	45	[18]
M/P11	11	1	61	59	—	79	63	**49**	—	62	51	69	[18]
M/P12	13	1	80	73	—	112	83	70	—	80	**65**	90	[18]
x67/x127	7	14	94	228	—	244	308	141	—	**85**	114	—	—
x131/x251	8	16	239	430	—	513	352	535	—	**187**	259	—	—
x257/x509	9	18	116	348	—	364	396	194	—	**112**	337	—	—
x521/x1021	10	20	—	905	—	983	540	1,289	—	**326**	484	—	—

Another important property of optimized circuit designs initialized by Polymorphic Multiplexing is that the percentage share of polymorphic gates utilized in resulting circuits is relatively high. Tables 10.2, 10.3, 10.4 and 10.5 show that the resulting circuits optimized from the ABC-based synthesis contain 22% polymorphic gates on average (23% was achieved for the IM-based synthesis), while he BDD-based synthesis only led to 8% and the Espresso-based synthesis to 3% on average.

From these results, we can observe that the new SeS-2' strategy significantly outperforms the standard SeS-1' strategy in most cases. This is a very interesting phenomenon that has been investigated for the conventional-circuit evolution proposed in Chapter 7.

We have also optimized circuit designs which support other polymorphic gates. Table 10.8 (and Table 10.9 respectively) shows the results using a gate set which includes the polymorphic NAND/XOR gates (and the NAND/NOR and NOR/NAND gates respectively). We summarized all results in Table 10.10 which shows the gate-level cost of the best-known implementations discovered so far.

We can observe that utilization of the NAND/XOR gate is especially useful for the Majority/Parity benchmarks which is natural because of the XOR-based parity construction. Including the NOR/NAND gate allowed us to reduce the number of gates in comparison with the original gate set. However, it still remains unclear how to select a suitable set for a particular polymorphic circuit benchmark.

Table 10.8: The minimum number of gates obtained by all methods for the benchmark problems. The set of gates includes the NAND/XOR polymorphic gate. The 'o' suffix of labels in the right half of the table means that the CGP-based optimization was applied.

Circuit	**Ins.**	**Outs.**	**CGP**	**BDD**	**ABC**	**IM**	**BDDo**	**ABCo**	**IMo**
M/S4	4	4	22	31	33	35	18	18	**17**
M/S5	5	5	35	50	56	52	36	**30**	32
M/S6	6	6	60	94	113	73	70	**56**	59
M/S7	7	7	94	150	191	104	102	**80**	81
M/S8	8	8	—	269	351	137	221	165	**108**
M/S9	9	9	—	428	670	171	413	295	**146**
M/P7	7	1	23	32	38	32	23	**21**	22
M/P9	9	1	**34**	42	57	48	37	39	43
M/P11	11	1	56	60	78	62	**39**	61	57
M/P13	13	1	—	74	111	82	**60**	81	68
x67/x127	7	14	109	229	230	294	143	**84**	104
x131/x251	8	16	214	431	497	336	410	**181**	250
x257/x509	9	18	**98**	349	346	378	176	115	325
x521/x1021	10	20	—	906	963	520	1171	**307**	475

Table 10.9: The minimum number of gates obtained by all methods for the benchmark problems. The set of gates includes NAND/NOR and NOR/NAND polymorphic gates. The 'o' suffix of labels in the right half of the table means that the CGP-based optimization was applied.

Circuit	**Ins.**	**Outs.**	**CGP**	**BDD**	**ABC**	**IM**	**BDDo**	**ABCo**	**IMo**
M/S4	4	4	17	28	37	39	**16**	**16**	**16**
M/S5	5	5	33	47	61	57	31	**30**	31
M/S6	6	6	57	91	119	79	64	**55**	**55**
M/S7	7	7	101	147	198	111	97	85	**77**
M/S8	8	8	—	226	359	145	236	155	**102**
M/S9	9	9	—	425	679	180	403	293	**141**
M/P7	7	1	**22**	27	39	33	24	24	24
M/P9	9	1	**38**	39	58	49	**38**	**38**	39
M/P11	11	1	55	55	79	63	**42**	54	57
M/P13	13	1	—	71	112	83	69	75	**63**
x67/x127	7	14	100	222	244	308	126	**75**	104
x131/x251	8	16	215	424	513	352	368	**168**	244
x257/x509	9	18	109	342	366	396	172	**108**	310
x521/x1021	10	20	—	899	983	540	1054	**283**	476

Table 10.10: The minimum number of gates achieved by the proposed methods for the benchmark problems. In addition to polymorphic gates, the gate sets include the AND, OR, NAND, NOR, XOR and NOT gate.

Benchmark	**Ins.**	**Outs.**	{NAND/NOR} $\cup\Gamma^{(1)}$	{NAND/XOR} $\cup\Gamma^{(1)}$	{NAND/NOR, NOR/NAND} $\cup\Gamma^{(1)}$
M/S4	4	4	18	17	**16**
M/S5	5	5	**30**	**30**	**30**
M/S6	6	6	**52**	56	55
M/S7	7	7	80	80	**77**
M/S8	8	8	105	108	**102**
M/S9	9	9	148	146	**141**
M/P7	7	1	**21**	**21**	22
M/P9	9	1	41	**34**	38
M/P11	11	1	49	**39**	42
M/P13	13	1	65	**60**	63
x67/x127	7	14	85	84	**75**
x131/x251	8	16	187	181	**168**
x257/x509	9	18	112	**98**	108
x521/x1021	10	20	326	307	**283**

Chapter 11

Conclusions

Evolutionary synthesis and optimization of ordinary and polymorphic circuits are research topics where unexplored open areas still exist. Polymorphic electronics gave us opportunity to enhance conventional digital electronics in several application domains. We have employed or modified various well-known approaches, including Cartesian Genetic Programming and Binary Decision Diagrams to improve the synthesis and optimization process of reasonably large circuits.

Research results have been published in the proceedings of well-recognized international conferences, including the NASA/ESA Conference on Adaptive Hardware and Systems (AHS), IEEE Congress on Evolutionary Computation (CEC), International Conference on Evolvable Systems (ICES) or Genetic and Evolutionary Computation Conference (GECCO). Our research was also published in the International Journal of Unconventional Computing. A summarized description of the proposed design and optimization methods for polymorphic circuits was accepted for publication in the Journal of Multiple-Valued Logic and Soft Computing. However, there are still new and original results in this thesis which have not been published elsewhere.

11.1 Contributions

This section summarizes the main contributions of this thesis.

We have employed Cartesian Genetic Programming (CGP) and its modifications in the ordinary-circuit optimization and improved existing results of combinational circuit synthesis. It was shown that the novel selection strategy can provide more compact circuits. We have verified that the selection of the parent individual in Cartesian Genetic Programming on the basis of its functionality, instead of compactness, leads to smaller phenotypes at the end of the evolution. The new selection strategy is especially useful for the optimization of nontrivial circuits if sufficient redundancy is available in terms of available gates and sufficient time is allowed for the evolution. It is curious that the strategy does not explicitly prefer a smaller number of gates in a parent individual selection and after everything, it is still able to minimize. This phenomenon has been confirmed using experiments performed on common benchmark circuits. We have proposed evaluation enhancements, such as the short-circuit evaluation or train-set reordering, that speed up computational time of the candidate circuit evaluation. Results of the ordinary-circuit optimization have been partially published in the proceedings of the GECCO and ICES conferences [20, 19].

We have utilized specific polymorphic gates in ordinary circuits to minimize the total

number of gates. We have used the fitness function with priorities to reach this goal. We have clearly demonstrated that the evolutionary design of digital circuits which is conducted at the gate level is able to produce human-competitive circuits at the transistor-level. We have discovered novel implementations of adders and majority circuits that utilize the polymorphic NAND/NOR and NOR/NAND gates. The experimental results show that the concept of Cartesian Genetic Programming is powerful to solve unusual problems of digital synthesis. A paper on this topic has been published in the proceedings of the GECCO conference [17].

We have proposed and investigated new approaches to polymorphic-circuit design which are based on polymorphic multiplexing and polymorphic Binary Decision Diagrams. We have compared these new approaches with the direct evolution using CGP. Since there is not any benchmark set available, we have proposed a new set of benchmark problems to compare the approaches. The new proposed approaches can generate candidate solutions with an arbitrary size (if they are available by a particular SW implementation, such as Espresso or the ABC-System tool) in a reasonable time. As we have mentioned earlier, there has not been any reasonable method for the synthesis of nontrivial polymorphic circuits. The original CGP can evolve compact solutions, but only for small problems. The papers on polymorphic-circuit design have been published in the proceedings of the AHS and CEC conferences [86, 83, 18] and also in an international journal [87].

We have employed the extended CGP to optimize the synthesized polymorphic circuits at the gate level. In order to reduce the number of gates to a reasonable amount, a time consuming optimization has to be conducted. However, the evolutionary-based optimization procedure seems to be the only way on how to obtain reasonable implementations of polymorphic circuits nowadays. The results obtained by these proposed methods represent the best-known solutions if the number of gates is considered as the target criterion. A paper on the evolutionary optimization of polymorphic circuits has been published in the proceedings of the CEC conference [18]. The evolved constant-coefficient multipliers were utilized in the polymorphic FIR filter [84]. We have also physically demonstrated the behavior of some small polymorphic circuits in the REPOMO32 chip [85].

11.2 Possibilities of Future Research

We have been exploring several research areas in this thesis, but there are still possibilities to extend the research topics. Some of them are briefly given.

Scalability problem of the polymorphic circuit synthesis is still an important topic. In order to deal with this problem better, the more advanced adaptation of the ordinary circuit synthesis is needed. This adaptation has to better utilize merits of polymorphic gates.

The evaluation time is the main bottleneck of the evolutionary optimization using CGP, so the new approach to reduce the computational time is needed. The adaptation of the SAT resolving approach proposed by Vašíček and Sekanina in recent times should significantly speed up the evaluation time of large polymorphic benchmark circuits.

In proposed experiments, the polymorphic gates are utilized in circuits of ordinary functionality in order to decrease the number of utilized transistors. Only the NAND/NOR and NOR/NAND gates controlled by an external signal were utilized. Future research will be devoted to searching for those other unconventional components[1] which were overlooked in the past, but which could serve as area-efficient building blocks for the evolutionary

[1] Unconventional components could be other polymorphic gates as well.

design conducted at the gate level.

All proposed methods are based on a methodology which assumes that the logic functions are fully-defined through the fully-defined truth tables. The extended methods will take "don't care values" into account, so that only some input or output combinations have to be defined.

The new proposed selection strategy is focused on the number of gates in implementations. The extended approach will take the number of transistors into account. The approach will not extend a fitness function but a mutation operator. As we assumed, the proposed selection strategy is influenced by the mutation operator, which in fact, does the selection. The current operator changes (mutates) all genes of a chromosome with the same probability. The extended operator will change the genes, which represent larger gates, more frequently.

Bibliography

[1] T. Aoki, N. Homma, and T. Higuchi. Evolutionary synthesis of arithmetic circuit structures. *Artificial Intelligence Review*, 20(3–4):199–232, 2003.

[2] Berkley Logic Synthesis and Verification Group. *ABC: A System for Sequential Synthesis and Verification*. http://www.eecs.berkeley.edu/~ alanmi/abc/, 2010.

[3] M. Bidlo. Evolutionary design of generic combinational multipliers using development. In *Evolvable Systems: From Biology to Hardware*, volume 2007 of *LNCS*, pages 77–88, 2007.

[4] M. Bidlo. *Evolutionary Design of Generic Structures Using Instruction-Based Development*. PhD thesis, 2009.

[5] R. K. Brayton, C. McMullen, G. D. Hatchel, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, Boston, Massachusetts, USA, 1984.

[6] R. Brooks. The relationship between matter and life. *Nature*, 409(6816):409–411, 2001.

[7] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transaction on Computers*, 35(8):677–691, 1986.

[8] D. Chai, J.-H. Jiang, Y. Jiang, Y. Li, A. Mishchenko, and R. Brayton. *MVSIS 2.0 User's Manual*. http://embedded.eecs.berkeley.edu/mvsis/doc/mvsis_20_manual.pdf, 2010.

[9] D. Chen, T. Aoki, N. Homma, T. Terasaki, and T. Higuchi. Graph-based evolutionary design of arithmetic circuits. *IEEE Trans. on Evolutionary Computing*, 6(1):86–100, 2002.

[10] K.-H. Cheng and V.-C. Hsieh. High efficient 3-input XOR for low-voltage low-power high speed applications. In *Proc. of The First IEEE Asic Pacific Conference on ASICs*, pages 166–169, Seoul, Korea, 1999. IEEE Computer Society.

[11] M. Collins. Finding needles in haystacks is harder with neutrality. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1613–1618. ACM, 2005.

[12] D. Dasgupta and Z. Michalewicz. *Evolutionary Algorithms in Engineering Applications*. Springer, Berlin Heidelberg New York, 1997.

[13] R. Drechsler and B. Becker. *Binary Decision Diagrams: Theory and Implementation.* Kluwer Academic Publishers, Boston, USA, 1998.

[14] P. J. Bentley (ed). *Evolutionary Design by Computers.* Morgan Kaufmann Publishers, San Francisco, California, USA, 1999.

[15] P. Fiser. *Column-Matching Based Mixed-Mode BIST Technique.* PhD thesis, Czech Technical University in Prague, 2007.

[16] M. Fujita, P. C. McGeer, and J.C.-Y. Yang. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. *Formal Methods in System Design*, pages 149–169, 2004.

[17] Z. Gajda and L. Sekanina. Reducing the number of transistors in digital circuits using gate-level evolutionary design. In *2007 Genetic and Evolutionary Computation Conference*, pages 245–252. Association for Computing Machinery, 2007.

[18] Z. Gajda and L. Sekanina. Gate-level optimization of polymorphic circuits using Cartesian Genetic Programming. In *Proc. of 2009 IEEE Congress on Evolutionary Computation*, pages 1599–1604. IEEE Computational Intelligence Society, 2009.

[19] Z. Gajda and L. Sekanina. An efficient selection strategy for digital circuit evolution. In *Evolvable Systems: From Biology to Hardware*, LNCS 6274, pages 13–24. Springer Verlag, 2010.

[20] Z. Gajda and L. Sekanina. When does Cartesian Genetic Programming minimize the phenotype size implicitly? In *Proceeding of Genetic and Evolutionary Computation Conference, GECCO 2010*, pages 983–984. Association for Computing Machinery, 2010.

[21] K. Glette, J. Torresen, T. Gluber, B. Sick, P. Kafmann, and M.Platzner. Comparing evolvable hardware to conventional classifiers for electromyographic prostetic hand control. In *Proc. of 2008 NASA/ESA Conference on Adaptive Hardware and Systems*, pages 32–39. IEEE Computer Society, 2008.

[22] F. Gomez and R. Miikkkulainen. Incremental evolution of complex general behaviour. *Adaptive Behaviour*, 5:317–342, 1997.

[23] T. Gordon and P. Bentley. Evolving hardware. In A. Zomaya, editor, *Handbook of Nature Inspired and Innovative Computing*, pages 387–432. Springer Verlag, 2006.

[24] D. Green. *Modern Logic Design.* Addison-Wesley, 1986.

[25] G. W. Greenwood and A. M. Tyrrell. *Introduction to Evolvable Hardware: A Practical Guide for Designing Self-Adaptive Systems.* Wiley-IEEE Press, 2006.

[26] S. Harding. *Evolution in materio – PhD thesis.* University of York, UK, 2006.

[27] S. Harding, J. F. Miller, and W. Banzhaf. Evolution, development and learning with self modifying Cartesian Genetic Programming. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 699–706. ACM, 2009.

[28] S. L. Harding, J .F. Miller, and W. Banzhaf. Self-modifying cartesian genetic programming. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, volume 1, pages 1021–1028. ACM Press, 2007.

[29] L. Hellerman. A catalog of three-variable OR-Inverter and and-inverter logical circuits. *IEEE Trans. Electron. Comput.*, EC-12:198–223, 1963.

[30] T. Higuchi, Y. Liu, and X. Yao. *Evolvable Hardware*. Springer, 2006.

[31] B. I. Hounsell, T. Arslan, and R. Thompson. Evolutionary design and adaptation of high performance digital filters within an embedded reconfigurable fault tolerant hardware platform. *Soft Computing*, 8(5):307–317, 2004.

[32] S. M. Husband. *Programming the Nanocell,a Random Array of Molecules, PhD. Thesis*. PhD thesis, Rice University, Houston, Texas, USA, 2002.

[33] I. Kajitani, M. Iwata, and T. Higuchi. A GA hardware engine and its applications. In *Higuchi, T., Liu, Y., Yao, X., eds.: Evolvable Hardware*, pages 41–63. Springer, 2006.

[34] T. Kalganova. Bidirectional incremental evolution in extrinsic evolvable hardware. In *Proc. of the 2nd NASA/DoD Workshop on Evolvable Hardware*, pages 65–74. IEEE Computer Society, Silicon Valley, USA, July 2000.

[35] T. Kalganova and J. F. Miller. Evolving more efficient digital circuits by allowing circuit layout evolution and multi-objective fitness. In *The First NASA/DoD Workshop on Evolvable Hardware*, pages 54–63, Pasadena, California, 1999. IEEE Computer Society.

[36] M. Karnaugh. The map method for synthesis of combinational logic circuits. *Transactions of the American Institute of Electrical Engineers part I*, 72(9):593–599, 1953.

[37] P. Kaufmann and M. Platzner. Advanced techniques for the creation and propagation of modules in Cartesian Genetic Programming. In *Proc. of Genetic and Evolutionary Computation Conference, GECCO 2008*, pages 1219–1226. ACM, 2008.

[38] A. Keane, M. J. Streeter, and W. Mydlowec. *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Springer, New York, 2004.

[39] G. M. Khan and J. F. Miller. Evolution of cartesian genetic programs capable of learning. In *Genetic and Evolutionary Computation Conference, GECCO 2009, Proceedings, Montreal, Québec, Canada, July 8-12, 2009*, pages 707–714. ACM, 2009.

[40] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching – 2nd edition*. Addison Wesley, 1998.

[41] T. Kowaliw, W. Banzhaf, N. Kharma, and S. Harding. Evolving novel image features using genetic programming-based image transforms. In *Evolutionary Computation, 2009. CEC '09. IEEE Congress on*, pages 2502–2507, May 2009.

[42] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, Massachusetts, USA, 1992.

[43] J. R. Koza. *Genetic Programming II: Automatic Discoveryy of Reusable Programs.* MIT Press, Cambridge, Massachusetts, USA, 1994.

[44] J. R. Koza, F. H. Bennett III., D. Andre, and M. A. Keane. *Genetic Programming III: Darwinian Invention and Problem Solving.* Morgan Kaufmann Publishers, San Francisco, California, USA, 1999.

[45] S. Kumar. *Investigating Computational Models of Development for the Construction of Shape and Form, PhD thesis.* University of London, UK, 2004.

[46] S. Kumar and P. J. Bentley. *On Growth, Form and Computers.* Amsterdam, Netherlands, 2003.

[47] V. Kvasnicka, J. Pospichal, and P. Tino. *Evolucne algoritmy.* Tlac Vydavatelstvo STU v Bratislave, Bratislava, SK, 2000.

[48] W. B. Langdon and R. Poli. *Foundations of Genetic Programming.* Springer, Berlin Heidelberg, New York, 2002.

[49] J. Langeheine. *Intrinsic Hardware Evolution on the Transistor Level.* PhD thesis, Rupertus Carola University of Heidelberg, 2005.

[50] Z. Li, W. Luo, L. Yue, and X. Wang. On the completeness of the polymorphic gate set. *ACM Transactions on Design Automation of Electronics Systems*, 15(4), 2010.

[51] D. S. Linden. A system for evolving antennas in-situ. In *EH'01: Proceedings of The 3rd NASA/DoD Workshop on Evolvable Hardware*, pages 249–255. IEEE Computer Society, Washington D.C., USA, 2001.

[52] M. Loktev, O. Soloviev, and G. Vdovin. *Adaptive Optics – Product Guide.* OKO Technologies, Delft, 2003.

[53] W. Luo, Z. Zhang, and X. Wang. Designing polymorphic circuits with polymorphic gates: a general design approach. *IET Circuits, Devices & Systems*, 1(6):470–476, 2007.

[54] M. Mashayekhi, H. H. Ardakani, and A. Omidian. A new efficient scalable BIST full adder using polymorphic gates. *WORLD ACADEMY OF SCIENCE, ENGINEERING AND TECHNOLOGY*, 61:283–286, 2010.

[55] E. J. McCluskey. Minimization of Boolean functions. *Bell Sys. Tech. J.*, 35(5):1417–1444, 1956.

[56] E. J. McCluskey. *Introducing to the Theory of Switching Circuits.* McGraw-Hill, 1965.

[57] Z. Michalewicz and D. B. Fogel. *How to Solve It – Modern Heuristics.* Berlin, Heidelberg, New York: Springer, 2000.

[58] J. F. Miller. What bloat? Cartesian Genetic Programming on Boolean problems. In *2001 Genetic and Evolutionary Computation Conference Late Breaking Papers*, pages 295–302, 2001.

[59] J. F. Miller, D. Job, and V. K. Vassilev. Principles in the evolutionary design of digital circuits – part i. *Genetic Programming and Evolvable Machines*, 1(1):8–35, 2000.

[60] J. F. Miller, D. Job, and V. K. Vassilev. Principles in the evolutionary design of digital circuits – part ii. *Genetic Programming and Evolvable Machines*, 1(3):259–288, 2000.

[61] J. F. Miller and S. L. Smith. Redundancy and computational efficiency in Cartesian Genetic Programming. *IEEE Transactions on Evolutionary Computation*, 10(2):167–174, 2006.

[62] J. F. Miller and P. Thomson. Cartesian Genetic Programming. In *Proc. of the 3rd European Conference on Genetic Programming EuroGP2000*, volume 1802 of *LNCS*, pages 121–132. Springer, 2000.

[63] A. Mishchenko and R. Brayton. Scalable logic synthesis using a simple circuit structure. In *Proc. IWLS*, 2006.

[64] M. Murakawa, S. Yoshizawa, T. Adachi, S. Suzuki, K. Takasuka, M. Iwata, and T. Higuchi. Analogue EHW chip for intermediate frequency filters. In *Evolvable Systems: From Biology to Hardware, Second International Conference, ICES 98*, volume 1478 of *LNCS*, pages 134–143, Luasanee,Switzerland, 1998. Springer Berlin, Heidelberg, Germany.

[65] T. Pecenka, L. Sekanina, and Z. Kotasek. Evolution of synthetic RTL benchmark circuits with predefined testability. *ACM Transactions on Design Automation of Electronic Systems*, 13(3):1–21, 2008.

[66] W. V. Quine. A way to simplify truth functions. *Am. Math. Monthly*, 60(9):627–631, 1955.

[67] J. M. Quintana, M. J. Avedillo, R. Jimenez, and Rodriguez-Villegas E. Practical low-cost CPL implementations of threshold logic functions. In *Proc. of the 11th ACM Great Lakes Symposium on VLSI 2001*, pages 139–144, West Lafayette, Indiana, USA, 2001. ACM.

[68] R. L. Rudell and A. Sangiovanni-Vincentelli. Multiple-valued minimization for PLA optimization. *IEEE Trans. CAD*, CAD-6(5):727–750, 1987.

[69] R. Ruzicka. New polymorphic NAND/XOR gate. In *Proceedings of 7th WSEAS International Conference on Applied Computer Science*, volume 2007 of *Computer Science Challenges*, pages 192–196. World Scientific and Engineering Academy, 2007.

[70] R. Ruzicka. On bifunctional polymorphic gates controlled by a special signal. *WSEAS Transactions on Circuits And Systems*, 7(3):96–101, 2008.

[71] R. Ruzicka. Dependable controller design using polymorphic counters. In *Proc. of 12th Euromicro Conference on Digital System Design*, pages 355–362. IEEE Computer Society, 2009.

[72] R. Ruzicka and R. Prokop. Bifunctional NAND/NOR gates as building blocks for polytronics. In *Proceedings of CSE 2008*, pages 200–207. The University of Technology Kosice, 2008.

[73] R. Ruzicka and L. Sekanina. Evolutionary circuit design in REPOMO - reconfigurable polymorphic module. In *Proceedings of the Second IASTED International Conference on Computational Intelligence*, pages 237–241. ACTA Press, 2006.

[74] R. Ruzicka, L. Sekanina, and R. Prokop. Physical demonstration of polymorphic self-checking circuits. In *Proc. of 14th IEEE International On-Line Testing Symposium*, pages 31–36. IEEE, 2008.

[75] H. Sakanashi, M. Iwata, and T. Higuchi. A lossless compression method for halftone images using evolvable hardware. In *Evolvable Systems: Form Biology to Hardware, 4th International Conference, ICES 2001*, volume 2210 of *LNCS*, pages 314–326, Tokyo, Japan, 2001. Springer.

[76] H.-P. Schwefel. *Evolution and Optimum Seeking.* John Wiley, New York, USA, 1995.

[77] L. Sekanina. Image filter design with evolvable hardware. In *Applications of Evolutionary Computing – Proc. of the 4th Workshop on Evolutionary Computation in Image Analysis and Signal Processing EvoIASP'02*, volume 2279 of *LNCS*, pages 255–266, Kinsale, Ireland, 2002. Springer Verlag.

[78] L. Sekanina. *Evolvable Components: From Theory to Hardware Implementations.* Natural Computing Series, Springer Verlag, 2004.

[79] L. Sekanina. Evolutionary design of gate-level polymorphic digital circuits. In *Applications of Evolutionary Computing*, volume 3449 of *LNCS*, pages 185–194, Lausanne, Switzerland, 2005. Springer Verlag.

[80] L. Sekanina. Design and analysis of a new self-testing adder which utilizes polymorphic gates. In *Proc. of the 10th IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop DDECS 2007*, pages 246–246, Krakow, Poland, 2007. IEEE Computer Society.

[81] L. Sekanina. Evolution of polymorphic self-checking circuits. In *Proc. of the 7th Conf. on Evolvable Systems: From Biology to Hardware*, number 4684 in LNCS, pages 186–197, Wuhan, China, 2007. Springer.

[82] L. Sekanina and M. Bidlo. Evolutionary design of arbitrarily large sorting networks using development. *Genetic Programming and Evolvable Machines*, 6(3):319–347, 2005.

[83] L. Sekanina, T. Martinek, and Z. Gajda. Extrinsic and intrinsic evolution of multifunctional combinational modules. In *2006 IEEE World Congress on Computational Intelligence*, pages 9676–9683. IEEE Computational Intelligence Society, 2006.

[84] L. Sekanina, R. Ruzicka, and Z. Gajda. Polymorphic FIR filters with backup mode enabling power savings. In *Proc. of the 2009 NASA/ESA Conference on Adaptive Hardware and Systems*, pages 43–50. IEEE Computer Society, 2009.

[85] L. Sekanina, R. Ruzicka, Z. Vasicek, R. Prokop, and L. Fujcik. REPOMO32 - new reconfigurable polymorphic integrated circuit for adaptive hardware. In *Proc. of the 2009 IEEE Symposium Series on Computational Intelligence - Workshop on Evolvable and Adaptive Hardware*, pages 39–46. IEEE Computational Intelligence Society, 2009.

[86] L. Sekanina, L. Starecek, Z. Gajda, and Z. Kotasek. Evolution of multifunctional combinational modules controlled by the power supply voltage. In *Proc. of the 1st NASA/ESA Conference on Adaptive Hardware and Systems*, pages 186–193. IEEE Computer Society, 2006.

[87] L. Sekanina, L. Starecek, Z. Kotasek, and Z. Gajda. Polymorphic gates in design and test of digital circuits. *International Journal of Unconventional Computing*, 4(2):125–142, 2008.

[88] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical report, University California, Berkeley, 1992.

[89] L. Starecek, L. Sekanina, Z. Gajda, Z. Kotasek, R. Prokop, and V. Musil. On properties and utilization of some polymorphic gates. In *6th Electronic Circuits and Systems Conference (ECS 2007)*, pages 77–81. Faculty of Informatics and Information Technology STU, 2007.

[90] L. Starecek, L. Sekanina, and Z. Kotasek. Reduction of test vectors volume by means of gate-level reconfiguration. In *Proc. of 2008 IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop*, pages 255–258. IEEE Computer Society, 2008.

[91] A. Stoica, D. Keymeulen, A. Thakoor, T. Daud, G. Klimech, Y. Jin, R. Tawel, and V. Duong. Evolution of analog circuits on field programmable transistor arrays. In *Proc. of the 2000 NASA/DoD Conference on Evolvable Hardware*, pages 99–108, Palo Alta, California, USA, 2002. IEEE Computer Society.

[92] A. Stoica, D. Keymeyulen, R. S. Zebulum, S. Katkoori, S. Fernando, H. Sankaran, M. Mojarradi, and T. Daud. Self-reconfigurable mixed-signal integrated circuits architecture comprising a field programmable analog array and a general purpose genetic algorithm IP core. In *Evolvable Systems: From Biology to Hardware, 8th International Conference, ICES 2008*, volume 5216 of *LNCS*, pages 225–236, 2008.

[93] A. Stoica, R. Z., X. Guo, D. Keymeulen, I. Ferguson, and V. Duong. Taking evolutionary circuit design from experimentation to implementation: Some useful techniques and a silicon demonstration. *IEE Proc.-Comp. Digit. Tech.*, 151(4):295–300, 2004.

[94] A. Stoica and R. S. Zebulum. Faster evolution of more multifunctional logic circuits. *NASA Tech Briefs*, page 10, 2005.

[95] A. Stoica and R. S. Zebulum. Multifunctional logic gate controlled by supply voltage. *NASA Tech Briefs*, page 18, 2005.

[96] A. Stoica and R. S. Zebulum. Multifunctional logic gate controlled by temperature. *NASA Tech Briefs*, page 18, 2005.

[97] A. Stoica, R. S. Zebulum, and D. Keymeulen. Polymorphic electronics. In *Proc. of Evolvable Systems: From Biology to Hardware Conference*, volume 2210 of *LNCS*, pages 291–302. Springer, 2001.

[98] A. Stoica, R. S. Zebulum, D. Keymeulen, and J. Lohn. On polymorphic circuits and their design using evolutionary algorithms. In *Proc. of IASTED International Conference on Applied Informatics AI2002*, Insbruck, Austria, 2002.

[99] S. Tanachutiwat, J. U. Lee, W. Wang, and C. Y. Sung. Reconfigurable multi-function logic based on graphene P-N junctions. In *Design Automation Conference, DAC*, pages 883–888. ACM, 2010.

[100] A. Thompson. Silicon evolution. In *GECCO '96: Proceedings of the First Annual Conference on Genetic Programming*, pages 444–452, Cambridge, Massachusetts, USA, 1996. MIT Press.

[101] J. Torresen. A divide-and-conquer approach to evolvable hardware. In *Evolvable Systems: From Biology to Hardware, Second International Conference, ICES 98*, volume 1478 of *LNCS*, pages 57–65, Luasanee,Switzerland, 1998. Springer Berlin, Heidelberg, Germany.

[102] J. Torresen. Evolving multiplier circuits by training set and training vector partitioning. In *Proc. of Evolvable Systems: From Biology to Hardware Conference*, volume 2606 of *LNCS*, pages 165–172. Trondheim, Norway, 2003.

[103] Z. Vasicek, L. Capka, and L. Sekanina. Analysis of reconfiguration options for a reconfigurable polymorphic circuit. In *Proc. of the 2008 NASA/ESA Conference on Adaptive Hardware and Systems*, pages 3–10. IEEE Computer Society, 2008.

[104] V. K. Vassilev, D. Job, and J. F. Miller. Towards the automatic design of more efficient digital circuits. In *The Second NASA/DoD Workshop on Evolvable Hardware*. Palo Alto, California, USA, 2000.

[105] V. K. Vassilev and J. F. Miller. The advantages of landscape neutrality in digital circuit evolution. In *ICES '00: Proceedings of the Third International Conference on Evolvable Systems*, volume 1801 of *LNCS*, pages 252–263. Springer-Verlag, 2000.

[106] V. K. Vassilev and J. F. Miller. Scalability problems of digital circuit evolution. In *Proc. of the 2000 NASA/DoD Conference on Evolvable Hardware*, pages 55–64, Palo Alta, California, USA, 2000. IEEE Computer Society.

[107] S. P. Vingron. Karnaugh maps. *Switching Theory: Insight Through Predicate Logic*, pages 57–76, 2004.

[108] W. S. Lau and G. Li and K. H. Lee and K. S. Leung and S. M. Cheang. Multi-logic-unit processor: A combinational logic circuit evaluation engine for genetic parallel programming. *Genetic Programming*, 3447:167–177, 2005.

[109] J. F. Wakerly. *Digital Design: principles and practices – 3d edition.* Prentice Hall, New Jersey, USA, 2000.

[110] J. A. Walker and J. Miller. The automatic acquisition, evolution and re-use of modules in Cartesian Genetic Programming. *IEEE Transactions on Evolutionary Computation*, 12(4):397–417, 2008.

[111] J. A. Walker, J. F. Miller, and R. Cavill. A multi-chromosome approach to standard and embedded Cartesian Genetic Programming. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 903–910, New York, USA, 2006. ACM.

[112] N. Weste and D. Harris. *CMOS VLSI Design: A Circuits and Systems Perspective – 3rd edition.* Addison Wesley, 2004.

[113] S. Yang. *Logic Synthesis and Optimization Benchmarks User Guide, Version 3.0.* University of California and Standford University, California, USA, 1991.

[114] X. Yao and T. Higuchi. Promises and challenges of evolvable hardware. *IEEE Transactions on Systems, Man, and Cybernetics*, 29(1):87–97, 1999.

[115] T. Yu and J. F. Miller. Neutrality and the evolvability of Boolean function landscape. In *EuroGP '01: Proceedings of the 4th European Conference on Genetic Programming*, volume 2038 of *LNCS*, pages 204–217. Springer-Verlag, 2001.

[116] L. Zaloudek and L. Sekanina. Transistor-level evolution of digital circuits using a special circuit simulator. In *Evolvable Systems: From Biology to Hardware*, LNCS 5216, pages 320–331. Springer Verlag, 2008.

[117] R. S. Zebulum, M. Pacheco, and M. Vellasco. *Evolutionary Electronics – Automatic Design of Electronic Circuits and Systems by Genetic Algorithms.* The CRC Press International Series on Computational Intelligence, 2002.

[118] R. S. Zebulum and A. Stoica. Four-function logic gate controlled by analog voltage. *NASA Tech Briefs*, 30(3):8, 2006.

[119] R. S. Zebulum and A. Stoica. Three-function logic gate controlled by analog voltage. *NASA Tech Briefs*, page 17, 2006.

[120] R. Zeno. *A Reference of the Best-Known Sorting Networks for up to 16 Inputs.* http://www.angelfire.com/blog/ronz, 2010.

[121] S. Zhao and L. Jiao. Multi-objective evolutionary design and knowledge discovery of logic circuits based on an adaptive genetic algorithm. *Genetic Programming and Evolvable Machines*, 7(3):195–210, 2006.

[122] N. Zhuang and W. Haomin. A new design of the CMOS full adder. *IEEE journal of solid-state circuits*, 7(5):840–844, 1992.

[123] R. Zimmermann and R. Gupta. Low-power logic styles: CMOS vs CPL. In *Proceedings of the 22nd European Solid-State Circuits Conference*, pages 112–115, Neuchatel, Switzerland, 1996.