BRNO UNIVERSITY OF TECHNOLOGY

Faculty of Information Technology

PHD THESIS

Brno, 2023 Ing. Jitka Kocnová

BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

EVOLUTIONARY SYNTHESIS OF COMPLEX DIGITAL CIRCUITS

EVOLUČNÍ SYNTÉZA KOMPLEXNÍCH ČÍSLICOVÝCH OBVODŮ

PHD THESIS
DISERTAČNÍ PRÁCE

AUTHOR **Ing. JITKA KOCNOVÁ**
AUTOR PRÁCE

SUPERVISOR **Doc. Ing. ZDENĚK VAŠÍČEK, Ph.D.**
ŠKOLITEL

BRNO 2023

Abstract

The research presented in this thesis focuses on the field of evolutionary optimization of complex combinational circuits. The work begins with a study of the existing conventional and nonconventional approaches to the optimization of combinational circuits. Features and issues connected with the internal circuit representations commonly used by present synthesis tools. Boolean networks and their scalability were discussed. Attention was also paid to the evolutionary synthesis, with focus on the CGP (Cartesian Genetic Programming).

A new approach to the evolutionary optimization of combinational circuits was proposed. By extracting a sub-circuit containing a suitable number of gates of the original circuit and by optimizing this sub-circuit by the CGP, it was possible to reduce the number of gates of the circuit significantly more than by optimizing the whole circuit by the CGP. For the extraction phase, three methods were proposed. The first method is based on the cut computing algorithm. This method was able to reduce the number of gates of every benchmark circuit and it overcame the results of the globally working CGP in majority of cases. The second method is based on the windowing algorithm. This allows to expand the sub-circuit selection with the gates in the output direction of the root node of the selection and not only with the gates in its input direction. This method significantly improved the results obtained by using the cut-based method. It also overcame the issue of the cut-based method with selecting the sub-circuit near the primary inputs of the circuit and thus creating a selection too small for a subsequent optimization. The third method is based on the reconvergent-paths selection algorithm. The existence of a reconvergent-path in the sub-circuit increases the probability of presence of don't care nodes and thus the higher efficiency of the optimization. Also, an evolutionary optimization method targeting the non-uniform delay on the sub-circuit's inputs. By using this method, it is possible to extract and optimize a sub-circuit without an influence on the delay of the whole circuit.

By applying the principle of local evolutionary optimization, a significantly better gate reduction of the circuits was achieved then by applying the CGP optimization on whole circuits. However, it is important to choose the sub-circuit's root node carefuly with respect to its position in the circuit. Also, it is necessary to set the parameters of evolution, extraction and the whole optimization process carefully (e.g. the number of gates in each sub-circuit, number of CGP generations and number of sub-circuits that should be optimized).

Abstrakt

Tato dizertační práce prezentuje výzkum v oblasti evoluční optimalizace komplexních kombinačních obvodů. Práce začíná studiem existujících konvenčních i nekonvenčních přístupů k optimalizaci kombinačních obvodů. Byly analyzovány vlastnosti a problémy spjaté s nejčastěji používanými interními reprezentacemi v současných syntézních nástrojích. Dále byly představeny Booleovské sítě a možnosti jejich škálování. Pozornost byla také věnována evoluční syntéze logických obvodů, s důrazem na CGP (Kartézské Genetické Programování).

Byl navržen nový přístup k evoluční optimalizaci kombinačních obvodů. Extrahováním částí obvodů o vhodném počtu hradel a jejich následnou optimalizací pomocí CGP bylo dosaženo větší redukce počtu hradel v obvodech, než tomu bylo při optimalizaci celých obvodů pomocí CGP. K extrakci částí obvodů byly navrženy tři metody. První je založena na algoritmu vytvářejícím tzv. řezy. Tato metoda byla schopna optimalizovat každý testovací kombinační obvod a ve většině případů překonala výsledky dosažené optimalizací celých obvodů pomocí CGP. Druhá extrakční metoda je inspirována windowing algoritmem, díky čemuž je možné do výběru zahrnout i hradla nacházející se ve směru výstupů kořenového hradla výběru, a ne jen hradla ve směru jeho vstupů. Tato metoda výrazně vylepšila výsledky dosažené pomocí metody založené na tvorbě řezů. Metoda taktéž umožňuje, narozdíl od metody první, extrahovat část obvodu z jakéhokoli jeho místa, aniž by došlo k výběru příliš malého počtu hradel nevhodného k následné optimalizaci. Třetí metoda je založena na principu vyhledávání rekonvergentních cest v obvodech. Přítomnost rekonvergentní cesty ve vybrané části obvodu zvyšuje pravděpodobnost přítomnosti redundantních hradel a tím i vyšší efektivitu navrhovaného optimalizačního procesu. Byla také navržena a implementována evoluční optimalizační metoda zohledňující zpoždění obvodu. Touto metodou je možné extrahovat a optimalizovat část obvodu, aniž by celkové zpoždění obvodu přesáhlo požadovanou mez.

Pomocí principu lokální evoluční optimalizace bylo dosaženo lepších výsledků než při evoluční optimalizaci celých obvodů, čímž byla potvrzena hypotéza. Je však důležité vhodně zvolit umístění kořenového hradla výběru, vzhledem k jeho pozici v obvodě. Taktéž je třeba vhodně zvolit nastavení parametrů evoluce, extrakce i optimalizační metody jako celku (např. počet hradel v extrahovaných částech obvodu, počet CGP generací a počet částí obvodu, které projdou optimalizací).

Keywords

Evolutionary algorithms, CGP, synthesis, optimization, combinational circuits, delay.

Klíčová slova

Evoluční algoritmy, CGP, syntéza, optimalizace, kombinační obvody, zpoždění.

Reference

KOCNOVÁ, Jitka. *Evolutionary synthesis of complex digital circuits.* Brno, 2023. PhD thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Doc. Ing. Zdeněk Vašíček, Ph.D.

Evolutionary synthesis of complex digital circuits

Declaration

Prohlašuji, že jsem tuto dizertační práci vypracovala samostatně pod vedením pana doc. Ing. Zdeňka Vašíčka, Ph.D. Další informace mi poskytl prof. Ing. Lukáš Sekanina, Ph.D. Uvedla jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpala.

. . . .
Jitka Kocnová
December 4, 2023

Acknowledgements

Chci poděkovat svému školiteli Zdeňku Vašíčkovi za jeho odborné rady a velkou a vytrvalou pomoc při tvorbě této disertační práce. Dále děkuji kolegům z ústavu za odborné konzultace. Dále děkuji rodině, přátelům a známým za jejich podporu. Děkuji mému nejmilejšímu Davidu Grocholovi za každý společný den, obejmutí a slova útěchy.

Dále děkuji následujícím projektům, v rámci kterých tato dizertační práce vznikla:

- Application-specific HW/SW architectures and their applications, FIT-S-23-8141
- Navrhování a využívání knihoven aproximativních obvodů, GA14-04197S.
- Návrh, optimalizace a evaluace aplikačně specifických počítačových systémů, FIT-S-20-6309
- IT4Innovations excellence in science, NPU II - IT4IXS
- Rozvoj kryptoanalytických metod prostřednictvím evolučních výpočtů, GA16-08565S
- Pokročilé paralelní a vestavěné počítačové systémy, FIT-S-17-3994
- Ministerstvo školství, mládeže a tělovýchovy – INTER-COST projekt LTC18053
- AppNeCo: Aproximativní neurovýpočty, GA22-02067S

Contents

Chapter 1

Introduction

In recent years, the electronic industry has been experiencing a large growth. Increasing complexity of modern digital designs implies a large number of transistors in present circuits. As there is a linear dependence between the number of tranzistors and power consumption, limiting the dynamic as well as leakage power consumption has become one of the major interests when designing an electronic device [12, 13, 14]. Hence, during the designing phase, optimization of the number of components used in the final device is a crucial step. Furthermore, the size (eg. the number of nodes and interconnecting wires of a circuit) of each component is also important [45].

Nowadays, complex electronic problems require sophisticated solutions that can not be handled in practice without automation. Usually, the desired device or component is firstly described at a higher level, e.g. in a hardware description language (HDL) and then transformed by a sequence of steps into its final circuit representation. The transformation phase is called logic synthesis, a very complex and demanding process. It consists of a sequence of steps which transform a high-level description into a gate-level or transistor-level implementation based on a set of constraints and requirements. Automated logic synthesis plays a crucial role in development of Application Specific Integrated Circuits (ASIC), which experience an exponential growth [45].

An important part of logic synthesis is logic optimization. Its goal is to transform a suboptimal circuit solution into an optimal gate-level implementation with respect to given synthesis goals (eg. to reduce gate count or improve performance). Logic synthesis is not always able to create an optimal gate-level implementation, but it can significantly improve the efficiency and performance of a circuit. Due to the scalability issues, the problem is typically expressed using a suitable internal representation. Then, technology mapping tries to transpose it onto its best standard cell implementation using the desired library of logic gates or cells. The efficiency of mapping depends on the cell library and capability to recognize certain portions of logic as a library gate [10].

Historically, the synthesis is broken down into two isolated processes: optimization and technology mapping. By dividing the optimization and technology mapping into separate processes, an inefficiency is brought up due to the loss of information. Optimization working with the internal representation may not be able to recognize all of the cells (eg. adders) of the circuit because the internal representation (such as AIG – and-inverter graph) does not carry any information about certain nodes belonging to larger logic blocks.

1.1 Conventional Approaches

Current state-of-the-art logic synthesis tools represent circuits using an intermediate representation. The most commonly used is the and-inverter graph (AIG), a directed acyclic graph composed of two-input AND nodes connected by direct or negated edges [32]. This representation is simple and scalable, and leads to simple yet efficient algorithms. However, it suffers from an inherent bias in representation. Eight of the ten possible two-input logic gates can be represented by means of a single AIG node. The remaining two, namely XOR and XNOR gates, require three AIG nodes each. The efficiency of synthesis is then limited as it mostly fully relies on transformations that disallow an increase in the number of AIG nodes. Also, the synthesis algorithms typically do not treat XORs explicitly – they rely on identification of XORs during the technology mapping phase which works independently on the logic optimization phase. The ability to capture XOR gates is, however, essential for efficient representation of arithmetic and XOR-intensive circuits [20, 19].

To address the inherent bias in AIG representation, various approaches have been proposed. E.g. binary decision diagrams (BDDs, two-input multiplexer networks also denoted as binary decision diagrams) can be employed [62, 59, 53]. BDDs are useful for a quick check if a function can replace another one by inspecting the tautology of their equivalence. Also, BDDs are frequently used for logic function decomposition which is obtained by using a maximum fanout free cone (MFFC) based on partial collapse technique. Efficient BDD-based variable partitioning heuristics are used to decompose all collapsed nodes into k-LUT feasible sub-functions in order to achieve minimized LUT area. Despite having excellent results for XOR-intensive networks, this representation and the methods built upon it do not excel when working with other circuits, such as AND/OR-intensive networks [1].

In order to explicitly support XOR gates in logic synthesis, XOR-AIG representation was introduced by Fiser et al. [17, 9]. The synthesis can be then based on a modified rewriting algorithm where subgraphs with four leaves are selected. Due to the limited scalability of XOR-AIG, a two step synthesis process based on a selective and distinct manipulation of AND/OR and XOR-intensive portions of the logic circuit has been employed [1]. In the first phase, XOR-intensive regions are identified in the input Boolean network. These regions are then optimized independently of the rest of the network. In the second phase, technology independent mapping is done.

Moreover, Majority-inverter Graphs (MIGs consisting of majority gates MAJ and inverters) [2] have been introduced as many modern technologies can be modelled by three-input majority gates together with inverters. Arithmetically complex circuit described by MIG can be optimized more effectively than AIGs thanks to the majority operation. Instead of AIG, XOR majority graphs (XMGs) have been employed to extend the capabilities of the synthesis oriented on area optimization [27]. Each gate in such a network represents MAJ or XOR gate and each connection can be inverted. This representation enables smaller networks, which implies faster synthesis.

To summarize, the conventional methods rely on circuit preprocessing or circuit decomposition [1], precomputation of optimal solutions [17, 9] or presence of advanced technology cells such as XMG.

1.2 Unconventional Approaches

Besides the conventional techniques, there are also unconventional ones presented in the scientific literature that apply machine learning approaches to synthesise circuits. These techniques, that are typically based on a mathematical model, predominantly rely on the soft-computing methods such as the evolutionary algorithms (EA) [36]. Unconventional techniques have been successfully applied for circuit optimization since early nineties [34, 35]. Advancements in technology developed in the early nineties enabled researchers to successfully apply techniques of evolutionary computation in various problem domains. In the middle nineties, Higuchi and Thompson, two of the most prominent pioneers, demonstrated that evolutionary algorithms are able to solve non-trivial hardware-related problems [28, 55]. Genetic algorithm has been employed also by Coello, who evolved various 2-bit adders and multipliers [11]. Finally, Miller et al. demonstrated that evolutionary design systems are not only able to rediscover standard designs as it has been shown in the past, but they can, in some cases, improve them [40, 37]. The achievements presented in the seminal paper of Higuchi et al. [28] motivated other scientists to intensively explore a new and promising research topic. As a consequence of that, new research direction referred to as *Evolvable hardware* has emerged [25] focusing on the use of evolutionary algorithms to create specialized electronics without manual engineering.

One of the most successful methods in the field of digital circuit synthesis is the Cartesian Genetic Programming (CGP) [39, 35, 38]. Various modifications of CGP working directly at the level of gates were successfully applied to address the internal representation problem [48, 56]. Vasicek demonstrated that the evolutionary synthesis using CGP conducted directly at the level of gates is able to provide significantly better results compared to the state-of-the-art synthesis operating on AIGs [56]. A similar approach was successfully applied even to synthesis of conventionally hard to synthesize circuits [18].

It was observed, however, that the efficiency of the evolutionary approach deteriorates with an increasing number of gates. Substantially more generations were required to reduce circuits consisting of more than ten thousand gates. While [56] focuses strictly on the improvement of the scalability of the evaluation, Sekanina et al. employed a divide and conquer strategy to address the problem of scalability of representation [48]. The authors were able to obtain better results than other locally operating methods reported in the literature, however, the performance of this method was significantly worse than the evolutionary global optimization proposed in [56].

In the past, machine learning methods (eg. neural networks, evolutionary algorithms, etc.) were mostly researched and applied in the electronic design automation – for example, in design placement [60] and reduction [15]. Nowadays, modern logic synthesis tools also benefit from developments in machine learning field [26, 44]. Eg. Authors in [29] introduced approaches based on deep neural networks and reinforcement learning (RL) to obtain optimal structural transformations.

1.3 Research Objectives

In order to improve the results of EA-based synthesis, this thesis proposes a divide and conquer strategy that does not rely on circuit decomposition, preprocessing or precomputation of optimal solutions. This strategy combines the evolutionary optimization with the principle of the so-called Boolean network scoping and it is based on an iterative optimization of large portions of the original circuit similarly to the conventional rewriting. Boolean network scoping represents a common approach incorporated in the conventional synthesis tools for maintaining the good scalability of the synthesis process. The optimization strategy presented in this thesis works as follows. At first, a logic circuit is optimized by means of a common synthesis approach. Then, the obtained circuit is mapped to standard gates and optimized again using our method that extracts sub-circuits of a manageable size that are subsequently optimized by the CGP. The extracted sub-circuit is then replaced by its optimized variant provided that there is an improvement at the global level and the whole process is repeated. This approach can be understood as the EA-based resynthesis [21, 5]. Compared to the rewriting, which iteratively selects sub-circuits and replaces them with smaller precomputed subgraphs (e.g low tens of gates), the evolutionary resynthesis benefits from a substantially higher number of gates (e.g. low hundreds of gates) that are more likely to be further reduced.

We hypothesize that the circuit resynthesis based on the iterative sub-circuit selection and the subsequent evolutionary optimization of every sub-circuit can achieve a significantly better gate reduction of the circuit compared to the optimization applied to the whole circuit.

The following research objectives were formulated:

- To study present conventional and evolutionary techniques of optimization of digital circuits.
- To propose various methods which extract sub-circuits suitable for the evolutionary optimization.
- To experimentally validate proposed methods using relevant set of circuits.
- Based on previously obtained results, modify the sub-circuit extraction method to account for non-uniform delays at the inputs of the extracted sub-circuits.

Undoubtedly, there is a certain risk of worsening some of the circuit parameters that will not be targeted during optimization, such as delay or power. Additionally, there is a chance that the sub-circuit extraction methods will not be able to select a sub-circuit suitable for the subsequent optimization - e.g. the sub-circuit will contain only a few gates when selecting a sub-circuit near the primary inputs. Also, the sub-circuit selection is not trivial and requires domain knowledge.

1.4 Thesis Outline

This thesis is composed of a collection of papers and is organized as follows. The study of the state-of-the-art is present in Chapter 2. It includes the principles of logic synthesis and evolutionary algorithms. A special attention is devoted to the logic optimization and the Cartesian Genetic Programming. Chapter 3 presents the research methodology and scientific papers contributing to this thesis. Chapter 4 summarizes and concludes the thesis and proposes possible future research. Finally, five research related and peer-reviewed papers are attached in the section Related Papers.

Chapter 2

State of the Art

This chapter provides the background to the research covered in this thesis. Chapter 2.1 introduces Boolean networks and contains notation used in the rest of this work. The most common optimization methods are discussed in Chapter 2.2. Different approaches to the Boolean network scoping are discussed in Chapter 2.3.

2.1 Boolean Networks

There are several possibilities on how to represent the combinational and sequential logic circuits. Further, we will focus on the combinational circuits only. Boolean Networks, among others, represent one of the most popular mathematical model used to describe the logic circuits.

Considering the combinational logic, a Boolean network can be understood as a directed acyclic graph (DAG) with nodes represented by Boolean functions [43]. Further is an explanation of the most important terms connected to the Boolean networks, that will be used in this work. Some of the terms are also illustrated in Figure 2.1. The sources of the graph are the *primary inputs* (PIs). These are signals that are driven by the environment, there is no node driving these signals in the network. The sinks of the network are the *primary outputs* (POs). POs are signals that drive the environment and are needed by inner network nodes as well. The output of a node can be an input to other nodes called *fanout nodes* (nodes 12, 13 and 14 in Figure 2.1). *Fanout* refers to the number of digital inputs that a single output of a logic gate can drive without violating the electrical specifications of the gate. Specifically, fanout refers to the maximum number of gate inputs that can be connected to the output of a gate, such that the voltage levels at the output remain within acceptable ranges and the gate's output can still drive the connected inputs. The inputs of a node are called *fanin nodes* (nodes 6 and 7 in Figure 2.1). *Fanin* is the number of inputs that a node is driven by. An *edge* connects two nodes that are in fanin/fanout relationship. *Transitive fanout* is a set of nodes on every possible path between a certain node and POs (nodes 12–16 in Figure 2.1). Similarly, *transitive fanin* is a set of nodes that exist on every possible path between PIs and a certain node (nodes 2, 3, 4, 6, 7 in Figure 2.1). The *size* of the network is the number of the nodes (primary inputs and outputs are not considered). The *delay* is further understood as the number of gates on the longest topological path (LTP) of the network. *Don't cares* are understood as input combinations that never come from the environment and outputs that are not observed by the environment.

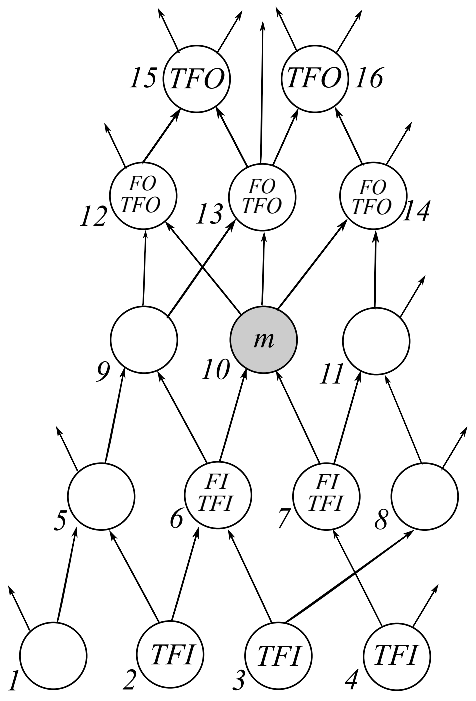


Figure 2.1: Example of the *fanin* and *fanout nodes* of a selected node *m*. *Fanout nodes* are marked as *FO* and *fanin nodes* as *FI*. *Transitive fanout* set is marked as *TFO* and *transitive fanin* set is marked as *TFI*. The *FO* nodes are included in the *TFO* and *FI* nodes in *TFI*.

Representing a network as a graph is beneficial, as graphs allow reuse of nodes – a node can be connected to the output of any of the previous nodes [39]. A circuit can be modeled by means of a Boolean network with various set of logic functions $f_i \in \Gamma$. Thus, a different set of logic functions in nodes can be considered, eg. $\Gamma = \{NAND\}$ or $\Gamma = \{XOR, AND\}$. Considering this notation, AIG is a Boolean network composed of two-input ANDs and inverted edges.

2.2 Boolean Optimization

Logic optimization methods can be divided into two categories, the algebraic methods and the Boolean methods. The algebraic methods treat a Boolean function as a polynomial and are usually faster. However, the Boolean methods are more accurate when it comes to the true nature of logic functions [54]. Current state-of-the-art logic synthesis tools often represent a Boolean network using an internal representation, such as AIG, XOR-AIG, XMGs etc. as mentioned in Chapter 1. Thus, the optimization process needs to use algorithms adapted to work with such representations.

The optimization of AIGs is mostly based on *rewriting*, an algorithm which minimizes the size of AIG by replacing its parts with a precomputed optimal networks [42, 46, 16]. The principle of rewriting can be seen in Figure 2.2, where the networks N1, N2 and N3 represent the precomputed optimal networks and can be used as a replacement for a part of the network. In order to precompute the optimum network, conventional rewriting has to enumerate the space of Boolean functions. The bigger the Boolean network, the larger the number of all possible candidate solutions is and its enumeration becomes impossible

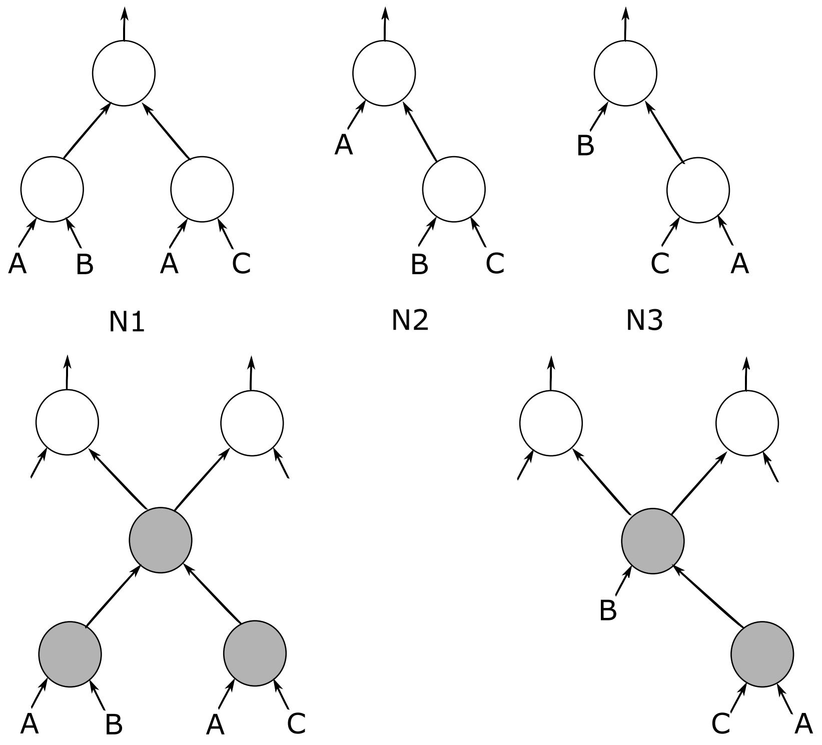


Figure 2.2: Principle of rewriting. The three Boolean networks on the top (labeled as N1, N2, N3) represent three possible variants of a simple circuit represented using AIG. The bottom-left image shows identification of a subcircuit (gray nodes) which is replaced by its smaller precomputed variant (N3) in the bottom-right picture. [43]

to cope with [27]. In order to be able to work with large networks, rewriting became a greedy algorithm which minimizes the size of AIG by iteratively selecting subgraphs rooted at a node and replacing them with smaller precomputed subgraphs, while preserving the functionality of the root node [42, 46]. AIG rewriting is local, however, the scope of changes becomes global by application of rewriting many times. This process has its drawbacks, such as limited scalability (working only with small networks) and unsatisfactory strategies used for subnetwork replacement [27].

In addition to that, *resubstitution* and *refactoring* can be employed. Boolean resubstitution is considered to be one of the most powerful methods in logic synthesis [54]. Its principle can be seen in Figure 2.3 where the original network was reduced by one gate while preserving the logic function of the circuit. Resubstitution expresses the function of a node using other nodes present in the AIG [43] in order to achieve a more compact implementation. This can be done thanks to the so-called *don't-cares* [6] and *permissible functions* [54, 47] (original function of a node n is changed to a different function without changing the behavior in PIs thanks to don't-cares). Resubstitution is an expensive method when it comes to runtime, which implies the necessity of partitioning the network into smaller subgraphs which can be easily manipulated with.

Refactoring iteratively selects large cones of logic rooted at a node and tries to replace them with a more efficient implementation [42]. Refactoring can be seen as a variant of rewriting. The main difference is that rewriting selects subgraphs containing few leaves

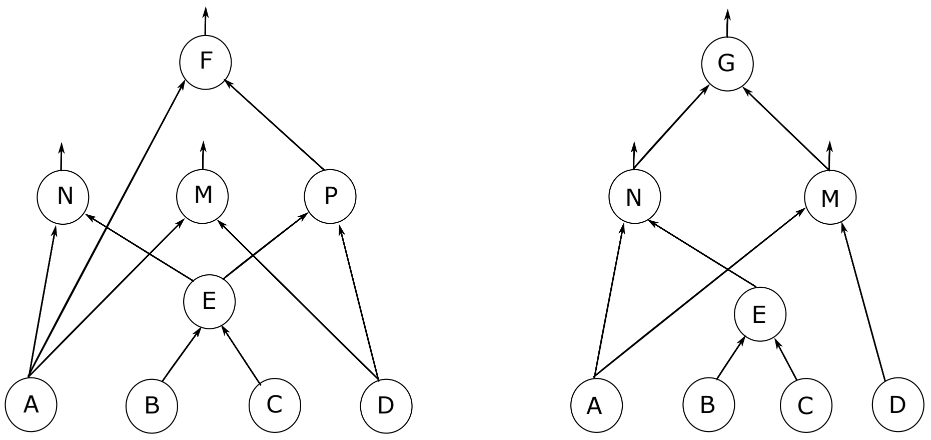


Figure 2.3: Principle of resubstitution. Node P can be removed and its functionality is replaced by other nodes while preserving the function of the circuit. [43]

because the number of leaves determines the number of variables of a Boolean function whose optimal implementation is sought.

2.3 Scaling of Boolean Synthesis

Scalability is one of the essential issues of the synthesis process, mainly when working with large Boolean networks. The more complex the circuit's structure is and the more inefficient work with the internal representation is, the less scalable the synthesis is likely to be. The goal is to ensure a suitable and simple internal representation and adjust the synthesis algorithms to be able to execute the transformations effectively over a circuit described by such an internal representation [43, 7].

Network scoping represents a key operation to ensure a good scalability of synthesis tools. In addition, it forms an integral part of rewriting as well as refactoring. Two approaches have been proposed to limit the scope of logic synthesis to work only on a small portion of a Boolean network – *cut computation* and *windowing* [43].

Cut computation is an approach based on computing the so-called k-feasible cuts [33]. It is usually preferred to avoid determining the required number of logic levels that are needed to be traversed to get a sub-circuit of the desired size. A cut of a node, called root node, is a set of nodes of the network, called leaves, such that each path from PI to the root node passes through at least one leaf. A cut is k-feasible if the number of leaf nodes (i.e. cut size) in the cut does not exceed k. The volume of a cut is the total number of nodes encountered on all paths between the root node and the cut leaves. An example of two different 3-feasible cuts is shown in Figure 2.4. To maximize the cut volume, a reconvergence-driven heuristic is applied in practice. The problem is that the cut computed using a naive bread-first-search algorithm may include only a few nodes and results in tree-like logic structures. Such a structure does not lead to any don't-cares in the local scope of the node and attempting optimization using such a cut would be wasted time. A simple and efficient cut computation algorithm producing a cut close to a given size while heuristically maximizing the cut volume and the number of reconvergent paths subsumed in the cut has been introduced in [43].

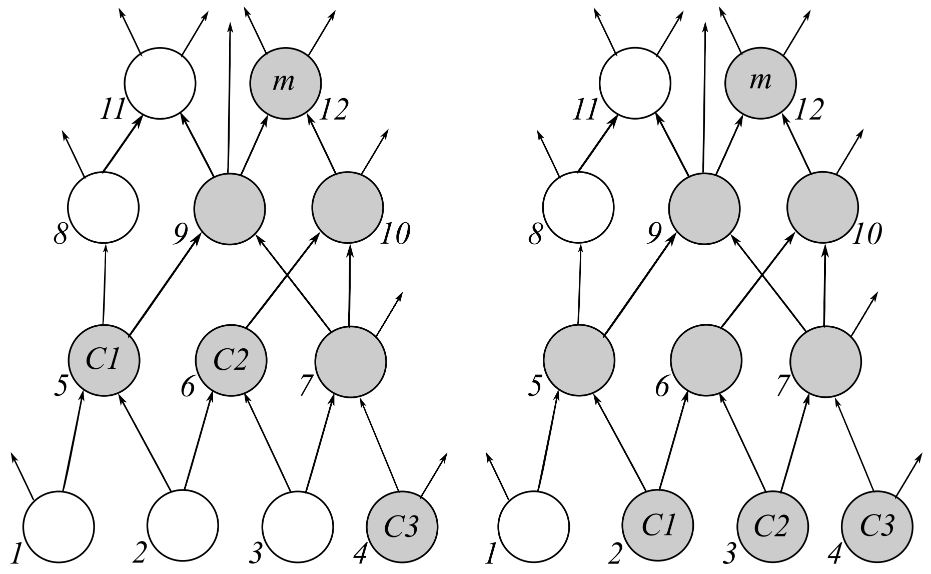


Figure 2.4: Example of two possible 3-feasible cuts for root node m and given Boolean network. The cut on the right is preferred as its volume is four (root node m and contained nodes 5, 7, and 9). There is only one contained node (node 8) in the case of the left cut.

The k-feasible cuts are important not only for the gate-level logic synthesis but also for FPGA-based synthesis as a k-feasible cut can be implemented as a k-input LUT. For resubstitution and FPGA-based mapping, so-called maximum fanout free cone (a subnetwork where no node in the cone is connected to a node not in the cone) is requested. It means that the cut-based scoping must always produce a single-output sub-circuits. Otherwise it would be impossible to replace the whole sub-circuit by a precomputed optimal implementation / a single LUT. Typically, 4-feasible and 5-feasible cuts are used for rewriting-based logic synthesis [43, 33]. Small k is used not only to make the cut enumeration possible but also to manage memory requirements to store the precomputed optimal implementations of all k-input Boolean functions. For FPGA-based mapping, 5-input and 6-input LUTs are used. Apart from the rewriting, the reconvergence-driven cuts have been applied to refactoring and resubstitution [43]. Typically, k is between 5 and 12 for refactoring depending on the computation effort allowed [43].

The windowing algorithm determining the window for a given node takes a node and two integers defining the number of logic levels on the fanin/fanout sides of the node to be included in the window. Two sets are produced as the result of windowing – leaf set and root set. The window of a Boolean network is the subset of nodes of the network containing nodes from the root set together with all nodes on paths between the leaf set and the root set. The nodes in the leaf set are not included in the window. This method has been used, for example, for computing don't-cares and redundancy removal [43]. The main problem of this algorithm is that it is hard to predict how many logic levels have to be traversed to get a window of the desired size and required number of leaves.

The size of the sub-circuits has an impact not only on the scalability of the synthesis tool but also on the efficiency of the whole optimization process. Small sub-circuits ensure a good scalability of the evolutionary optimization, but they lead to minor improvements at the global level because we obtained a method which operates mainly locally similarly to the conventional rewriting. Huge sub-circuits, on the other hand, increase possibilities for an improvement but the performance of the evolutionary optimization deteriorates with increasing the size of the optimized circuit. In order to have a reasonable optimization method, it is necessary to find a good trade-off between the mentioned two extremes.

2.4 Evolutionary Synthesis of Logic Circuits

Evolutionary algorithms (EAs) are a class of stochastic algorithms inspired by the principles of biological evolution. In order to solve a particular optimization problem, EAs use genetic operators to modify the set of candidate solutions consisting of a population of individuals. These operations are *mutation*, *recombination* and *reproduction* [22]. By mutation, some parts of an individual are randomly modified. Recombination exchanges some parts between two or more individuals. When using reproduction, an individual is copied to the offspring set of candidate solutions without any modification. The fitness function is evaluated for the set of candidate solutions to express the quality of each individual [30].

Evolutionary computation techniques have been successfully applied in various problem domains since the early nineties [28, 55]. The first results in the area of digital circuit synthesis were reported by Koza in 1992, who investigated the evolutionary design of even-parity circuits in his extensive discussions of the standard genetic programming (GP) paradigm [31]. EAs have been used to synthesize logic circuits since the late nineties [34, 35]. Miller et al., the author of Cartesian Genetic Programming [39], is considered as a pioneer in the field of logic synthesis of gate-level circuits. He utilized his own variant of genetic programming to synthesize compact implementations of multipliers described by means of a behavioral specification [58]. CGP is described in detail in the next section because it is relevant for this thesis.

2.5 Cartesian Genetic Programming

Based on the results achieved over the last ten years, it appears that CGP seems to be the most powerful evolutionary technique in the domain of EA-based logic synthesis and optimization [35]. The word Cartesian in CGP means that the graph nodes are arranged in the Cartesian coordinate system – in a 2D array where each node is located at its x and y coordinates. In the past, 2D array was commonly used, whereas today a linear form of CGP is preferred. In this case, CGP models a candidate circuit having n_i PIs and n_o POs as a linear 1D array of n_n configurable nodes that are addressed in a Cartesian coordinate system. This way, the circuit is represented as the DAG. Each node has n_a inputs and corresponds with a single gate with up to n_a inputs. Two-input and single-output nodes are typically used. The inputs can be connected either to the output of a node placed in the previous L columns or directly to PIs. This avoids a feedback. The function of a node can be chosen from a set Γ consisting of $|\Gamma| = n_f$ functions. A genotype is a list of node connections and functions and can be seen as a one-dimensional string of integers. Depending on the function of a node, some of its inputs may become redundant. In addition to that, some of the nodes may become redundant because they are not referenced by any node connected to a PO. It means that the fixed number of nodes n_n does not mean that all the nodes are used effectively. The redundant nodes and inputs lead to the presence of non-coding genes in the genotype. This feature makes the search effective [41].

The candidate circuits are encoded as follows. Each PI as well as each node has associated an unique index. Each node is encoded using $n_a + 1$ integers $(x_1, \cdots, x_{n_a}, f)$ where

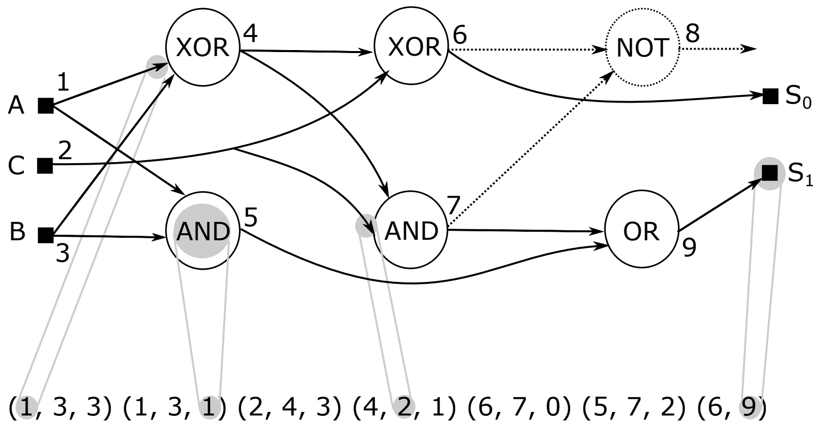


Figure 2.5: Example of a CGP encoding of a logic circuit (one-bit full adder) with $n_i = 3$ inputs and $n_o = 2$ outputs. The individual is encoded using an array of $n_n = 6$ two-input single-output nodes whose functions are chosen from a set of primitive functions $\Gamma = \{\mathrm{NOT}, \mathrm{AND}, \mathrm{OR}, \mathrm{XOR}\}$. Note that the nodes are arranged in a two-dimensional grid for improved readability. Redundant connections and nodes, i.e. those that do not contribute to the outputs, are highlighted using a dotted line.

the first n_a integers denote the indices of its fanins and the last integer determines the function of that node. Every candidate circuit is encoded using $n_n(n_a + 1) + n_o$ integers where the last n_o integers specify the indices corresponding with each PO. An example of a CGP encoding a one-bit full adder circuit can bee seen in Figure 2.5.

The most common search technique used in connection with the CGP is Evolutionary strategy (ES) [35, 8]. Typically $(1 + \lambda)$-ES is employed, where λ corresponds with the number of new candidate solutions generated from a single parental solution. In the circuit optimization, the initial population is seeded by the original circuit ought to be optimized. Every new population consists of the best circuit chosen from the previous population and its λ offspring created using a mutation operator. Either point or probabilistic mutation is used in the standard CGP. Point mutation is typically preferred because it is easier to implement and more efficient than using a probabilistic mutation [41].

The point mutation randomly modifies up to h genes (integers) of a parent genotype to create an offspring. Considering the CGP encoding, a single mutated gene causes either reconnection of a node, reconnection of a primary output or change in function of a node. Due to the presence of redundant genes, the mutation may occur in the redundant part, which means that the mutated genotype has the same phenotype as its parent. Such a mutation is sometimes denoted as neutral since the fitness value remains unchanged. To avoid wasted fitness evaluations, several mutation strategies have been proposed [24, 41]. Single Active Mutation strategy, for example, mutates the offspring until one active gene is changed. Another possibility is to detect the neutral mutations and skip the time-consuming fitness evaluation procedure. Considering the usage of CGP in the optimization of logic circuits, the latter approach has been typically used [52, 57, 56]. Crossover is not used in the standard CGP because it was found that crossover has little effect on the efficiency of CGP [41].

The main disadvantage of the CGP encoding in connection with the point mutation operator is the presence of a strong length and positional bias that results in large portions of the genotype that are always redundant and never used by any ancestor. To address this issue, several approaches have been proposed [41]. Goldman and Punch, for example, proposed to apply Reorder operation once each generation shuffles the position of nodes in the parent [23]. Reorder does not semantically change the parent but it allows active nodes to be evenly distributed within the whole genotype. This approach eliminates the length as well as positional bias and improves the efficiency of the search.

The selection of the individuals is typically based on a cost function (e.g. the number of active nodes). In the case that there are more individuals with the same score, the individual that has not served as a parent will be selected as the new parent. This procedure is typically repeated for a predefined number of iterations.

There are several modifications to the original CGP, such as embedded CGP, modular CGP, multi-chromozome CGP (MC-CGP) or self-modifying CGP [39, 51, 50, 63, 49]. Embedded CGP extends the CGP by including so-called Automatically Defined Functions (ADFs) to address the problem of scalability of representation. ADFs are modules that can be dynamically created, destructed, evolved (by the mutation operator) and reused. Modules are created from a section of the CGP's genotype. Modular CGP is a modification of Embedded CGP, where the modules can be nested. MC-CGP uses multiple chromosomes within a single genotype. This way, a complex multiple-output problem can be divided into a set of single-output sub-problems that are co-evolved. The sub-chromosomes have equal length. Each sub-chromosome is connected to a single program output. In self-modifying CGP (SMCGP), the phenotype of an individual can change over time. Connections of a node in SMCGP are not strictly defined and the node is not connected to particular nodes. Instead, a node has a relative address of its connections, meaning, that its inputs are connected to nodes in a certain distance (eg. two nodes back). Modification is ensured by the presence of nodes that modify their own graph.

Chapter 3

Research Summary

3.1 Methodology

The goal of this thesis is to improve the possibilities of EAs in the digital circuit synthesis. Attention is focused on developing suitable methods for extraction of small portions of logic circuits that can be optimized by means of CGP and placed back to their original location providing there is an improvement to the desired criterion in the whole circuit. First steps of the research are concentrated on development of the network scoping methods. Then these methods are combined with the optimization engine based on CGP and evaluated on a set of benchmark circuits. The last step presents a modification of the scoping methods that allow the optimization process to consider other criteria such as the circuit delay. Figure 3.1 shows the flow of the EA-based resynthesis compared to the flow of EA-based optimization where the CGP optimizes whole circuits without sub-circuit extraction.

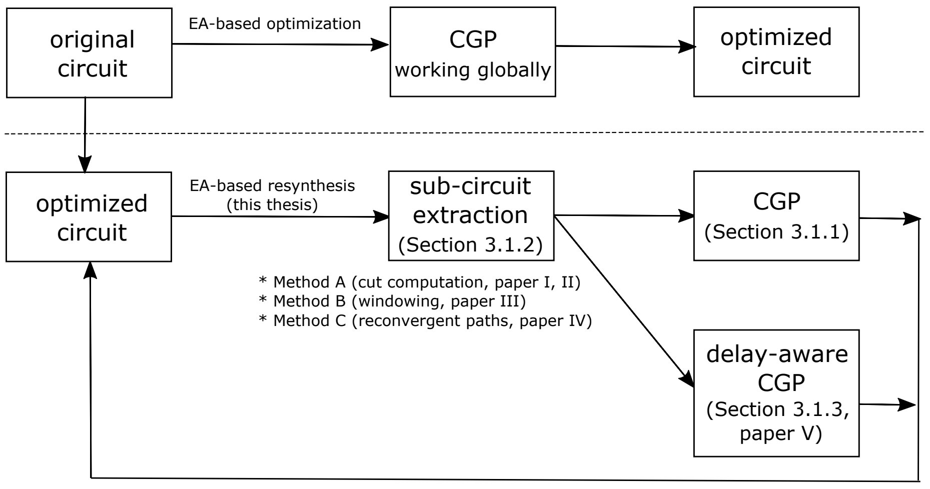

original circuit
EA-based optimization
CGP working globally
optimized circuit
optimized circuit
EA-based resynthesis (this thesis)
sub-circuit extraction (Section 3.1.2)
CGP (Section 3.1.1)
* Method A (cut computation, paper I, II)
* Method B (windowing, paper III)
* Method C (reconvergent paths, paper IV)
delay-aware CGP (Section 3.1.3, paper V)

Figure 3.1: Schematic visualization of the EA-based optimization flow.

3.1.1 Evolutionary optimization

Algorithm 1 describes the overall principle of the evolutionary optimization method presented in this thesis. It is an iterative process that consists of a sequence of three steps that are executed in a loop. The goal is to obtain a more efficient alternative to the original

Boolean network. To determine the improvement a cost function is used. The cost function can reflect various objectives (and their combinations as well), such as circuit's size, delay, power consumption, etc. At first, such a sub-circuit W is selected by one of the extraction methods that are mentioned in Section 3.1.2. In order to avoid wasting computational time, the sub-circuit's suitability for subsequent optimization is inspected. The inspection can consider various parameters, for example volume of a cut – if a cut contains only a few gates, the subsequent evolutionary optimization will likely not have a significant impact on it. If the sub-circuit is not suitable, it is discarded and a different sub-circuit is selected. Then, the sub-circuit is optimized by means of the CGP with respect to the desired criterion This part of the algorithm represents the evolutionary step of the optimization. Each node in the sub-circuit W is assigned an unique index and a chromosome corresponding with the nodes in the sub-circuit is created. The chromosome then represents the sub-circuit W in the CGP optimization. After the evolutionary phase, an optimized sub-circuit W' is built from the chromosome. W' has to satisfy the optimization conditions on the sub-circuit's cost (e.g. contains less or equal gates than the original sub-circuit) while being functionally equal to its non-optimized version W. If these two conditions are met, the original sub-circuit W is replaced by its optimized variant W' in the circuit N. This way, the optimized circuit N' is obtained. Then, the optimization continues by selecting and optimizing another sub-circuit or finishes if the termination condition is satisfied (e.g. desired number of optimization iterations was computed). The number of iterations should be determined heuristically, as well as the CGP parameters.

Algorithm 1: Optimization of digital circuits using EA-based resynthesis

```
1  A Boolean network $N$ Optimized network $N'$, $cost(N') \leq cost(N)$
2  $N' \leftarrow N$
3  while termination condition not satisfied do
4      $W \leftarrow$ GetSubcircuit($N'$) ;
5      if $W$ is not a suitable candidate then
6          continue
7      $W' \leftarrow$ OptimizeNetworkUsingEA($W$,$N'$)
8      if cost($(N' \setminus W) \cup W'$) < cost($N'$) then
9          $N' \leftarrow (N' \setminus W) \cup W'$
10 return $N'$
```

3.1.2 Limiting the Scope of Boolean Networks

The method introduced in this thesis is inspired by the network scoping approach that helps conventional logic synthesis algorithms to work only on a small portion of the original circuit. This is motivated by the fact that CGP is more efficient when dealing with medium-sized networks due to the exponential expansion of the large search space. Chapter 2.3 introduced two of the most common techniques that are used for network scoping: cut computation and windowing. However, as our preliminary experiments showed, using their original form for the purpose of this thesis would be inefficient because these techniques were developed for a different purpose. For example, the cut computation, based on k-feasible cuts, is commonly used for extracting a subcircuit having only k inputs.

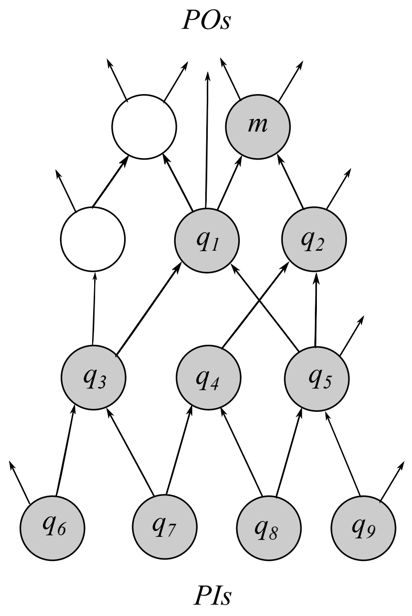


Figure 3.2: Example of a cut consisting of 9 nodes created using the Method A. The root node is marked as m and is the first node that the cut consists of. The labels q_i inside the nodes denote the order i in which the nodes were chosen. Leave nodes q_1 and q_2 of the root node m are added to the cut. Then, leave nodes of q_1 and q_2 are one after another added to the cut until the cut reaches its desired volume.

However, we would not want to be limited by the number of *PIs* or *POs* of the sub-circuit. The method should be able to extract a sub-circuit of desired volume no matter how many *PIs* or *POs* it has. This can be overcome by using the windowing algorithm. The windowing algorithm needs the number of logic levels on the fanin/fanout sides of the node to be included in the window. Our preliminary experiments showed that these features would be quite limiting for the evolutionary optimization method developed in this thesis. Instead of limiting the number of inputs of the subcircuits or predicting the number of logic levels, our attention is focused on the volume (number of gates present in the selection) and on the presence of don't-cares in the subcircuit.

Three approaches are proposed in this thesis to limit the scope of logic synthesis to work only on a small portion of a Boolean network: *cut-based computation*, *windowing* and *reconvergent paths selection.*

The cut computation approach (Method A) works as follows. The algorithm starts with a set of leaves consisting of a single root node and an empty cut set. If the leaves set is empty, the procedure terminates. Otherwise, a node that minimizes a cost function is chosen from the set of leaves and added to the cut set. The chosen node is removed from the leaf set and all its fanins are included instead of it. This causes expansion of the cut. If the cut-volume limit of the cut set is exceeded, the procedure quits and returns the cut. The complete algorithm can be found in attached Paper I and Paper II and an example of a cut consisting of nine nodes can be seen in Figure 3.2. At first, the root node m is present in the leaves set. After it is added to the cut, nodes q_1 and q_2 replace m in the leaves set. One by one, nodes q_1 and q_2 are removed from the leaves set, added to the cut and their leaves, q_3, q_4, q_5 and q_6 are put in the leaves set. The procedure continues when the cut volume reaches nine nodes.

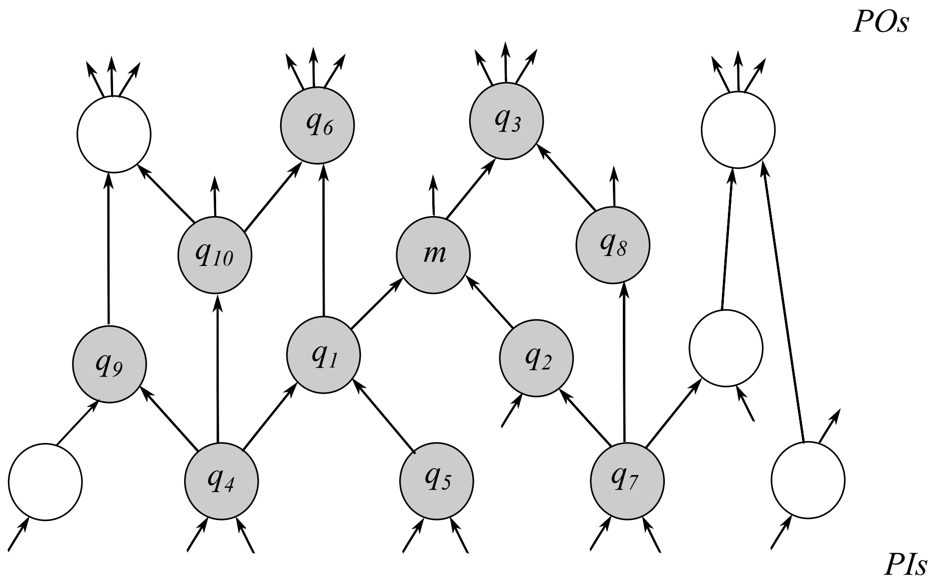


Figure 3.3: Example of the window consisting of 10 nodes ($w_{max} = 10$) created using the proposed alternative windowing algorithm. The neighboring nodes added into W are highlighted using the filled nodes. The nodes at the bottom are primary inputs. The nodes in the window have assigned an index (the number located below a particular node) used to uniquely identify each node in the CGP. The labels q_i inside the nodes denote the order i in which the nodes were chosen.

The windowing algorithm (Method B) starts with the root node and a number determining the desired volume of the window. In the beginning, the window contains only the root node denoted as m. In the first iteration, all the nodes, denoted as q_i, that are in the *fanin* and *fanout* of the root node are added to the window. Then, every node that is directly connected to any of the nodes already present in the window is also added to the window, until the window reaches the desired volume. The complete algorithm can be found in attached Paper III and an example of a final state of a window consisting of 10 nodes is shown in Figure 3.3.

In Method A, the cut expands only towards the PIs of the network. If the position of the root node m is close to the PIs, the cut expansion can terminate before the desired volume of the cut is reached. Compared to Method B, the window expands in the direction both of PIs and POs of the network. Thus, the position of the root node m doesn't have such a crucial influence of the window volume. It means that there is a high chance that the window contains at least one node with fanin or fanout nodes that the window can be expanded with.

The reconvergent paths selection (Method C) is derived from windowing. Reconvergent paths lead from one source node through two different areas of the network and meet again in a receiving node that is in the fanout cone of the source node. At first, the reconvergent path is found in the network and its nodes are included into the window. The window is then filled in with nodes in the fanin/fanout relationship with the nodes on the reconvergent path, until the window reaches the desired size. The complete algorithm can be found in attached Paper IV. An example of a window containing a reconvergent path can be seen in Figure 3.4, where the reconvergent path consists of five gates and is accompanied by six surrounding nodes in the window selection.

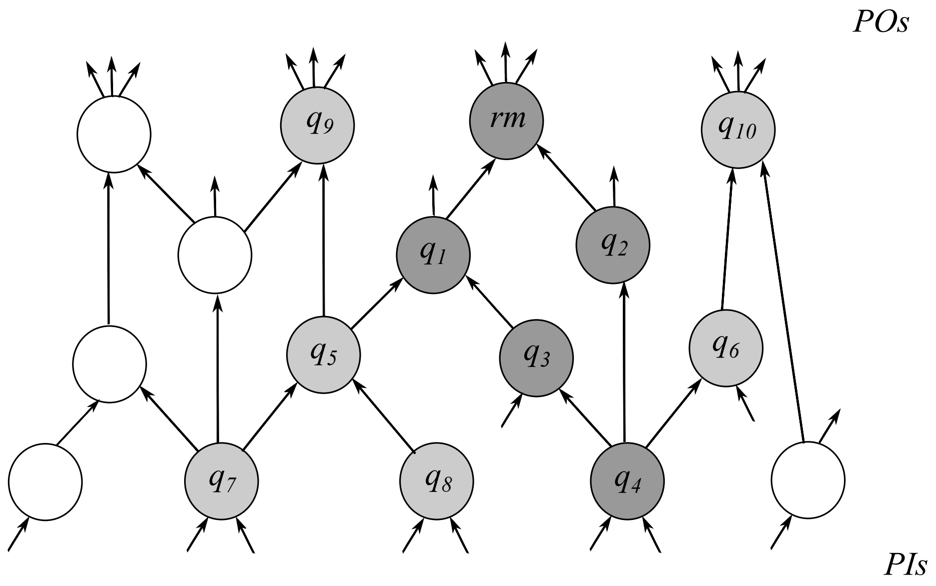


Figure 3.4: Example of the window consisting of 10 nodes ($rw_{max} = 10$) created using the proposed algorithm based on selecting reconvergent paths. The root node is marked as m. Nodes on the reconvergent path are highlighted using the dark gray filled nodes. These nodes are the first nodes added to the RW The neighboring nodes added into RW are highlighted using the light gray filled nodes. The nodes at the bottom are primary inputs. The nodes in the window have assigned an index (the number located below a particular node) used to uniquely identify each node in the CGP. The labels q_i inside the nodes denote the order i in which the nodes were chosen.

All of the proposed extraction approaches can be easily used for the extraction of the subcircuits suitable for CGP-based optimization. However, the sub-circuit extraction itself is connected to some issues that have to be considered. It is not obvious how to identify a suitable sub-circuit that can be successfully optimized and improves the cost of the whole circuit after it is implanted back. Another unknown parameter is the proper size of a sub-circuit. Optimization of large sub-circuits may produce good results. On the other hand, the computation would need a lot of time for execution. Working with small sub-circuits would be fast, however, such sub-circuits are hard to improve by the evolution as they contain only a few nodes. Thus, optimization of small sub-circuits may waste computational time while not improving the circuit at all. Another issue is the influence of the optimization process on the non-targeted circuit parameters, e.g. delay. While the optimized circuit may have a good cost function of the targeted parameter, the cost functions of the other ones may get significantly worse.

3.1.3 Targeting the Non-uniform Delay on the Sub-circuit's Inputs

As the experiments with the extraction methods described in Paper I and Paper III and Paper IV showed, the concept of locally applied optimization is effective and achieves promising results considering the complexity of the circuits. It means that the number of gates was reduced even for circuits consisting of tens of thousands of gates. However, the number of gates is not the only parameter that is affected by the optimization process. Extensive analysis of the obtained results showed that the delay is also affected – in majority of cases negatively.

As a result of the analysis, we proposed a new method, extending the EA-based optimization proposed in Section 3.1.2. This method consists of the following steps. At first, a sub-circuit is determined using windowing or reconvergent paths selection algorithm described in Section 3.1.2. Then, the delays of POs of the sub-circuit are determined and every gate in the sub-circuit is assigned with an information about the highest depth value of the primary output that the sub-circuit has in its fanout. Secondly, the maximal delay increase is set up. Thirdly, optimization by means of the CGP targeting the smallest possible number of gates is performed, reflecting the maximum allowed depth of the circuit. After the CGP step finishes, the optimized sub-circuit is implemented back to the circuit. The depth of the optimized circuit must not exceed the allowed increase set before the optimization step. The complete algorithm can be found in attached Paper V and an example is shown in Figure 3.5.

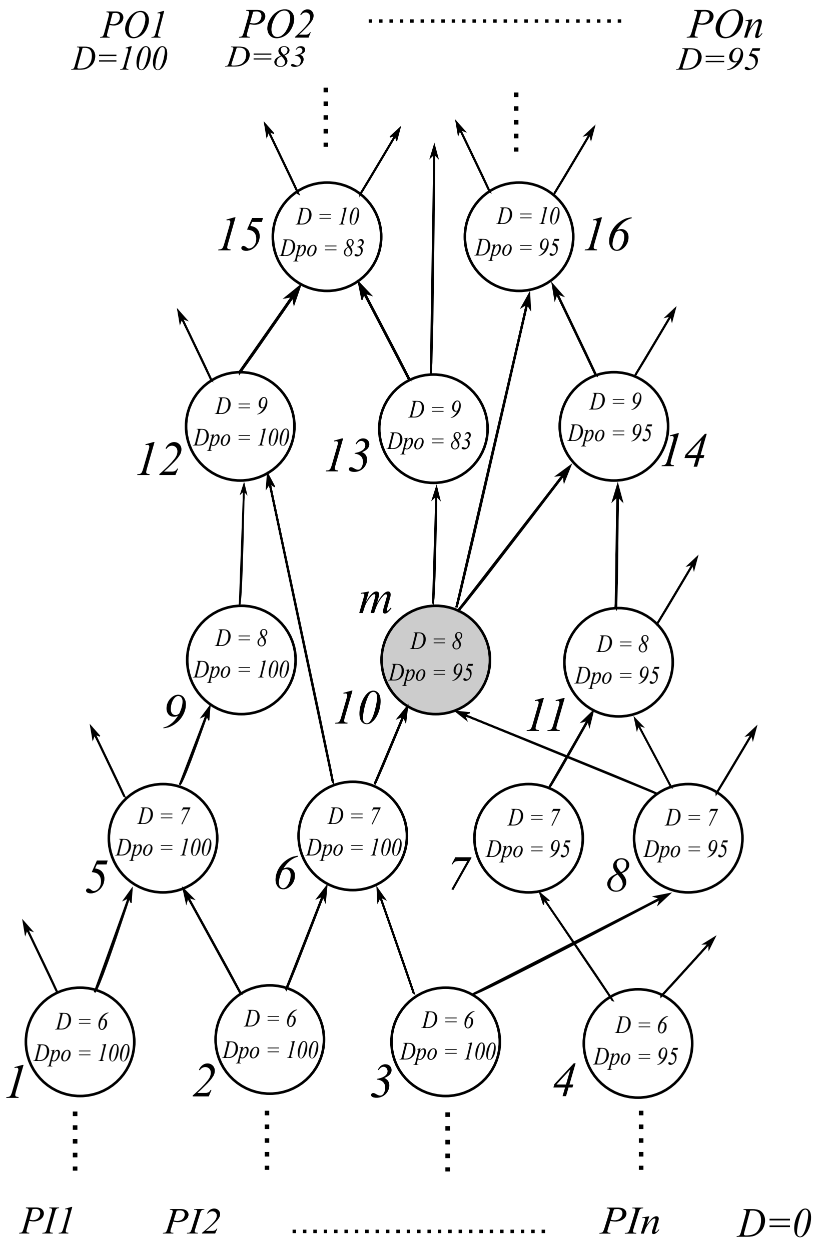


Figure 3.5: Example of the window build from the root node m consisting of sixteen nodes. At the bottom of the picture, there are *PIs*. At the top of the picture, there are *POs*. Each PO is assigned with its depth *D*. After the window is established, each of its nodes is assigned with two values: its depth (D) and the worst depth (D_{po}) of a *PO* that the node has in its transitive fanout. The worst D_{po} of the window is then subtracted from the delay of the whole circuit. The CGP is then assigned with this value representing the value of a depth that the optimized window is allowed to have. Eg. if the circuit's depth is $D = 120$ and the worst D_{po} is $D_{po} = 100$, this window is allowed to grow its delay by 20.

3.1.4 Experimental Evaluation

We have implemented the proposed algorithms in C++ and integrated them as a part of Yosys open synthesis suite [61]. The advantage of this tool, among others, is that it allows us to directly manipulate the Verilog files and that it integrates ABC [7], a state-of-the-art academic tool for hardware synthesis and verification.

To ensure fair evaluation, each of the proposed sub-circuit extraction algorithms was evaluated on the same set of circuit benchmarks. This set contains 28 highly optimized real-world combinational networks containing IWLS'05 Open Cores benchmarks and a set of arithmetic circuits [4, 3].[1] The circuits were firstly optimized by ABC and mapped to common 2-input gates including XORs/XNORs gates. Optimization by the proposed methods was then executed and final number of gates in circuits together with the circuit's depth is examined. All of the optimized circuits were then formally verified with respect to their original form (ABC command 'cec'). Number of CGP generations and optimization iterations were set up heuristically. The root node was chosen randomly for extraction of each sub-circuit by each proposed method. To avoid wasting computational time on optimizing sub-circuits that are too small, the lowest possible number of gates in a sub-circuit was set to 10 gates, based on our preliminary experiments.

Table 3.1 shows overall results for all proposed extraction methods. In particular, it reports the average and the best gate reduction when optimizing the set of benchmarks mentioned above. Also, last two columns of the table present the results of globally working CGP.

The first experimental evaluation was done using the Method A. The experiments demonstrated the strength of the locally applied optimization and confirmed the hypothesis introduced in Section 1.3. It also confirmed the inefficiency of the internal circuit representation and of the optimization performed before technology mapping which is a method commonly used in the conventional tools. The proposed methods were able to reduce the number of gates in every circuit. This implies a presence of redundant gates in the circuits that were firstly optimized by the ABC tool as was mentioned earlier in this section. Each method also outperformed the CGP working with whole circuits in majority of cases, especially when optimizing the arithmetic circuits.

However, this approach also showed an inefficiency of its computation. When we analyzed its results, the ratio of the amount of CGP generations that caused a change to the sub-circuit and contributed to the circuit optimization was lower than the number of generations applied to every sub-circuit during the optimization process. This was caused by the nature of the extraction method. According to the analysis of the parameters of the extracted subcircuits, this method produces sub-circuits with a lack of possible don't-cares in the sub-circuit. The optimization thus converged prematurely (reached the local sub-optimal state) before all of the CGP generations were applied. Thus a significant amount of time was wasted on computing CGP generations that did not change the circuit at all while being stuck in a stable state without any change.

This was the motivation to improve the extraction method in order to be able to select sub-circuits more suitable for optimization. Method B is able to select sub-circuits containing as many interconnections between the selected nodes as possible, while building the sub-circuit from a root node in every possible direction. Thus, a much more compact sub-circuit with a higher probability of presence of don't-care nodes is extracted compared to the cut-based method. This has a positive effect on the gate reduction, where the

[1]The benchmarks can be found at https://lsi.epfl.ch/MIG

windowing-based method significantly outperformed the cut-based method in 22 out of 28 cases. Convergence of this method was more gradual compared to the cut-based method thanks to compactness of the sub-circuits.

As can be seen, both extraction Methods A and B have advantages and disadvantages. In order to further improve the extraction process, mainly its convergence and efficiency, we tried to extract the best features of both techniques. This resulted in the reconvergent paths selection based method being developed. Presence of a reconvergent path in a window-like sub-circuit increases the possibility of presence of redundant nodes in the sub-circuit. The Method C, was constructed based on our assumption of a higher probability of a good optimization result. Evaluation of this extraction method confirmed this assumption as it outperformed the windowing-based method in more than 70% of cases. However, when optimizing circuits with only a few reconvergent paths, the optimization was not so successful, as the sub-circuit selection often extracted a sub-circuit from a similar place in the circuit containing the same reconvergent path.

In Section 3.1.3 the impact of the proposed local evolutionary optimization approach on the circuit's delay was discussed. Simultaneously, the modification of Method B and Method C was proposed. This modification targeted possible growth of the delay due to the negative impact of the optimization process to the circuit's delay. These two delay-aware methods were able to identify and remove a significant number of redundant gates while preserving the delay at the original or desired value. Naturally, the experiments demonstrated that limiting the maximal circuit delay during the optimization causes a worse gate reduction. When using the extraction methods without delay-aware modification, the delay at the end of the optimization process was usually lower than the worst delay achieved during optimization. Also, in some cases a slight increase led to a significant gate reduction (more than 50% of the reduction compared to the original methods). The other cases were not so successful. That is caused mainly by the depth limitation itself: a potentially good optimization iteration that would bring a good gate reduction or subcircuit modification is wasted because the depth of the whole circuit would surpass the allowed limit. This result is expectable as the number of gates and delay are conflicting criteria. Thus, the optimization can not be expected to produce both compact and fast circuits. The results obtained from experimental evaluation of the two proposed delay-aware methods can be seen in Paper V.

As follows from the experimental evaluation, we were able to successfully combine our sub-circuit extraction methods together with the CGP. This has made the evolutionary optimization of complex combinational circuits more efficient compared to the CGP working with whole circuits at once. We have confirmed the hypothesis that the circuit resynthesis based on the iterative sub-circuit selection and the subsequent evolutionary optimization of every sub-circuit can achieve a significantly better gate reduction of the circuit compared to the optimization applied to the whole circuit. Also, we have proposed and experimentally evaluated the method targeting the non-uniform delay at the inputs of the circuits, that is able to reduce the number of gates while keeping the desired delay.

Table 3.1: Comparison of the proposed methods based on reconvergent-paths selection (Method C), windowing (Method B) and cuts (Method A) against ABC and globally working CGP. For each of the proposed methods and globally working CGP, the average and the best result is presented in means of percentage of removed gates. Column ‘ABC‘ contains parameters of the optimized circuits after mapping ('gates' is the number of gates, D is logic depth).

			ABC		Impr. Method C		Impr. Method B		Impr. Method A		Impr. global [56]	
Benchmark	PIs	POs	gates	D	avg	best	avg	best	avg	best	avg	best
DSP	4223	3792	43491	45	**4.9%**	**5.3%**	1.4%	2.13%	**3.6%**	**3.6%**	0.0%	0.0%
ac97_ctrl	2255	2136	11433	10	**4.6%**	**5.6%**	3.0%	4.0%	2.9%	2.9%	1.4%	1.4%
aes_core	789	532	21128	20	**8.5%**	**9.8%**	4.2%	5.5%	2.9%	2.9%	0.6%	1.7%
des_area	368	70	5199	25	**6.4%**	**7.4%**	4.5%	5.2%	6.0%	6.1%	2.1%	2.3%
des_perf	9042	1654	78972	16	**7.3%**	**9.5%**	2.8%	4.2%	1.8%	1.8%	0.0%	0.1%
ethernet	10672	10452	60413	23	**1.9%**	**2.6%**	1.6%	1.7%	0.5%	0.5%	0.0%	0.0%
i2c	147	127	1161	12	**24.7%**	**25.7%**	18.3%	18.5%	9.2%	9.2%	10.0%	10.7%
mem_ctrl	1198	959	10459	24	9.6%	10.9%	6.2%	10.0%	7.0%	7.0%	**24.8%**	**25.4%**
pci_bridge32	3519	3136	19020	21	**6.7%**	**7.1%**	3.4%	4.7%	3.5%	3.5%	0.5%	0.6%
pci_spoci_ctrl	85	60	1136	15	**43.0%**	**46.9%**	31.5%	36.7%	18.3%	18.5%	34.8%	35.7%
sasc	133	123	746	8	**21.0%**	**24.9%**	7.4%	7.4%	6.2%	6.2%	2.4%	2.8%
simple_spi	148	132	822	11	**11.9%**	**14.0%**	6.6%	7.4%	5.5%	5.7%	4.4%	4.6%
spi	274	237	3825	26	9.3%	10.4%	5.0%	8.4%	5.6%	5.6%	**13.5%**	**20.2**%
ss_pcm	106	90	437	7	**12.4%**	**13.0%**	4.8%	5.5%	5.7%	6.7%	2.3%	2.3%
systemcaes	930	671	11352	27	**12.1%**	**19.8%**	11.7%	12.7%	11.9%	12.3%	0.0%	0.0%
systemcdes	314	126	2601	25	**19.5%**	**21.4%**	15.7%	15.9%	4.8%	5.0%	9.1%	9.9%
tv80	373	360	8738	39	10.5%	11.6%	**13.5%**	**14.2%**	6.6%	6.9%	11.1%	11.3%
usb_funct	1860	1692	15405	23	**10.4%**	**14.0%**	10.2%	11.3%	5.8%	5.9%	2.6%	2.6%
usb_phy	113	73	452	9	**29.1%**	**30.3%**	17.7%	18.0%	13.9%	14.0%	12.2%	12.2%
average (IWLS'05 benchmarks)			15620	20	**13.4%**	**15.3%**	6.4%	6.5%	6.4%	6.5%	7.0%	7.6%
mult32	64	64	8225	42	**21.6%**	**24.6%**	19.5%	20.9%	16.5%	16.6%	0.0%	0.0%
sqrt32	32	16	1462	307	17.1%	**26.2%**	6.6%	9.5%	**22.3%**	24.3%	3.0%	3.0%
diffeq1	354	193	20719	218	8.7%	11.4%	**25.5%**	**28.6%**	**11.5%**	**11.5%**	0.0%	0.0%
div16	32	32	5847	152	20.6%	25.1%	**29.5%**	**42.7%**	15.7%	15.8%	0.0%	0.0%
hamming	200	7	2724	80	45.9%	52.9%	**58.8%**	**58.9%**	28.6%	30.1%	14.6%	14.6%
MAC32	96	65	7793	55	5.5%	6.4%	**9.5%**	**10.5%**	7.7%	7.8%	0.0%	0.0%
revx	20	25	8131	171	7.9%	9.0%	**18.0%**	**21.2%**	14.5%	14.5%	0.0%	0.1%
mult64	128	128	21992	190	**12.6%**	**12.9%**	5.0%	6.2%	7.4%	7.4%	0.3%	0.5%
max	512	130	3719	117	5.2%	**5.6%**	5.1%	5.2%	**5.3%**	5.3%	0.7%	0.8%
average (arithmetic benchmarks)			8956	148	14.5	17.4	**19.7%**	**22.6%**	14.4%	14.8%	2.1%	2.1%

3.2 Papers

The introduced methodology and results have been published within several scientific papers. An abstract, a brief description and a conclusion of the contribution is presented for each paper in this section. Full text of each paper can be found in Section 4.1.

3.2.1 Paper I

Kocnova, J., Vasicek, Z. (2019). Towards a Scalable EA-Based Optimization of Digital Circuits. In: Sekanina, L., Hu, T., Lourenço, N., Richter, H., García-Sánchez, P. (eds) Genetic Programming. EuroGP 2019. Lecture Notes in Computer Science, vol 11451. Springer, Cham. https://doi.org/10.1007/978-3-030-16670-0_6

Author participation: 50%
Conference rank: B1 (Qualis)

Abstract Scalability of fitness evaluation was the main bottleneck preventing adoption of the evolution in the task of logic circuits synthesis since the early nineties. Recently, various formal approaches have been introduced to this field to overcome this issue. This made it possible to optimise complex circuits consisting of hundreds of inputs and thousands of gates. Unfortunately, we are facing another problem – scalability of representation. The efficiency of the evolutionary optimization applied at the global level deteriorates with the increasing complexity. In this paper, we propose to apply the concept of local resynthesis. Resynthesis is an iterative process based on the extraction of smaller sub-circuits from a complex circuit that are optimized locally and implanted back to the original circuit. When applied appropriately, this approach can mitigate the problem of scalability of representation. Our evaluation on a set of non-trivial real-world benchmark problems shows that the proposed method provides better results compared to the global evolutionary optimization. In more than 60% cases, a substantially higher number of redundant gates was removed while keeping the computational effort at the same level.

Contribution In the past, various evolutionary approaches working directly at the level of gates were successfully applied in the field of circuit optimization. These works aim to overcome problems with scalability of common internal representations, such as AIG, used by state-of-the-art synthesis tools. The main goal of our work is to improve the result of EA-based synthesis by combining EA-based approach with refactoring while following the principle of local resynthesis applied in common logic synthesis tools. Performance of the proposed method is evaluated on a set of highly optimized real-world combinational circuits and compared to the result provided by ABC (the state-of-the-art academic tool for hardware synthesis and verification) and CGP working with whole circuits.

This work introduces a new approach to the evolutionary optimization. It iteratively selects subcircuits which are then optimized by means of CGP and returned back to the circuit provided that there is a reduction of gates. The subcircuits are selected by a relatively naive method based on the cut computation with random root node selection and no limitation on the number of inputs of the cut. Using this divide-and-conquer strategy, this approach was able to reduce the number of gates in circuits by more than 9% on average. The highest achieved reduction was 28.6% when optimizing the hamming circuit. Compared to the CGP optimizing the whole circuits globally, the proposed method achieved better results in more than 60% of cases.

3.2.2 Paper II

J. Kocnova and Z. Vasicek, „EA-Based Refactoring of Mapped Logic Circuits,“ 2019 IEEE International Symposium on Circuits and Systems (ISCAS), Sapporo, Japan, 2019, pp. 1-5, doi: 10.1109/ISCAS.2019.8702084.

Author participation: 50%
Conference rank: A1 (Qualis)

Abstract The increasing complexity of the designs and problematic scalability of original representations led to a shift in internal representations used in logic synthesis and optimization. Heterogeneous representations were replaced with homogeneous intermediate representations. And-inverter graph (AIG) has been identified as the most promising structure for scalable logic optimization and many efficient algorithms were implemented on top of it. However, the inability of AIG to efficiently represent XOR gates together with the heuristic nature of logic optimization algorithms leads to some inefficiency causing that the logic can be further minimized even after it has been mapped. This paper presents an optimization technique based on refactoring targeting mapped combinational circuits. It iteratively selects large cones of logic, optimizes them and returns them back to the original structure provided that there is an improvement in some metric. Performance of the method is evaluated on a set of complex academic and industrial benchmarks. We show that a 9.2% reduction in area can be achieved on average compared to the highly optimized results obtained using the academic state-of- the-art synthesis tool. On average, more than 14% reduction was observed for arithmetic circuits.

Contribution The recent conventional optimization methods need to perform circuit preprocessing or decomposition in order to handle the complex networks more efficiently. Moreover, internal representation, such as AIG, used in logic synthesis brings up an inefficiency – the circuits can be further optimized after they are mapped to the standard gate representation. What is more, optimizing circuit before mapping can be limiting as it does not allow an increase to the number of gates of the internal representation. However, this increase can be beneficial because during the mapping phase, XOR or NXOR gates may be identified, which implies mapping three AIG gates to one final logic gate. This work brings out experimental results of the proposed evolutionary optimization method based on iterative selection of subcircuits which are subsequently optimized by CGP. This method was able to reduce the number of gates of each benchmark circuit at the level of common gates. Additionally, this work demonstrates the inefficiency of AIG by comparing the number of AIG nodes of each circuit before and after the evolutionary optimization.

Despite using a very simple strategy of root node selection which may degrade the capabilities of the refactoring, the proposed method is able to outperform the AIG-based as well as the original EA-based optimization applied to the whole Boolean networks. The proposed method outrun the AIG-based optimization tool by 6.4% on the controller benchmarks and by more than 14% on the arithmetic benchmarks.

3.2.3 Paper III

Kocnova, J., Vasicek, Z. EA-based resynthesis: an efficient tool for optimization of digital circuits. Genetic Programming and Evolvable Machines 21, 287–319 (2020).
https://doi.org/10.1007/s10710-020-09376-3

Author participation: 50%
Conference rank: Q2 (Core)

Abstract Scalability of fitness evaluation was the main bottleneck preventing adopting the evolution in the task of logic circuits synthesis since the early nineties. Recently, various formal approaches such as SAT and BDD solvers have been introduced to this field to overcome this issue. This made it possible to optimise complex circuits consisting of hundreds of inputs and thousands of gates. Unfortunately, we are facing another problem – scalability of representation. The efficiency of the evolutionary optimization applied at the global level deteriorates with the increasing complexity. To overcome this issue, we propose to apply the concept of local resynthesis in this work. Local resynthesis is an iterative process based on the extraction of smaller sub-circuits from a complex circuit that are optimized locally and implanted back to the original circuit. When applied appropriately, this approach can mitigate the problem of scalability of representation. Two complementary approaches to the extraction of the sub-circuits are presented and evaluated in this work. The evaluation is done on a set of highly optimized complex benchmark problems representing various real-world controllers, logic and arithmetic circuits. The experimental results show that the evolutionary resynthesis provides better results compared to globally operating evolutionary optimization. In more than 85% cases, a substantially higher number of redundant gates was removed while keeping the computational effort at the same level. A huge improvement was achieved especially for the arithmetic circuits. On the average, the proposed method was able to remove 25,1% more gates.

Contribution Local resynthesis is an iterative process based on the extraction of smaller sub-circuits from a complex circuit that are optimized locally and implanted back to the original circuit. This work enhances the effectivity of the local evolutionary resynthesis proposed previously. A new extraction method, based on the windowing algorithm, is presented in this work. Also, the experimental results are compared with the evolutionary method working with the whole combinational circuits and with the results produced by the state-of-the-art synthesis tool ABC.

This work proposes a combination of subcircuit selection and evolutionary optimization as a possible solution to the scalability of fitness evaluation of the EAs. Two different methods of subcircuit extraction were proposed. The first one is based on computing so-called cuts. The second one has its origin in the windowing algorithm. During the experimental phase of both methods, the subcircuits were chosen randomly and then evolutionarily optimized by CGP in each iteration of the optimization. Despite using such a naive approach to the subcircuit selection, in more than 85% cases, substantially higher number of redundant gates was removed compared to the CGP operating globally with the whole circuits.

3.2.4 Paper IV

J. Kocnová and Z. Vasicek, „Resynthesis of logic circuits using machine learning and reconvergent paths,“ 2021 24th Euromicro Conference on Digital System Design (DSD), Palermo, Italy, 2021, pp. 69-76, doi: 10.1109/DSD53832.2021.00020.

Author participation: 50%
Conference rank: B1 (Qualis)

Abstract Boolean network scoping represents a common approach incorporated in conventional synthesis tools for maintaining good scalability of the synthesis process. Recently, an approach to the local resynthesis based on combination of evolutionary optimization with the principle of Boolean network scoping has been proposed. Local resynthesis is an iterative process based on the extraction of smaller sub-circuits from a complex circuit that are optimized locally and implanted back to the original circuit. The main advantage of the local resynthesis is that it can mitigate the problem of scalability of representation which is typical to the evolutionary algorithms as the efficiency of the evolutionary optimization applied at the global level deteriorates with the increasing circuit complexity. Unfortunately, the efficiency of local resynthesis depends on the efficiency of the sub-circuit extraction process. We propose an alternative method, based on the reconvergent paths. The evaluation is performed on a set of highly optimized benchmark problems representing various real-world controllers, logic and arithmetic circuits. The method provides better results compared to the state-of-the-art logic synthesis tool and evolutionary optimization techniques operating locally and globally. A substantially higher number of redundant gates was removed in more than 70% cases, while keeping the computational effort at the same level. A huge improvement was achieved especially for the controllers. On average, the proposed method was able to remove more than 14.3% of gates. The highest achieved gate reduction was more than 45% of gates.

Contribution Recently proposed approach to the local resynthesis based on a combination of Boolean network scoping and evolutionary optimization overcame the problem of scalability of the internal circuit representation as well as the scalability of fitness computation. However, the success of such an approach relies mostly on the efficiency of the sub-circuit extraction process. This work proposes a new method of sub-circuit selection based on the reconvergent path identification. It is a modification to the previously proposed windowing-based selection method. Firstly, a reconvergent path is determined and then a window is built up arround this particular circuit area. This increases the chance of having don't-cares in the window and so the probability of a good optimization result also increases. This method is evaluated on the set of highly optimized benchmark circuits and compared to the previously proposed optimization methods using modified cut computation and windowing-based algorithm for circuit extraction.

This work proposes a new method of sub-circuit selection based on reconvergent path identification. Presence of such a path in the selection increases the chance of a good optimization result based on a high don't-cares occurence in the selection. As the experimental results show, the proposed method outperformed the previously introduced methods based on cut-computation and windowing in 22 out of 28 cases. Compared to the globally working EA-based optimization, the proposed method won in 26 out of 28 cases. The overall gate reduction was 14%.

3.2.5 Paper V

J. Kocnová and Z. Vasicek, „Delay-aware evolutionary optimization of digital circuits,“ 2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), Nicosia, Cyprus, 2022, pp. 188-193, doi: 10.1109/ISVLSI54635.2022.00045.

Author participation: 50%
Conference rank: B1 (Qualis)

Abstract In the recent years, machine learning techniques have successfully been applied in various fields of digital circuit development including logic synthesis. One of the approaches is evolutionary resynthesis. It is based on the idea of iterative local optimization of parts of the original circuit. The main advantage is the possibility to mitigate various scalability issues connected with the usage of evolutionary algorithms. However, success of this method depends heavily on the ability to identify suitable candidates for local optimization. Despite that, it has been shown that the local optimization produces significantly compact solutions compared to the evolutionary optimization performed at the level of the original circuit. In this paper, we analyze how the local optimization affects the delay of the circuit and propose a modified approach to the optimization of digital circuits. Compared to the existing techniques, the proposed method allows the presence of the non-uniform delay at the inputs of the circuits selected for the local optimization. This modification enables to maintain the delay of the optimized circuit at a reasonable level without significant overhead. The evaluation done on a set of non-trivial highly optimized benchmark circuits representing various real-world circuits demonstrated that our method is able to remove a significant number of gates while preserving the delay at the original value.

Contribution Evolutionary resynthesis, based on the iterative optimization of parts of the original circuit, is able to overcome several scalability issues connected to the EAs. This work focuses on how the local optimization affects the delay of the optimized circuit. Previously proposed method of the subcircuit selection based on windowing and reconvergent paths are modified in order to cope with the non-uniform delay present on the inputs of the subcircuits. This modification allows the optimization process to keep the circuit's delay on a desired value while still being able to reduce the number of gates by a significant amount. It means that the designer can specify the maximum allowed increase to the circuit's delay and thus keeping the depth changes under control.

This work addresses the possible depth degradation during circuit optimization by limiting the optimization process with the desired depth of the optimized circuit. This change to the existing windowing and reconvergent path based methods showed that it is possible to achieve a good optimization result in means of number of gates while allowing quite a small increase of the circuit's depth. The maximum number of removed gates was more than 35% while extending the circuit's depth by only a 20% of its original value.

3.3 List of Other Papers

- Jitka Kocnova and Zdenek Vasicek. 2019. Impact of subcircuit selection on the efficiency of CGP-based optimization of gate-level circuits. In Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO '19). Association for Computing Machinery, New York, NY, USA, 377–378.

Author participation: 50%

Conference rank: N/A

- Jitka Kocnova and Zdenek Vasicek. EA-based Optimization of Digital Circuits. In Proceedings of the 8th Prague Embedded Systems Workshop (PESW 2020). Czech Technical University in Prague. ISBN 978-80-01-06772-7

  Author participation: 50%

  Conference rank: N/A

Chapter 4

Discussion and Conclusions

This chapter summarizes the research presented in this thesis and discusses open questions that could be addressed in future work.

This thesis presented the research focused on EA-based resynthesis of complex combinational circuits. The work started with a study of the state-of-the-art circuit synthesis and optimization together with the strengths and weaknesses of the most commonly used internal circuit representations, such as AIG. Separation of the synthesis and technology mapping into two different steps causes a significant inefficiency as all of the gates may not be recognized properly. Moreover, evolutionary algorithms were studied, with focus on the CGP. It is a powerful EA algorithm, however its performance deteriorates with the increasing size of networks.

To target the sources of inefficiency identified in our preliminary research, the EA-based resynthesis module was developed. It combined a cut-based sub-circuit selection method and CGP representing the evolutionary optimization. The knowledge gained from this initial research can be found in Chapter 3.1.4 and Paper I and Paper II. In order to improve the sub-circuit extraction, the windowing-based selection method was implemented and evaluated w.r.t. the previously mentioned methods (see Paper III). Reconvergent-paths based selection method was then proposed to increase the possibility of don't-cares in the sub-circuit. The experimental results can be seen in the Paper IV. In the last phase, implementation of the delay-aware extraction method was targeted. This was done by a modification to the windowing-based and reconvergent-paths based extraction methods, so that the resynthesis allowed only such modifications to the circuit that did not extend its delay more than desired. In Paper V, experimental evaluation of this method can be found.

In order to provide a fair and consistent evaluation, all of the proposed methods were evaluated on the same set of highly-optimized real-world mapped combinational circuits. Each method was also compared to the state-of-the-art synthesis tool (ABC) and CGP working with the whole circuits without extracting sub-circuits. Interestingly, as the experimental results demonstrated, each of the proposed methods reached a much better results than the ABC and globally working CGP. The obtained results confirmed the hypothesis that the circuit resynthesis based on the iterative sub-circuit selection and subsequent evolutionary evolution of every sub-circuit can achieve a significantly better gate reduction of the circuit compared to the optimization applied to the whole circuit.

4.1 Future Work

Future research topics were identified based on the experiences and experimental evaluation:

- In this thesis, the number of CGP generations was fixed to ensure fair evaluation. A strategy that adapts the maximum number of generations according to a circuit parameter, such as the size of the optimized sub-circuit, could be introduced. This modification could potentially improve the runtime of the resynthesis as it could prevent optimizing sub-circuits that have a low chance of being further reduced.
- Also, an adaptive strategy that identifies the optimal number of resynthesis iterations can improve the optimization efficiency. In this work, the number of resynthesis iterations was set to a fixed number for each of the proposed methods to ensure a fair evaluation. Usually, the resynthesis reached a stable state much sooner than the allowed number of iterations were executed. Terminating the computation before the computation of uneffective iterations would save computational time. On the other hand, this may stop the optimization slightly before reaching a significant gate removal that might be done few iterations later (e.g. due to selection of a suitable sub-circuit that has not yet been extracted).
- The root node of a sub-circuit is chosen randomly in this study because the root selection problem is NP complete problem itself. Intuitively, the root node selection is a non-trivial problem and it is necessary to set a suitable criterion that extracts the suitable root node candidates from all of the circuit's nodes. It is thus hypothesized that selecting the root node from a particular area or with a certain interconnections can improve the resynthesis result.
- In this thesis, each proposed method was executed independently during the experimental phase in order to examine its capabilities. As the results show, the no-free-lunch theorem is applied – eg. the windowing based method produced significantly better results than the CGP working on whole circuits; however there are situations where it is the other way around. Interlacing eg. the globally working CGP with the windowing-based method could improve the results and lead to a better convergence. This leads to the necessity of introducing an adaptive strategy that will decide on the basis of performance of currently applied method.

Bibliography

[1] Amaru, L., Gaillardon, P. E. and Micheli, G. D. MIXSyn: An efficient logic synthesis methodology for mixed XOR-AND/OR dominated circuits. In: *2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2013, p. 133–138.

[2] Amaru, L., Gaillardon, P. E. and Micheli, G. D. Majority-Inverter Graph: A New Paradigm for Logic Optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. 2016, vol. 35, no. 5, p. 806–819.

[3] Amaru, L., Gaillardon, P. E. and Micheli, G. D. Majority-Inverter Graph: A New Paradigm for Logic Optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. 2016, vol. 35, no. 5, p. 806–819.

[4] Amaru, L., Gaillardon, P.-E. and De Micheli, G. The EPFL Combinational Benchmark Suite. *Proc. of the 24th Int. Workshop on Logic and Synthesis*. 2015.

[5] Amarú, L., Soeken, M., Vuillod, P., Luo, J., Mishchenko, A. et al. Improvements to boolean resynthesis. In: *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2018, p. 755–760. DOI: 10.23919/DATE.2018.8342108.

[6] Brand. Redundancy and Don't Cares in Logic Synthesis. *IEEE Transactions on Computers*. 1983, C-32, no. 10, p. 947–952. DOI: 10.1109/TC.1983.1676139.

[7] Brayton, R. and Mishchenko, A. ABC: An Academic Industrial-Strength Verification Tool. In: *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, p. 24–40. ISBN 978-3-642-14295-6.

[8] Bäck, T., Hoffmeister, F. and Schwefel, H.-P. A Survey of Evolution Strategies. In:. January 1991, p. 2–9.

[9] Çalik, c., Sönmez Turan, M. and Peralta, R. The Multiplicative Complexity of 6-Variable Boolean Functions. *Cryptography Commun.* Berlin, Heidelberg: Springer-Verlag. 2019, vol. 11, no. 1, p. 93–107. DOI: 10.1007/s12095-018-0297-2. ISSN 1936-2447. Available at: https://doi.org/10.1007/s12095-018-0297-2.

[10] Chatterjee, S., Mishchenko, A., Brayton, R. K., Wang, X. and Kam, T. Reducing Structural Bias in Technology Mapping. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. 2006, vol. 25, no. 12, p. 2894–2903. DOI: 10.1109/TCAD.2006.882484.

[11] Coello, C. C. A., Christiansen, A. D. and Aguirre, A. H. Automated Design of Combinational Logic Circuits by Genetic Algorithms. In: *Artificial Neural Nets and Genetic Algorithms: Proceedings of the International Conference in Norwich, U.K., 1997*. Vienna: Springer Vienna, 1998, p. 333–336. DOI: 10.1007/978-3-7091-6492-1_73. ISBN 978-3-7091-6492-1.

[12] Conceição, C., Moura, G., Pisoni, F. and Reis, R. A cell clustering technique to reduce transistor count. In: *2017 24th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. 2017, p. 186–189. DOI: 10.1109/ICECS.2017.8291996.

[13] Conceição, C. M. d. O. and Reis, R. A. d. L. Transistor Count Reduction by Gate Merging. *IEEE Transactions on Circuits and Systems I: Regular Papers*. 2019, vol. 66, no. 6, p. 2175–2187. DOI: 10.1109/TCSI.2019.2907722.

[14] Darringer, J. A., Joyner, W. H., Berman, C. L. and Trevillyan, L. Logic Synthesis Through Local Transformations. *IBM Journal of Research and Development*. 1981, vol. 25, no. 4, p. 272–280. DOI: 10.1147/rd.254.0272.

[15] Ellouz, S., Gamand, P., Kelma, C., Vandewiele, B. and Allard, B. Combining Internal Probing with Artificial Neural Networks for Optimal RFIC Testing. In: *2006 IEEE International Test Conference*. 2006, p. 1–9. DOI: 10.1109/TEST.2006.297705.

[16] Fiser, P., Halecek, I. and Schmidt, J. Are XORs in logic synthesis really necessary? In: *IEEE 20th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*. 2017, p. 138–143.

[17] Fiser, P., Halecek, I. and Schmidt, J. SAT-Based Generation of Optimum Function Implementations with XOR Gates. In: *2017 Euromicro Conference on Digital System Design (DSD)*. 2017, p. 163–170.

[18] Fiser, P., Schmidt, J., Vasicek, Z. and Sekanina, L. On logic synthesis of conventionally hard to synthesize circuits using genetic programming. In: *13th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems*. 2010, p. 346–351.

[19] Fiser, P. and Schmidt, J. Small but Nasty Logic Synthesis Examples. In: *Proc. 8th Int. Workshop on Boolean Problems*. 2008, p. 183–190.

[20] Fiser, P. and Schmidt, J. The Observed Role of Structure in Logic Synthesis Examples. In: *18th Int. Workshop on Logic and Synthesis*. 2009, p. 210–213.

[21] Fiser, P. and Schmidt, J. It Is Better to Run Iterative Resynthesis on Parts of the Circuit. In: *Proc. of the 19th Int. Workshop on Logic and Synthesis*. Univ. of California Irvine, 2010, p. 17–24.

[22] Frenzel, J. Genetic algorithms. *IEEE Potentials*. 1993, vol. 12, no. 3, p. 21–24. DOI: 10.1109/45.282292.

[23] Goldman, B. W. and Punch, W. F. Analysis of cartesian genetic programming's evolutionary mechanisms. *IEEE Transactions on Evolutionary Computation*. IEEE. 2015, vol. 19, no. 3, p. 359–373.

[24] Goldman, B. and Punch, W. Reducing wasted evaluations in cartesian genetic programming. *Lecture Notes in Computer Science*. Vienna: [b.n.]. 2013, 7831 LNCS, p. 61–72. DOI: 10.1007/978-3-642-37207-0_6. ISSN 03029743.

[25] Gordon, T. G. W. and Bentley, P. J. On evolvable hardware. In: *Soft Computing in Industrial Electronics*. London, UK: Physica-Verlag, 2002, p. 279–323.

[26] Grosnit, A., Malherbe, C., Tutunov, R., Wan, X., Wang, J. et al. BOiLS: Bayesian Optimisation for Logic Synthesis. In: *2022 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2022, p. 1193–1196. DOI: 10.23919/DATE54114.2022.9774632.

[27] Haaswijk, W., Soeken, M., Amaru, L., Gaillardon, P. E. and Micheli, G. D. A novel basis for logic rewriting. In: *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2017, p. 151–156.

[28] Higuchi, T., Niwa, T., Tanaka, T., Iba, H., Garis, H. de et al. Evolving Hardware with Genetic Learning: A First Step Towards Building a Darwin Machine. In: *Proc. of the 2nd International Conference on Simulated Adaptive Behaviour*. MIT Press, 1993, p. 417–424.

[29] Huang, G., Hu, J., He, Y., Liu, J., Ma, M. et al. Machine Learning for Electronic Design Automation: A Survey. *ACM Trans. Des. Autom. Electron. Syst.* New York, NY, USA: Association for Computing Machinery. jun 2021, vol. 26, no. 5. DOI: 10.1145/3451179. ISSN 1084-4309. Available at: https://doi.org/10.1145/3451179.

[30] Kinnear, K. Fitness landscapes and difficulty in genetic programming. In: *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*. 1994, p. 142–147 vol.1. DOI: 10.1109/ICEC.1994.350026.

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

[32] Kuehlmann, A., Paruthi, V., Krohm, F. and Ganai, M. Robust Boolean reasoning for equivalence checking and functional property verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. 2002, vol. 21, no. 12, p. 1377–1394. DOI: 10.1109/TCAD.2002.804386.

[33] Li, N. and Dubrova, E. AIG rewriting using 5-input cuts. In: *Proc. of the 29th Int. Conf. on Computer Design*. IEEE CS, 2011, p. 429–430.

[34] Lohn, J. D. and Hornby, G. S. Evolvable Hardware: Using Evolutionary Computation to Design and Optimize Hardware Systems. *IEEE Computational Intelligence Magazine*. 2006, vol. 1, no. 1, p. 19–27.

[35] Miller, J. and Thomson, P. Cartesian Genetic Programming. In: *Proc. of the 3rd European Conference on Genetic Programming EuroGP2000*. Springer, 2000, vol. 1802, p. 121–132. LNCS.

[36] Miller, J. D. . V. V. Principles in the Evolutionary Design of Digital Circuits. In: *Genetic Programming and Evolvable Machines 1*. 2000, p. 7–35 vol.1.

[37] Miller, J. F. Digital Filter Design at Gate-level Using Evolutionary Algorithms. In: *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 1999*. Morgan Kaufmann, 1999, p. 1127–1134.

[38] Miller, J. F. An Empirical Study of the Efficiency of Learning Boolean Functions Using a Cartesian Genetic Programming Approach. In: *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation - Volume 2*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, p. 1135–1142. GECCO'99. ISBN 1558606114.

[39] Miller, J. F. *Cartesian Genetic Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. 17–34 p. ISBN 978-3-642-17310-3. Available at: https://doi.org/10.1007/978-3-642-17310-3_2.

[40] Miller, J. F., Thomson, P. and Fogarty, T. Designing Electronic Circuits Using Evolutionary Algorithms. Arithmetic Circuits: A Case Study. In: *Genetic Algorithms and Evolution Strategies in Engineering and Computer Science*. Wiley, 1997, p. 105–131.

[41] Miller, J. F. Cartesian genetic programming: its status and future. *Genetic Programming and Evolvable Machines*. Aug 2019. DOI: 10.1007/s10710-019-09360-6. ISSN 1573-7632.

[42] Mishchenko, A., Chatterjee, S. and Brayton, R. DAG-aware AIG rewriting: a fresh look at combinational logic synthesis. In: *2006 43rd ACM/IEEE Design Automation Conference*. July 2006, p. 532–535. DOI: 10.1145/1146909.1147048. ISSN 0738-100X.

[43] Mishchenko, A. and Brayton, R. Scalable Logic Synthesis using a Simple Circuit Structure. In: *Int. Workshop on Logic and Synthesis*. 2006, p. 15–22.

[44] Rai, S., Neto, W. L., Miyasaka, Y., Zhang, X., Yu, M. et al. Logic Synthesis Meets Machine Learning: Trading Exactness for Generalization. *ArXiv preprint arXiv:2012.02530*. 2020.

[45] Reis, R. Trends on EDA for low power. In: *2015 IEEE MTT-S International Conference on Numerical Electromagnetic and Multiphysics Modeling and Optimization (NEMO)*. 2015, p. 1–4. DOI: 10.1109/NEMO.2015.7415104.

[46] Riener, H., Haaswijk, W., Mishchenko, A., De Micheli, G. and Soeken, M. On-the-fly and DAG-aware: Rewriting Boolean Networks with Exact Synthesis. In: *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2019, p. 1649–1654. DOI: 10.23919/DATE.2019.8715185.

[47] Sato, H., Yasue, Y., Matsunaga, Y. and Fujita, M. Boolean Resubstitution with Permissible Functions and Binary Decision Diagrams. In: *Proceedings of the 27th ACM/IEEE Design Automation Conference*. New York, NY, USA: Association for Computing Machinery, 1991, p. 284–289. DAC '90. DOI: 10.1145/123186.123276. ISBN 0897913639. Available at: https://doi.org/10.1145/123186.123276.

[48] Sekanina, L., Ptak, O. and Vasicek, Z. Cartesian Genetic Programming as Local Optimizer of Logic Networks. In: *2014 IEEE Congress on Evolutionary Computation*. IEEE CIS, 2014, p. 2901–2908.

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

[50] Shanthi, A. P. and Parthasarathi, R. Practical and scalable evolution of digital circuits. *Applied Soft Computing.* 2009, vol. 9, no. 2, p. 618–624.

[51] Stomeo, E., Kalganova, T. and Lambert, C. Generalized Disjunction Decomposition for Evolvable Hardware. *IEEE Transaction Systems, Man and Cybernetics, Part B.* 2006, vol. 36, no. 5, p. 1024–1043.

[52] Tao, Y., Zhang, L. and Zhang, Y. A projection-based decomposition for the scalability of evolvable hardware. *Soft Computing.* Jun 2016, vol. 20, no. 6, p. 2205–2218. DOI: 10.1007/s00500-015-1636-2. ISSN 1433-7479.

[53] Tavares, R., Meinhardt, C. and Reis, R. OrBDDs Direct Mapping for Structured Logic Circuits. In: *2006 13th IEEE International Conference on Electronics, Circuits and Systems.* 2006, p. 1057–1060. DOI: 10.1109/ICECS.2006.379620.

[54] Testa, E., Amarú, L., Soeken, M., Mishchenko, A., Vuillod, P. et al. Extending Boolean Methods for Scalable Logic Synthesis. *IEEE Access.* 2020, vol. 8, p. 226828–226844. DOI: 10.1109/ACCESS.2020.3045014.

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

[56] Vasicek, Z. Cartesian GP in Optimization of Combinational Circuits with Hundreds of Inputs and Thousands of Gates. In: *Proceedings of the 18th European Conference on Genetic Programming – EuroGP.* Springer International Publishing, 2015, p. 139–150. LCNS 9025.

[57] Vasicek, Z. and Sekanina, L. Formal Verification of Candidate Solutions for Post-Synthesis Evolutionary Optimization in Evolvable Hardware. *Genetic Programming and Evolvable Machines.* 2011, vol. 12, no. 3, p. 305–327.

[58] Vassilev, V., Job, D. and Miller, J. F. Towards the Automatic Design of More Efficient Digital Circuits. In: Lohn, J., Stoica, A., Keymeulen, D. and Colombano, S., ed. *Proc. of the 2nd NASA/DoD Workshop on Evolvable Hardware.* Los Alamitos, CA, USA: IEEE Computer Society, 2000, p. 151–160.

[59] Vemuri, N., Kalla, P. and Tessier, R. BDD-based Logic Synthesis for LUT-based FPGAs. *ACM Trans. Des. Autom. Electron. Syst.* New York, NY, USA: ACM. october 2002, vol. 7, no. 4, p. 501–525. ISSN 1084-4309.

[60] Ward, S., Ding, D. and Pan, D. Z. PADE: A high-performance placer with automatic datapath extraction and evaluation through high-dimensional data learning. In: *DAC Design Automation Conference 2012.* 2012, p. 756–761. DOI: 10.1145/2228360.2228497.

[61] Wolf, C., Glaser, J. and Kepler, J. Yosys-a free Verilog synthesis suite. In: *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip).* 2013.

[62] Yang, C. and Ciesielski, M. BDS: a BDD-based logic optimization system. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*. Jul 2002, vol. 21, no. 7, p. 866–876.

[63] Zhao, S. and Jiao, L. Multi-objective evolutionary design and knowledge discovery of logic circuits based on an adaptive genetic algorithm. *Genetic Programming and Evolvable Machines*. 2006, vol. 7, no. 3, p. 195–210.

Related Papers

Appendix A

Towards a Scalable EA-based Optimization of Digital Circuits

KOCNOVA Jitka and VASICEK Zdenek

In: *European Conference on Genetic Programming 2019*. Berlin: Springer International Publishing, LNCS 11451, 2019, pp. 81–97. ISBN 978-3-030-16669-4.

Towards a Scalable EA-based Optimization of Digital Circuits

Jitka Kocnova and Zdenek Vasicek[0000−0002−2279−5217]

Brno University of Technology, Faculty of information Technology,
IT4Innovations Centre of Excellence,
Brno, Czech Republic,
Email: ikocnova@fit.vutbr.cz, vasicek@fit.vutbr.cz

Abstract. Scalability of fitness evaluation was the main bottleneck preventing adopting the evolution in the task of logic circuits synthesis since early nineties. Recently, various formal approaches have been introduced to this field to overcome this issue. This made it possible to optimise complex circuits consisting of hundreds of inputs and thousands of gates. Unfortunately, we are facing to the another problem – scalability of representation. The efficiency of the evolutionary optimization applied at the global level deteriorates with the increasing complexity. In this paper, we propose to apply the concept of local resynthesis. Resynthesis is an iterative process based on extraction of smaller sub-circuits from a complex circuit that are optimized locally and implanted back to the original circuit. When applied appropriately, this approach can mitigate the problem of scalability of representation. Our evaluation on a set of non-trivial real-world benchmark problems shows that the proposed method provides better results compared to global evolutionary optimization. In more than 60% cases, substantially higher number of redundant gates was removed while keeping the computational effort at the same level.

Keywords: Cartesian Genetic Programming · Resynthesis · Logic optimization.

1 Introduction

Logic synthesis, as understood by the hardware community, is a process that transforms a high-level description into a gate-level or transistor-level implementation. Due to the complexity of the problem, the synthesis process is typically broken into a sequence of steps. Among others, logic optimization represents an important part of the whole process. The goal of the logic optimization is to transform a suboptimal solution into an optimal gate-level implementation with respect to given synthesis goals. Due to the scalability issues, the problem is typically represented using a suitable internal representation. Current state-of-the-art logic synthesis tools, such as ABC, represent circuits using a directed acyclic graph composed of two-input AND nodes connected by direct or negated edges denoted as and-inverter graph (AIG). The optimization of AIGs is based on *rewriting*, a greedy algorithm which minimizes size of AIG by iteratively selecting subgraphs rooted at a node and replacing them with smaller precomputed subgraphs, while preserving the functionality of the root node [1]. AIG rewriting is local, however, the scope of changes becomes global by application of rewriting many times. In addition to that, *resubstitution* and *refactoring* can be employed. Resubstitution expresses the function of a node using other nodes present in the AIG [2]. Refactoring iteratively selects large cones of logic rooted at a node and tries to replace them with a more efficient implementation [1]. Refactoring can be seen as a variant of rewriting. The main difference is that rewriting selects subgraphs containing few leaves because the number of leaves determines the number of variables of a Boolean function whose optimal implementation is sought.

The AIG representation is simple and scalable, and leads to simple algorithms but it suffers from an inherent bias in representation. While eight of ten possible two-input logic gates may be represented by means of a single AIG node, XOR and XNOR gate require three AIG nodes each. The efficiency of synthesis is then limited as it mostly fully relies on transformations that disallow an increase the number of AIG nodes. It has been shown that there exists a huge class of real-world circuits for which the synthesis fails and provides very poor results [3–5]. In some cases, the area of the synthesized circuits is of orders of magnitude higher than the known optimum. If a large design is broken down to multiple smaller circuits and such a failure occurs during resynthesis, we obtain an unacceptably large circuit.

Various evolutionary approaches working directly at the level of gates were successfully applied to address this problem [3, 6]. Vasicek demonstrated that the evolutionary synthesis using Cartesian Genetic Programming (CGP) conducted directly at the level of common gates is able to provide significantly better results compared to the state-of-the-art synthesis operating on AIGs [6]. On average, the method enabled a 34% reduction in gate count on an extensive set of benchmark circuits when executed for 15 minutes. It was observed, however, that the efficiency of the evolutionary approach deteriorates with an increasing number of gates. Substantially more generations were required to reduce circuits consisting of more than ten thousands gates. While [6] focuses strictly on the improvement of the scalability of the evaluation, Sekanina et al. employed a divide and conquer strategy to address the problem of scalability of representation [3]. The authors were able to obtain better results than other locally operating methods reported in the literature, however, the performance of this method was significantly worse than the evolutionary global optimization proposed in [6].

In order to improve the results of EA-based synthesis, we propose to combine the EA-based approach with refactoring while following the principle of local resynthesis applied in common logic synthesis tools. Firstly, a logic circuit is optimized by means of a common synthesis approach. Then, the optimized circuit is mapped to standard gates and optimized using the proposed method that extracts a relatively small sub-circuits that are subsequently optimized by Cartesian Genetic Programming (CGP). The original sub-circuit is then replaced by its optimized variant provided that there is an improvement at the global level and the whole process is repeated. Our approach is based on iterative optimization of large portions of the original circuit. Compared to rewriting, we do not impose any limitation on the number of leaves because the larger subgraphs offer more opportunities for potential area improvement.

2 Background

This section presents relevant background on conventional as well as EA-based optimization of logic circuits and introduces the notation used in the rest of the paper.

2.1 Boolean networks

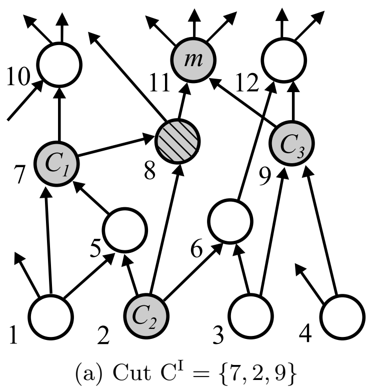
Every circuit can be represented using a Boolean network. A *Boolean network* is a directed acyclic graph (DAG) with nodes represented by Boolean functions [2]. The sources of the graph are the primary inputs (PIs) of the network and the sinks are the primary outputs (POs). The output of a node may be an input to other nodes called *fanouts*. The inputs of a node are called *fanins*. An edge connects two nodes that are in fanin/fanout relationship. Considering this notion, And-Inverter Graph is a Boolean network composed of two-input ANDs and inverters. The network primary inputs are signals that are driven by the environment, there is no node driving these signals in the network. Similarly, the primary outputs are signals that drive the environment and are needed by inner network nodes as well. The size of the network is the number of the nodes (primary inputs and outputs are not considered).

2.2 Limiting the scope of Boolean networks

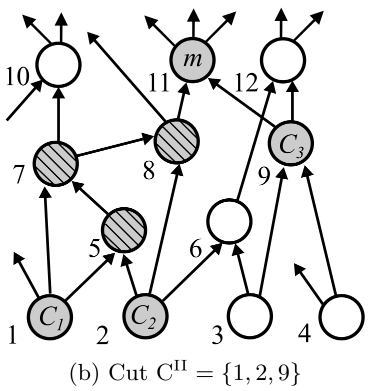
Network scoping represents a key operation to ensure a good scalability of synthesis tools when working with large Boolean networks. In addition, it forms an integral part of rewriting as well as refactoring. Two approaches have been proposed to limit the scope of logic synthesis to work only on a small portion of a Boolean network – *windowing* and *cut computation* [2].

The windowing algorithm determining the window for a given node takes a node and two integers defining the number of logic levels on the fanin/fanout sides of the node to be included in the window. Two sets are produced as the result of windowing – leaf set and root set. The window of a Boolean network is the subset of nodes of the network containing nodes from root set together with all nodes on paths between the leaf set and the root set. The nodes in the leaf set are not included in the window. The main problem of this algorithm is that it is hard to predict how many logic levels have to be traversed to get a window of the desired size and required number of leaves.

A complementary approach based on computing so called k-feasible cuts is usually preferred to avoid determining the required number of logic levels. A cut of a node, called root node, is a set of nodes of the network, called leaves, such that each path from PI to the root node passes through at least one leaf. A cut is k-feasible if the number of nodes (i.e. cut size) in the cut does not exceed k. The volume of a cut is the total number of nodes encountered on all paths between the root node and the cut leaves. An example of two different 3-feasible cuts is shown in Fig. 1. To maximize the cut volume, a reconvergence-driven heuristic is applied in practice. The problem is that the cut computed using a naive bread-first-search algorithm may include only few nodes and leads to tree-like logic structures (see Fig. 1a showing a cut determined by the naive approach and Fig. 1b showing the output of reconvergence-driven heuristic). Such a structure does not lead to any don't cares in the local scope of the node and attempting optimization using such a cut would be wasted time. A simple and efficient cut computation algorithm producing a cut close to a given size while heuristically maximizing the cut volume and the number of reconvergent paths subsumed in the cut has been introduced in [2]. As our work is based on the reconvergence-driven cuts, we briefly discuss this algorithm. The algorithm starts with a set of leaves consisting of a single root node. This set is incrementally expanded by adding one node in each step of a recursive procedure. If the set consists of only PIs, the procedure quits. Otherwise, a non-PI node that minimizes a cost function is chosen from the set of leaves. The chosen node is removed from the leaf set and all its fanins are included instead of it. This causes expansion of the cut. If the cut-size limit is exceeded, the procedure quits and returns the cut before expansion. The cost function returns the number of new nodes that should be added to the leaf set instead of the removed node.



(a) Cut $C^{I} = \{7, 2, 9\}$



(b) Cut $C^{II} = \{1, 2, 9\}$

Fig. 1: Example of two possible 3-feasible cuts for root node m and given Boolean network. The cut C^{II} is preferred as its volume is four (root node m and contained nodes 5, 7, and 9). There is only one contained node (node 8) in the case of C^{I}.

The k-feasible cuts are important not only for the gate-level logic synthesis but also for FPGA-based synthesis as a k-feasible cut can be implemented as a

k-input LUT. For resubstitution and FPGA-based mapping, so called maximum fanout free cone (a subnetwork where no node in the cone is connected to a node not in the cone) is requested. It means that the cut-based scoping must always produce a single-output sub-circuits. Otherwise it would be impossible to replace the whole sub-circuit by a precomputed optimal implementation / a single LUT. Typically, 4-feasible and 5-feasible cuts are used for rewriting-based logic synthesis [2, 7]. Small k is used not only to make the cut enumeration possible but also to manage memory requirements to store the precomputed optimal implementations of all k-input Boolean functions. For FPGA-based mapping, 5-input and 6-input LUTs are used. Apart from the rewriting, the reconvergence-driven cuts have been applied to refactoring and resubstitution [2]. Typically, k is between 5 and 12 for refactoring depending on the computation effort allowed [2].

2.3 Evolutionary Synthesis of Logic Circuits

Evolutionary algorithms (EAs) have been used to synthesize logic circuits since late nineties [8, 9]. Miller et al., the author of Cartesian Genetic Programming (CGP) [10], is considered as a pioneer in the field of logic synthesis of gate-level circuits. He utilized his own variant of genetic programming to synthesize compact implementations of multipliers described by means of a behavioral specification [11]. Despite of many advantages of this unconventional technique, only small problem instances were typically addressed. To tackle the limited scalability, various decomposition strategies have been proposed. A good survey of the existing techniques is provided, for example, in [12]. The projection-based decomposition approaches such as [13] or [12] helped to increase the complexity of problem instances that can be solved by EAs. Despite of that, the gap between the complexity of problems addressed by EAs and in industry continued to widen as the advancements in technology developed. In 2011, the scalability of CGP has been significantly improved by introducing a SAT-based CGP. The SAT-based CGP replaces determining of Hamming distance done by exhaustive simulation with a modern SAT solver [14]. It exploits the fact that the candidate solutions must be functionally equivalent with their parent in logic optimization in order to be further accepted. In addition to that, it exploits the knowledge of differences between parental and candidate circuits. The efficiency of SAT-based method was further improved by combining a SAT solver with an adaptive high-performance circuit simulator used to quickly identify the potential functional non-equivalence. The most advanced SAT-based CGP employs a simulator that is driven by counterexamples produced by the SAT solver [6]. Neither the original nor the latter approach rely on a decomposition. The gate-level circuits are optimized directly.

Since its introduction, CGP remains the most powerful evolutionary technique in the domain of logic synthesis and optimization [9]. In this area, a linear form of CGP is preferred today. CGP models a candidate circuit having n_i PIs and n_o POs as a linear 1D array of n_n configurable nodes. Each node has n_a inputs and corresponds with a single gate with up to n_a inputs. The inputs can be connected either to the output of a node placed in the previous L columns or directly to PIs. This avoids a feedback. The function of a node can be chosen from a set of n_f functions. Depending on the function of a node, some of its inputs may become redundant. In addition to that, the fixed number of nodes n_n does not mean that all the nodes contribute to the POs. These key features allow redundancy and flexibility of CGP.

The candidate circuits are encoded as follows. Each PI as well as each node has associated an unique index. Each node is encoded using $n_a + 1$ integers $(x_1, \cdots, x_{n_a}, f)$ where the first n_a integers denote the indices of its fanins and the last integer determines the function of that node. Every candidate circuit is encoded using $n_n(n_a + 1) + n_o$ integers where the last n_o integers specify the indices corresponding with each PO.

CGP is a population oriented approach which operates with $1 + \lambda$ candidate solutions. The initial population is seeded by the original circuit ought to be optimized. Every new population consists of the best circuit chosen from the previous population and its λ offspring created using a mutation operator that randomly modifies up to h integers. Considering the CGP encoding, a single mutation causes either reconnection of a gate, reconnection of primary outputs or change in function of a gate. The selection of the individuals is typically based on a cost function (e.g. the number of active nodes). In the case that there are more individuals with the same score, the individual that has not served as a parent will be selected as the new parent. This procedure is typically repeated for a predefined number of iterations.

3 The proposed method

Let $\mathcal{C}$ be a combinational circuit described at the level of common gates represented by a Boolean network N consisting of $|N|$ nodes. Each node corresponds with a single gate in $\mathcal{C}$. The pseudo-code of the proposed optimization procedure based on evolutionary resynthesis is shown in Algorithm 1.

Firstly a node which may potentially lead to the best improvement of N is determined. Since the identification of this node itself is a nontrivial problem, some heuristic needs to be implemented. The size of transitive fan-in cone, level of the node or a more complex information can be used to determine the most suitable candidate. Then, a working area (window) is extracted from the Boolean network. This procedure starts with computation of the reconvergence-driven cut C as described in Section 2.2. From the practical reasons, is also beneficial to limit the size of C to be able to enumerate a large number of sub-circuits in a reasonable time. Hence, we can define four parameters: c_{min} and c_{max} restricting the volume of C ($c_{min} < c_{max}$), and k_{min} and k_{max} ($k_{min} \leq |C| \leq k_{max}$) limiting the size of cut (feasibility).

This step is followed by expansion of the cut C into a window W, i.e. expansion of the set of leaf nodes to a set of contained nodes. In addition to the nodes inside the cut, we should consider also all nodes that are not contained in the cut but have fanins inside the cut. Our expansion is similar to that employed in the resubstitution [2] where transitive fanout of C is considered, however, we do

Algorithm 1: EA-BASED RESYNTHESIS

```
Input: A Boolean network $N$
Output: Optimized network $N'$, $cost(N') \leq cost(N)$
1  $N' \leftarrow N$
2  while terminated condition not satisfied do
3      $m \leftarrow$ identify the best candidate root node $m \in N'$
4      $C \leftarrow$ ReconvergenceDrivenCut($m$)
5      $W \leftarrow$ ExpandCutToWindow($m$, $C$)
6      if $W$ is not a suitable candidate then
7          continue
8      $W' \leftarrow$ OptimizeNetworkUsingEA($W$)
9      if cost(($N' \setminus W) \cup W'$) < cost($N'$) then
10         $N' \leftarrow (N' \setminus W) \cup W'$
11 return $N'$
```

not impose any limit on the number of included nodes or their maximum level. The process of cut identification and the subsequent expansion is illustrated in Fig. 2.

During the expansion, three set of nodes are created: the set of internal nodes I, the set of leaves L and the set of root nodes R. L contains nodes that will serve as PIs of the temporary network used in the subsequent optimization. Similarly R contains nodes whose outputs have to be connected to POs. Note that R contains not only the root node m but also other nodes whose fanouts are outside of the window (see Fig. 2). It holds that $C \subseteq L$ since the expansion may cause that some leaves of C become a fanout of a node inside the window. Two situations can happen for a leaf node. If all fanins are inside the window, the leaf can be simply removed from L. Otherwise, all fanins of the original leaf node need to be added to L (the case of C_1 in Fig. 2). This procedure has to be repeated iteratively to ensure that there are no leaves having a fanin already included the window.

Resynthesis is then applied to the window. Each window that is not suitable for the subsequent optimization is skipped. The motivation is to eliminate execution of a relatively time-consuming resynthesis for the windows that are unlikely to lead to any improvement. The identification of the suitable windows can be based on the size of W (small windows are filtered out) or a combination of size of C and size of W (thin windows are filtered out). In addition to that, we can use the information about the difference among level of the root node and leaves of C.

The resynthesis is performed by means of the CGP. At the beginning, each node in the window is assigned an unique index and chromosome corresponding with the nodes in the window is created. This chromosome is then used to seed the initial population. The evolutionary optimization is executed for a limited number of iterations. The number of iterations should be determined heuristically. The more iterations are allowed, the higher improvement can be

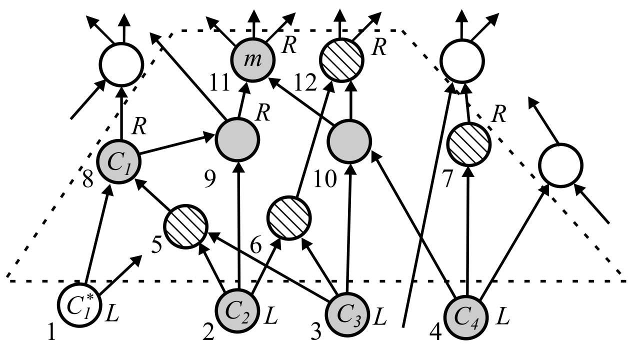


Fig. 2: Example of the window created using the proposed algorithm. The set of contained nodes of a 4-feasible cut C = $\{C_1, C_2, C_3, C_4\}$ rooted in node m is highlighted using the filled nodes. The hatched nodes are added to the window during the expansion of the cut. As a consequence of that, leave C_1 is replaced by C_1^*. The root and leaves of the window are denoted as R and L, respectively. The nodes in the window have assigned an index used to uniquely identify each node in the CGP. One of the many possibilities how to encode the window using CGP is for example: (2,3,AND) (2,3,OR) (4,1,INV) (1,5,XOR) (8,2,AND) (3,4,NOR) (9,10,AND) (6,10,OR) (7,8,9,11,12).

achieved. On the other hand, many iterations on a small window wastes time. Finally, the optimized logic network W' is evaluated w.r.t. N' and if it performs better, it replaces all non-leaf nodes included in W. The whole optimization algorithm is terminated when a predefined number of iterations or a given runtime is exhausted.

Table 1 compares the proposed method with various methods for optimization of logic circuits available in the literature. Compared to the conventional approaches, we consider windows consisting of substantially higher number of gates. In addition to that, we do not impose any limits on the number of window inputs and outputs. Compared to the evolutionary approach [3], substantially larger sub-circuits identified using a different scoping method (windowing based on reconvergence driven cuts) are considered during resynthesis.

4 Experimental evaluation

4.1 Experimental setup

The proposed method was implemented in C++ as a part of Yosys open synthesis suite [17]. The advantage of this tool, among others, is that it allows us to directly manipulate with Verilog files and that it integrates ABC [18], a state-of-the-art academic tool for hardware synthesis and verification.

To evaluate the proposed approach, we used 28 highly optimized real-world circuits and optimized them by means of the proposed as well as current state-of-the-art approach. Nineteen Verilog netlists are taken from IWLS'05 Open

Table 1: Comparison of the optimization approaches according to the applied constraints. Parameters k_{min} and k_{max} determine the minimum allowed and maximum acceptable number of inputs of the accepted windows. Parameters c_{min} and c_{max} represent the restrictions applied to the volume of the windows.

approach	k_{min}	k_{max}	c_{min}	c_{max}	scoping method
rewriting [15, 2]	–	4	–	–	cut computation
redundancy removal [2]	6	12	–	–	windowing
conventional resynthesis [16]	–	–	–	3360 (30%)	windowing (various)
evolutionary resynthesis [3]	1	10	8	50	radius-based windowing
proposed approach	–	–	10	10^4	cut-based windowing

Cores benchmarks, the remaining nine netlists represent various arithmetic circuits[1]. The circuits were optimized by ABC (several iterations of ABC command ‘resyn’) and mapped to gates using a library of common 2-input gates including XORs/XNORs gates (ABC command ‘map’). After mapping, optimization by the proposed and global method was executed and final number of mapped gates in circuits was examined. All of the optimized circuits were formally verified w.r.t their original form (ABC command ‘cec’).

The goal of this paper is to evaluate performance of the proposed method w.r.t. the state-of-the-art EA-based method (denoted as global) applied to the whole Boolean network and to compare both methods to the best result produced by the ABC. Both methods operate at the level of optimized and mapped Boolean networks to avoid the bias of AIG representation. The procedure OptimizeNetworkUsingEA is based on the CGP implemented as described in Section 2.3 with the following parameters: $n_a = 2$, $\lambda = 1$, $h = 2$, $n_n = |W|$. A single call of this procedure is executed for the global method (the procedure takes the whole Boolean network and returns its optimized version). On contrary, several calls of this procedure are executed in the proposed method. The termination conditions are designed as follows. The global method terminates when n_{iters} iterations are exhausted. One iteration corresponds with evaluation of a single candidate solution. In the case of the proposed method a simple divide-and-conquer strategy is employed. The proposed method is allowed to create n_{cuts} cuts. For each cut, the OptimizeNetworkUsingEA is allowed to perform n_{iters}/n_{cuts} iterations. This strategy is relatively naive because it supposes that the computation effort does not depend on the size of the window but it helps to fairly evaluate the impact of the proposed method. It ensures that exactly the same number of generations are evaluated in both cases. In this paper, we use $n_{iters} = 10^{10}$ iterations. Only windows whose volume is larger than 10 and less than 10^4 nodes are accepted, i.e. $c_{min} = 10$, $c_{max} = 10^4$. The root node m is chosen randomly in this study. This strategy simplifies the problem but it may lead to degradation of the performance especially if many unacceptable windows

[1] All the benchmarks are taken from https://lsi.epfl.ch/MIG

are produced. If this happens in 10% cases, for example, the total number of effective generations is in fact reduced to 90%. The only criterion in the fitness function considered in this paper is the area on a chip expressed as the number of gates. For each method and each benchmark, five independent runs were executed to obtain statistically reasonable results.

4.2 Experimental results

The overall results are summarized in Tab. 2. The first three columns contain information related to the benchmarks (name, number of PIs and POs). The next two columns show parameters of the mapped circuits and those numbers serve as a baseline for our comparison – the number of gates and logic depth is provided. Then, the achieved results expressed as the relative reduction with respect to the baseline are reported for the proposed and global method. For each method, we report the average and the best obtained improvement. The numbers are calculated across all independent runs.

The best results are very close to the average ones which suggests that the both EA-based methods are stable although they are in principle non-deterministic. According to the number of highlighted cases showing the better results, the proposed method performs substantially better considering the average as well as the best results. It won in 22 out of 28 cases. There are even cases, when the global method provided none or nearly none improvement (see e.g. benchmarks DSP, des_perf, ethernet, systemcaes and so on). The average reduction on the IWLS'05 benchmarks is slightly better in favor of the global method, but it is affected mostly by five cases (mem_ctrl, pci_spoci_ctrl, spi, systemcdes, and tv80), where the global method provided substantially better results. Looking at the arithmetic circuits, the global method was able to slightly improve only two circuits. In other cases, the reduction is negligible. We analyzed the five cases where the global method outperformed the proposed one and concluded that the global method works well especially for small instances (less than 10^4 gates) that have a reasonable depth (10 to 25 levels). The global optimization of circuits with large depth is unsatisfactory. A substantial improvement is achieved on the arithmetic circuits. The number of gates is reduced by nearly 15% on average. The highest reduction, 30.1%, is recorded for hamming benchmark. The detailed analysis revealed that this was possible due to better handling of XORs/XNORs compared to ABC and also by a relatively huge redundancy of the original circuit optimized by ABC. The relative number of AND/OR/NAND/NOR gates remained nearly the same (around 74%). The number of XORs/XNORs increased from 10% to 15%.

A more detailed analysis is shown in Tab. 3 where we reported the computational effort required to reduce the benchmark circuits by 1%, 5% and 10%. The table shows the mean number of generations that have to be evaluated to obtain a circuit whose number of gates is reduced by a given level. The empty cells mean that none of the evolutionary runs produced circuit satisfying the required condition. This can happen either because of the insufficient number of generations or because it is in principle impossible to obtain such a circuit (we

Table 2: Comparison of the proposed and global method against ABC. The columns ‘Impr. proposed‘ and ‘Impr. global‘ report the relative improvement in the number of gates compared to the optimized circuits obtained using ABC. Column ‘ABC‘ contains parameters of the optimized circuits after mapping (’gates’ is the number of gates, D is logic depth).

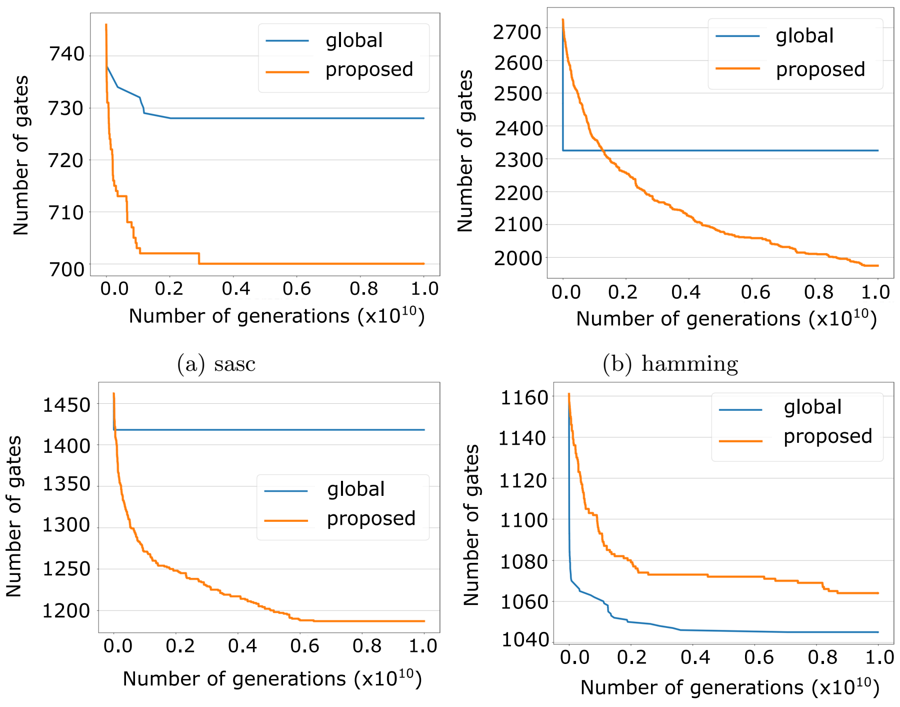
			ABC		Impr. proposed		Impr. global [6]	
Benchmark	PIs	POs	gates	D	avg	best	avg	best
DSP	4223	3792	43491	45	**3.6%**	**3.6%**	0.0%	0.0%
ac97_ctrl	2255	2136	11433	10	**2.9%**	**2.9%**	1.4%	1.4%
aes_core	789	532	21128	20	**2.9%**	**2.9%**	0.6%	1.7%
des_area	368	70	5199	25	**6.0%**	**6.1%**	2.1%	2.3%
des_perf	9042	1654	78972	16	**1.8%**	**1.8%**	0.0%	0.1%
ethernet	10672	10452	60413	23	**0.5%**	**0.5%**	0.0%	0.0%
i2c	147	127	1161	12	9.2%	9.2%	**10.0%**	**10.7%**
mem_ctrl	1198	959	10459	24	7.0%	7.0%	**24.8%**	**25.4%**
pci_bridge32	3519	3136	19020	21	**3.5%**	**3.5%**	0.5%	0.6%
pci_spoci_ctrl	85	60	1136	15	18.3%	18.5%	**34.8%**	**35.7%**
sasc	133	123	746	8	**6.2%**	**6.2%**	2.4%	2.8%
simple_spi	148	132	822	11	**5.5%**	**5.7%**	4.4%	4.6%
spi	274	237	3825	26	5.6%	5.6%	**13.5%**	**20.2%**
ss_pcm	106	90	437	7	**5.7%**	**6.7%**	2.3%	2.3%
systemcaes	930	671	11352	27	**11.9%**	**12.3%**	0.0%	0.0%
systemcdes	314	126	2601	25	4.8%	5.0%	**9.1%**	**9.9%**
tv80	373	360	8738	39	6.6%	6.9%	**11.1%**	**11.3%**
usb_funct	1860	1692	15405	23	**5.8%**	**5.9%**	2.6%	2.6%
usb_phy	113	73	452	9	**13.9%**	**14.0%**	12.2%	12.2%
average (IWLS’05 benchmarks)			15620	20	6.4%	6.5%	**7.0%**	**7.6%**
mult32	64	64	8225	42	**16.5%**	**16.6%**	0.0%	0.0%
sqrt32	32	16	1462	307	**22.3%**	**24.3%**	3.0%	3.0%
diffeq1	354	193	20719	218	**11.5%**	**11.5%**	0.0%	0.0%
div16	32	32	5847	152	**15.7%**	**15.8%**	0.0%	0.0%
hamming	200	7	2724	80	**28.6%**	**30.1%**	14.6%	14.6%
MAC32	96	65	7793	55	**7.7%**	**7.8%**	0.0%	0.0%
revx	20	25	8131	171	**14.5%**	**14.5%**	0.0%	0.1%
mult64	128	128	21992	190	**7.4%**	**7.4%**	0.3%	0.5%
max	512	130	3719	117	**5.3%**	**5.3%**	0.7%	0.8%
average (arithmetic benchmarks)			8956	148	**14.4%**	**14.8%**	2.1%	2.1%

are already at the optimum or close to the optimum). Looking at the first two columns showing the computation effort required for reduction by 1%, we can easily identify that the global method converges faster compared to the proposed method. On the other hand, it has tendency to stuck at a local optima which is evident especially on more complex benchmarks (arithmetic circuits having large logic depth and complex circuits consisting of tens thousands of gates). Nearly none improvement was achieved for arithmetic circuits. The only exception is

Table 3: The average number of CGP generations needed to achieve 1%, 5%, and 10% reduction

	1% improvement		5% improvement		10% improvement	
Benchmark	global	proposed	global	proposed	global	proposed
DSP	$> 10^{10}$	$\mathbf{8 \cdot 10^{8}}$	–	–	–	–
ac97_ctrl	$\mathbf{45 \cdot 10^{7}}$	$7 \cdot 10^{8}$	–	–	–	–
aes_core	$> 10^{10}$	$\mathbf{1 \cdot 10^{9}}$	–	–	–	–
des_area	$\mathbf{4 \cdot 10^{7}}$	$98 \cdot 10^{7}$	$> 10^{10}$	$\mathbf{11 \cdot 10^{8}}$	–	–
des_perf	$> 10^{10}$	$\mathbf{3 \cdot 10^{9}}$	–	–	–	–
i2c	$\mathbf{5 \cdot 10^{5}}$	$28 \cdot 10^{7}$	$\mathbf{35 \cdot 10^{5}}$	$5 \cdot 10^{8}$	$\mathbf{7 \cdot 10^{9}}$	$> 10^{10}$
mem_ctrl	$\mathbf{5 \cdot 10^{5}}$	$27 \cdot 10^{7}$	$\mathbf{5 \cdot 10^{5}}$	$45 \cdot 10^{8}$	$\mathbf{5 \cdot 10^{5}}$	$> 10^{10}$
pci_bridge32	$> 10^{10}$	$78 \cdot 10^{7}$	–	–	–	–
pci_spoci_ctrl	$\mathbf{5 \cdot 10^{5}}$	10^{7}	$\mathbf{5 \cdot 10^{5}}$	$14 \cdot 10^{7}$	$\mathbf{10^{6}}$	$42 \cdot 10^{7}$
sasc	$21 \cdot 10^{6}$	$\mathbf{15 \cdot 10^{5}}$	$> 10^{10}$	$\mathbf{43 \cdot 10^{6}}$	–	–
simple_spi	$\mathbf{5 \cdot 10^{6}}$	$86 \cdot 10^{6}$	$> 10^{10}$	$\mathbf{72 \cdot 10^{7}}$	–	–
spi	$\mathbf{5 \cdot 10^{6}}$	$416 \cdot 10^{6}$	$\mathbf{65 \cdot 10^{6}}$	$3 \cdot 10^{9}$	$\mathbf{72 \cdot 10^{6}}$	$> 10^{10}$
ss_pcm	$\mathbf{4 \cdot 10^{6}}$	10^{8}	$> 10^{10}$	$\mathbf{2 \cdot 10^{8}}$	–	–
systemcaes	$> 10^{10}$	$\mathbf{12 \cdot 10^{7}}$	$> 10^{10}$	$\mathbf{17 \cdot 10^{8}}$	$> 10^{10}$	$\mathbf{7 \cdot 10^{9}}$
systemcdes	$65 \cdot 10^{5}$	$\mathbf{17 \cdot 10^{7}}$	$\mathbf{55 \cdot 10^{6}}$	$> 10^{10}$	$\mathbf{74 \cdot 10^{7}}$	$> 10^{10}$
tv80	$\mathbf{5 \cdot 10^{5}}$	$231 \cdot 10^{6}$	$\mathbf{26 \cdot 10^{6}}$	$3 \cdot 10^{9}$	$\mathbf{18 \cdot 10^{7}}$	$> 10^{10}$
usb_funct	$\mathbf{94 \cdot 10^{6}}$	$575 \cdot 10^{6}$	$> 10^{10}$	$\mathbf{65 \cdot 10^{8}}$	–	–
usb_phy	$\mathbf{5 \cdot 10^{5}}$	$12 \cdot 10^{6}$	$\mathbf{25 \cdot 10^{5}}$	$19 \cdot 10^{6}$	$55 \cdot 10^{7}$	$\mathbf{17 \cdot 10^{7}}$
average	$2.8 \cdot 10^{9}$	$5.2 \cdot 10^{8}$	$4.6 \cdot 10^{9}$	$2.4 \cdot 10^{9}$	$3 \cdot 10^{9}$	$7.2 \cdot 10^{9}$
mult32	$> 10^{10}$	$\mathbf{72 \cdot 10^{6}}$	$> 10^{10}$	$\mathbf{48 \cdot 10^{7}}$	$> 10^{10}$	$\mathbf{2 \cdot 10^{9}}$
sqrt32	$\mathbf{5 \cdot 10^{5}}$	$19 \cdot 10^{6}$	$\mathbf{37 \cdot 10^{5}}$	$11 \cdot 10^{7}$	$> 10^{10}$	$\mathbf{39 \cdot 10^{7}}$
diffeq1	$> 10^{10}$	$\mathbf{2 \cdot 10^{8}}$	$> 10^{10}$	$\mathbf{16 \cdot 10^{8}}$	$> 10^{10}$	$\mathbf{67 \cdot 10^{8}}$
div16	$> 10^{10}$	$\mathbf{13 \cdot 10^{7}}$	$> 10^{10}$	$\mathbf{5 \cdot 10^{8}}$	$> 10^{10}$	$\mathbf{24 \cdot 10^{8}}$
hamming	$\mathbf{5 \cdot 10^{5}}$	$17 \cdot 10^{6}$	$\mathbf{5 \cdot 10^{5}}$	$12 \cdot 10^{7}$	$\mathbf{2 \cdot 10^{6}}$	$5 \cdot 10^{8}$
MAC32	$> 10^{10}$	$\mathbf{6 \cdot 10^{7}}$	$> 10^{10}$	$\mathbf{96 \cdot 10^{7}}$	–	–
revx	$> 10^{10}$	$\mathbf{36 \cdot 10^{7}}$	$> 10^{10}$	$\mathbf{94 \cdot 10^{7}}$	$> 10^{10}$	$\mathbf{5 \cdot 10^{9}}$
mult64	$> 10^{10}$	$\mathbf{39 \cdot 10^{7}}$	$> 10^{10}$	$\mathbf{73 \cdot 10^{8}}$	–	–
max	$> 10^{10}$	$\mathbf{91 \cdot 10^{6}}$	$> 10^{10}$	$\mathbf{96 \cdot 10^{7}}$	–	–
average	$7.7 \cdot 10^{9}$	$2 \cdot 10^{8}$	$7.9 \cdot 10^{9}$	$1.5 \cdot 10^{9}$	$8.3 \cdot 10^{9}$	$2.8 \cdot 10^{9}$

the benchmark circuit 'hamming'. The proposed method converges in some cases slowly but it provides better results when we enable to run it longer. See for example benchmarks 'des_area', 'simple_spi', 'ss_pcm', 'usb_funct', or 'hamming'. In these cases the proposed method requires more generations to reduce the circuits by 1%, but substantially less generations are needed on average to achieve 5% reduction. The effect of slow convergence is especially noticeable on 'hamming' circuit, where approximately 250 times more generations were needed to reduce the original circuit by 5% and 10% percent. Despite of that, the proposed method was able to reach 30.1% reduction while the global method got stuck at 14.6%. The typical convergence curves for four benchmark circuits are shown in Fig. 3. The first three plots show how the global methods usually got stuck at local optima. The last plot depicts the situation where the global method performs better compared to the proposed one.



(a) sasc (b) hamming

(c) sqrt32 (d) i2c

Fig. 3: Typical convergence curves for four chosen benchmark circuits. The lower value (number of gates) the better result.

We assume that the slow convergence is caused by the fact that each sub-circuit produced by the proposed windowing algorithm is optimized for a fixed number of generations independently on its parameters (the number of gates, the number of PIs or POs, and so on). This simplifies the problem but leads to a potential inefficiency. Many generations can be wasted to optimize small circuits. In order to investigate this fact, we analyzed what is the average volume of the sub-circuit. The results are summarized in Tab. 4. The table contains the average number of leaves, roots and volume of the windows produced by the proposed windowing algorithm. Despite using a simple strategy for selecting a root node, the window parameters are relatively good and sub-circuits of a reasonable volume are produced. The number of leaves $|L|$ determining the number of primary inputs of the sub-circuit optimized by evolution is substantially higher compared to the numbers used in rewriting. Compared to the rewriting, a relatively complex portions of the original circuits are chosen for subsequent optimization. This could explain the reason, why the proposed EA-based method is able to achieve such reduction compared to the conventional state-of-the-art synthesis.

Table 4: Average number of leaves, roots and volume of the windows produced by the proposed windowing algorithm. The averages are reported for all windows (first three columns) and those leading to a reduction (last three columns).

	windows					
	all created			causing reduction		
Benchmark	$\lvert L\rvert$	$\lvert R\rvert$	volume	$\lvert L\rvert$	$\lvert R\rvert$	volume
DSP	32	26	53	46	38	86
mem_ctrl	27	25	38	28	26	44
pci_spoci_ctrl	14	13	21	18	19	32
systemcaes	22	15	35	14	13	26
systemcdes	27	26	51	38	39	78
mult32	20	16	34	26	21	52
sqrt32	33	29	62	20	17	37
diffeq1	30	27	53	28	26	55
div16	32	28	50	25	24	44
hamming	30	26	44	26	24	45

We analyzed all the evolutionary runs across all benchmarks circuits and determined the maximum number of generations that caused a reduction of a sub-circuit. For each run of CGP we recorded the last generation that caused a change in the number of gates together with the volume of the optimized sub-circuit. The obtained numbers are plotted as a function of sub-circuit volume using a boxplot in Fig. 4. As expected, the more nodes are there in the sub-circuit the more CGP generations are typically used to optimize it. We can also see that the dependence between these two values is exponential – this is illustrated also by the blue curve representing polynomial interpolation of the median value. As the volume of the window increases, the number of occurrences of such cases decreases (see the numbers above each boxplot showing how many times we seen a window having volume between X and X+10). Usually, small windows are produced. Windows up to 45 nodes were produced in more than 77% cases. Due to this fact, the interpolation is limited to 150 nodes because there is insufficient number of results for the bigger windows.

5 Conclusion

Compared to the conventional logic synthesis, state-of-the-art EA-based optimization is able to produce substantially better results but at the cost of a higher run time. Unfortunately, the run time increases with the increasing complexity

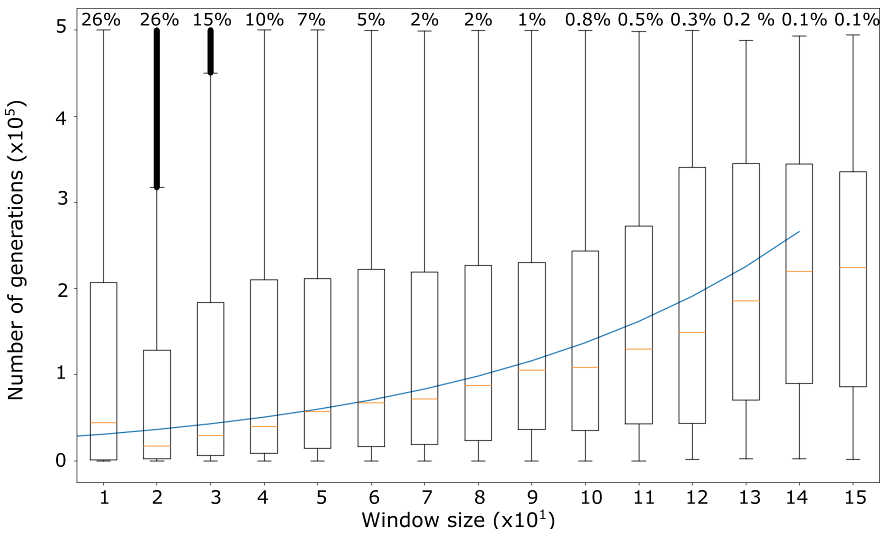


Fig. 4: Boxplots showing the number of generations that caused removal of a gate. The numbers above each boxplot show the number of occurrences of the window of a certain volume.

of the Boolean networks. This paper addresses this problem by combining the EA-based optimization with windowing that allows to work on a smaller portions of the original Boolean network. Even though we used a very simple strategy of root node selection which may degrade the capabilities of the resynthesis, the proposed method is able to outperform the original EA-based optimization applied to the whole Boolean networks. The number of nodes w.r.t the original method was improved by 9.2% on average. Even though only area was analyzed in this study, the depth of the optimized circuits is comparable with the original circuits.

In our future work, we would like to implement an adaptive strategy that modifies the maximum number of generations according to the size of the optimized logic circuit. In addition to that, we would like to focus on improvement of root node selection strategy. The question here is whether the result would be better if the cut is built from a node near to the previously chosen one.

6 Acknowledgments

This work was supported by The Ministry of Education, Youth and Sports of the Czech Republic – INTER-COST project LTC18053 and by the Brno University of Technology project FIT-S-17-3994.

References

1. Mishchenko, A., Chatterjee, S., Brayton, R.: DAG-aware AIG rewriting: a fresh look at combinational logic synthesis. In: 2006 43rd ACM/IEEE Design Automation Conference. pp. 532–535 (July 2006)
2. Mishchenko, A., Brayton, R.: Scalable logic synthesis using a simple circuit structure. In: Int. Workshop on Logic and Synthesis. pp. 15–22 (2006)
3. Sekanina, L., Ptak, O., Vasicek, Z.: Cartesian genetic programming as local optimizer of logic networks. In: 2014 IEEE Congress on Evolutionary Computation. pp. 2901–2908. IEEE CIS (2014)
4. Fiser, P., Schmidt, J., Vasicek, Z., Sekanina, L.: On logic synthesis of conventionally hard to synthesize circuits using genetic programming. In: 13th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems. pp. 346–351 (2010)
5. Fiser, P., Schmidt, J.: Small but nasty logic synthesis examples. In: Proc. 8th Int. Workshop on Boolean Problems. pp. 183–190 (2008)
6. Vasicek, Z.: Cartesian GP in optimization of combinational circuits with hundreds of inputs and thousands of gates. In: Proceedings of the 18th European Conference on Genetic Programming – EuroGP. pp. 139–150. LCNS 9025, Springer International Publishing (2015)
7. Li, N., Dubrova, E.: AIG rewriting using 5-input cuts. In: Proc. of the 29th Int. Conf. on Computer Design. pp. 429–430. IEEE CS (2011)
8. Lohn, J.D., Hornby, G.S.: Evolvable hardware: Using evolutionary computation to design and optimize hardware systems. IEEE Computational Intelligence Magazine 1(1), 19–27 (2006)
9. Miller, J., Thomson, P.: Cartesian Genetic Programming. In: Proc. of the 3rd European Conference on Genetic Programming EuroGP2000. LNCS, vol. 1802, pp. 121–132. Springer (2000)
10. Miller, J.F.: Cartesian Genetic Programming. Springer-Verlag (2011)
11. Vassilev, V., Job, D., Miller, J.F.: Towards the Automatic Design of More Efficient Digital Circuits. In: Lohn, J., Stoica, A., Keymeulen, D., Colombano, S. (eds.) Proc. of the 2nd NASA/DoD Workshop on Evolvable Hardware. pp. 151–160. IEEE Computer Society, Los Alamitos, CA, USA (2000)
12. Tao, Y., Zhang, L., Zhang, Y.: A projection-based decomposition for the scalability of evolvable hardware. Soft Computing 20(6), 2205–2218 (Jun 2016)
13. Stomeo, E., Kalganova, T., Lambert, C.: Generalized disjunction decomposition for the evolution of programmable logic array structures. In: First NASA/ESA Conference on Adaptive Hardware and Systems (AHS'06). pp. 179–185 (2006)
14. Vasicek, Z., Sekanina, L.: Formal verification of candidate solutions for post-synthesis evolutionary optimization in evolvable hardware. Genetic Programming and Evolvable Machines 12(3), 305–327 (2011)
15. Fiser, P., Halecek, I., Schmidt, J.: Are xors in logic synthesis really necessary? In: IEEE 20th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS). pp. 138–143 (2017)
16. Fiser, P., Schmidt, J.: It is better to run iterative resynthesis on parts of the circuit. In: Proc. of the 19th Int. Workshop on Logic and Synthesis. pp. 17–24. Univ. of California Irvine (2010)
17. Wolf, C., Glaser, J., Kepler, J.: Yosys-a free verilog synthesis suite. In: Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip) (2013)
18. Brayton, R., Mishchenko, A.: ABC: An academic industrial-strength verification tool. In: Computer Aided Verification. pp. 24–40. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)

Appendix B

EA-based refactoring of mapped logic circuits

KOCNOVA Jitka and VASICEK Zdenek

In: *2019 IEEE International Symposium on Circuits and Systems (ISCAS).* Sapporo, Japan, 2019, pp. 1-5, doi: 10.1109/ISCAS.2019.8702084.

EA-based refactoring of mapped logic circuits

Jitka Kocnova and Zdenek Vasicek
Brno, Czech Republic
Email: ikocnova, vasicek@fit.vutbr.cz

***Abstract*—The increasing complexity of the designs and problematic scalability of original representations led to a shift in internal representations used in logic synthesis and optimization. Heterogeneous representations were replaced with homogeneous intermediate representations. And-inverter graph (AIG) has been identified as the most promising structure for scalable logic optimization and many efficient algorithms were implemented on top of it. However, the inability of AIG to efficiently represent XOR gates together with heuristic nature of logic optimization algorithms leads to some inefficiency causing that the logic can be further minimized even after it has been mapped. This paper presents an optimization technique based on refactoring targeting mapped combinational circuits. It iteratively selects large cones of logic, optimizes them and returns them back to the original structure provided that there is an improvement in some metric. Performance of the method is evaluated on a set of complex academic and industrial benchmarks. We show that a 9.2% reduction in area can be achieved in average compared to the highly optimized results obtained using the academic state-of-the-art synthesis tool. In average, more than 14% reduction was observed for arithmetic circuits.**

I. Introduction

The goal of the logic optimization is to transform a suboptimal solution into an optimal gate-level implementation w.r.t. given synthesis goals, while technology mapping transposes it onto its best standard cell implementation. The circuit is typically represented by a suitable internal representation during the logic optimization. Current state-of-the-art logic synthesis tools, such as ABC, represent circuits using a directed acyclic graph composed of two-input AND nodes denoted as and-inverter graph (AIG). This representation is simple and scalable, and leads to simple algorithms. The optimization of AIGs is based on *rewriting* algorithm which minimizes size of AIG by iteratively selecting subgraphs rooted at a node and replacing them with smaller precomputed subgraphs [1].

Unfortunately, the AIGs suffer from an inherent bias in representation. While eight of ten possible two-input logic gates may be represented by means of a single AIG node, XOR/XNOR gates require three AIG nodes each. The efficiency of synthesis is then limited as it mostly fully relies on transformations disallowing to increase the number of AIG nodes. Also, the synthesis algorithms typically do not treat XORs explicitly – they rely on identification of XORs during the technology mapping phase which works independently on the logic optimization phase. The ability to capture XOR gates is, however, essential for efficient representation of arithmetic and XOR-intensive circuits [4].

To address this problem, e.g. binary decision diagrams (BDDs) can be employed [7], [8]. Due to their limited scalability, Amaru et al. employed a two step synthesis process based on a selective and distinct manipulation of AND/OR and XOR-intensive portions of the logic circuit [9]. Fiser et al. introduced XOR-AIGs to explicitly support XOR gates [10]. Haaswijk et al. employed XOR majority graphs (XMGs) to extend the capabilites of exact synthesis oriented on area optimization.

Other authors tried to avoid intermediate representation. Optimization based on a variant of Genetic Programming (GP) conducted directly at the level of common gates is able to provide significantly better results compared to the state-of-the-art synthesis operating on AIGs [11]. Optimization is done implicitly without any structural biases. In average, the method enabled a 34% reduction in gate count on an extensive set of IWLS benchmark circuits when executed for 15 minutes. A similar approach was successfully applied even to synthesis of conventionally hard to synthesize circuits [5]. The proposed method is able to optimize the circuits for which conventional synthesis completely fails. However, the efficiency of the evolutionary approach deteriorates with the increasing number of gates because of various scalability issues inevitably connected with the usage of GP.

The recent methods need to perform a preprocessing or circuit decomposition [9] or precomputation of ideal solutions[10]; other methods rely on XOR-avare transformations or presence technology cells (eg. XMGs). In order to eliminate the need for circuit preprocessing, we propose a novel logic synthesis methodology that implicitly targets XOR-intensive logic circuits.

II. Background

All circuits can be represented by a *Boolean network* – a directed acyclic graph (DAG) with nodes represented by Boolean functions [3]. The sources are the primary inputs (PIs) of the network and the sinks are the primary outputs (POs). The output of a node may be an input to other nodes called *fanouts*. The inputs of a node are called *fanins*. An edge connects two nodes in fanin/fanout relationship.

A. Limiting the scope of Boolean networks

Network scoping is a key operation to ensure a good scalability of synthesis tools when working with large Boolean networks. *Windowing* and *cut computation* have been proposed to limit the scope of logic synthesis to work only on a small portion of a Boolean network [3].

The windowing algorithm takes a node and two integers: the number of logic levels on the fanin/fanout sides of the node to be included in the window. Leaf set and root set are produced. The window is the subset of nodes of the network containing nodes from root set together with all nodes on paths between the leaf set and the root set. The nodes in the leaf set are not included in the window [3]. It is hard to predict how many logic levels have to be traversed to get a window of the desired size and required number of leaves. Hence, an approach based on k-feasible cuts is preferred. A cut of a node (root node) is a set of nodes of the network (leaves), such that each path from PI to the root node passes through at least one leaf. A cut is k-feasible if the number of nodes (i.e. cut size) in the cut does not exceed k. The volume of a cut is the total number of nodes encountered on all paths between the root node and the cut leaves. To maximize the cut volume, a reconvergence-driven heuristic is applied. The problem is that the cut computed using a naive bread-first-search algorithm may include only few nodes and leads to tree-like logic structures that do not lead to any don't cares in the local scope of the node and attempting optimization using such a cut would be wasted time [3].

A simple and efficient cut computation algorithm producing a cut close to a given size while heuristically maximizing the cut volume and the number of reconvergent paths subsumed in the cut has been introduced in [3]. Our work is based on the reconvergence-driven cuts and we discuss this algorithm more in the chapter III.

B. Synthesis of Boolean networks using EAs

Evolutionary algorithms (EAs) have been used to synthesize logic circuits since late nineties [12], [13]. Miller et al., the author of Cartesian Genetic Programming (CGP) [13], is considered as a pioneer in the field of logic synthesis of gate-level circuits. Despite of many advantages of this technique, only small problem instances were typically addressed. The scalability of CGP has been significantly improved by a SAT-based CGP simulator driven by counterexamples produced by the SAT solver [15] [11]. In this area, a linear form of CGP is preferred today. CGP models a candidate circuit having n_i PIs and n_o POs as a linear 1D array of n_n configurable nodes. Each node has n_a inputs and corresponds with a single gate with up to n_a inputs. To avoid a feedback, the inputs can be connected either to the output of a node placed in the previous L columns or directly to PIs. The function of a node can be chosen from a set of n_f functions. Depending on the function of a node, some of its inputs may become redundant. Moreover, the fixed number of nodes n_n does not mean that all the nodes contribute to the POs. These key features allow redundancy and flexibility of CGP. For details of candidate circuits encoding, please see [13].

CGP is a population oriented approach operating with $1+\lambda$ candidate solutions. The initial population is seeded by the original circuit. Every new population contains the best circuit from the previous population, that has not served as a parent yet and its λ offsprings created using a mutation operator that randomly modifies up to h integers. Selection of the individuals is typically based on a cost function (e.g. number of active nodes). Considering the CGP encoding, a single mutation causes either reconnection of a gate, reconnection of primary outputs or change in function of a gate. This procedure is typically repeated for a predefined number of iterations.

III. The proposed method

Let $\mathcal{C}$ be a combinational circuit described at the level of common gates represented by a Boolean network N consisting of $|N|$ nodes. Each node corresponds with a single gate in $\mathcal{C}$. The pseudo-code of the proposed optimization procedure based on evolutionary resynthesis is shown in Algorithm 1.

Algorithm 1: EA-BASED REFACTORING

```
   Input: A Boolean network $N$, maximum cut size $cutsize$
   Output: Optimized network $N'$, $cost(N') \leq cost(N)$
1  $N' \leftarrow N$
2  while terminated condition not satisfied do
3      $m \leftarrow$ identify the best candidate root node $m \in N'$
4      $C \leftarrow$ ReconvergenceDrivenCut($m$, $cutsize$)
5      $W \leftarrow$ ExpandCutToWindow($m$, $C$)
6      if $W$ is not a suitable candidate then
7          continue
8      $W' \leftarrow$ OptimizeNetworkUsingEA($W$)
9      if cost($(N' \setminus W) \cup W'$) < cost($N'$) then
10         $N' \leftarrow (N' \setminus W) \cup W'$
11 return $N'$
```

Firstly a node which may lead to the best improvement of N is determined. Identification of this node is a nontrivial problem, so some heuristic needs to be implemented – the size of transitive fan-in cone, level of the node or a more complex information can be used. A window is then extracted from the Boolean network. This procedure starts with computation of the reconvergence-driven cut (see Section II-A) and is followed by expansion of the cut C into a window W. In addition to the nodes inside the cut, we consider also all nodes that are not contained in the cut but have fanins inside the cut. Our expansion is similar to that employed in the resubstitution [3] where transitive fanout of C is considered, but we do limit the number of included nodes or their maximum level.

Resynthesis is then applied to the window. Each window potencially leading to no improvement is skipped in order to eliminate execution of a relatively time-consuming resynthesis. Identification of suitable windows can be based on the size of W or a combination of size of C and W (small and thin windows are skipped). We can also use the information about the difference among level of the root node and leaves of C.

The expansion leads to the set of internal nodes I, the set of leaves L and the set of root nodes R. L contains nodes serving as PIs of the temporary network used in the subsequent optimization. R contains nodes whose outputs have to be connected to POs. R contains the root node m and also other nodes with fanouts outside of the window. It holds that $C \subseteq L$ since the expansion may cause that some leaves of C become a fanout of a node inside the window. Two situations can happen for a leaf node. If all fanins are inside the window, the leaf can be simply removed from L. Otherwise, all fanins of the original leaf node need to be added to L. This procedure is repeated iteratively to ensure that there are no leaves having a fanin already included the window.

The resynthesis is performed by means of the CGP. The evolutionary optimization is executed for a limited number of iterations.The more iterations are allowed, the higher improvement can be achieved. However, many iterations on a small window mean a waste of time. Finally, the optimized logic network W' is evaluated w.r.t. N' and if it performs better, it replaces all non-leaf nodes included in W. The whole optimization algorithm is terminated when a predefined number of iterations or a given runtime is exhausted.

IV. Experimental evaluation

A. Experimental setup

Our goal is to evaluate performance of the proposed method w.r.t. the state-of-the-art EA-based method (denoted as global) applied to the whole Boolean network and to compare both methods to the best result produced by the ABC. Both methods operate at the level of optimized and mapped Boolean networks to avoid the bias of AIG representation. The procedure OptimizeNetworkUsingEA is based on the CGP implemented as described in Section II-B with following parameters: $n_a = 2$, $\lambda = 1$, $h = 2$, $n_n = |W|$. A single call of this procedure is executed for the global method. On contrary, several calls of this procedure are executed in the proposed method. The global method terminates after n_{iters} iterations. The proposed method uses a simple divide-and-conquer strategy. The proposed method is allowed to create n_{cuts} cuts. For each cut, the OptimizeNetworkUsingEA is allowed to perform n_{iters}/n_{cuts} iterations. In total, n_{iters} evolutionary iterations are evaluated in both cases. This naive strategy supposes that the computation effort does not depend on the window size but it helps to fairly evaluate the impact of the proposed method. In this paper, we use $n_{iters} = 10^9$ iterations. The $cutsize$ limit is set to 10^4. Only windows with more than 10 nodes are accepted. The root node m is chosen randomly.

This setup was considered the best amongst other setup combinations of the $cutsize$ (5, 10, 20, 35, 50, 75, 100, 150, 200, 250, 10^3, 10^4) and n_{iters} ($10^3, 10^5, 10^7, 10^9$). Experimental results showed a convergence of the number of removed cells after approximately $n_{iters} = 10^7$. $cutsize = 10^4$ cells ensures the possibility of the biggest possible cut creation w.r.t the root cell placement.

This strategy simplifies the problem but may lead to degradation of the performance if many unacceptable windows are produced. The only criterion we consider is the area on a chip expressed as the number of gates. For each method and each benchmark, five independent runs were executed to obtain statistically significant results.

B. Results

The proposed method was implemented in C++ and integrated in Yosys open synthesis suite. Tab. I summarizes the experimental results. The goal was to improve the size of mapped benchmark circuits optimized at the level of AIG by ABC. In particular, we took 19 highly optimized circuits from IWLS'05 Open Cores benchmarks and 9 highly optimized large arithmetic circuits[1].

The circuits were mapped to gates using a library of common 2-input gates including XORs/XNORs (ABC: 'map'). After mapping, optimization by the proposed and global method was executed and final number of mapped gates in circuits was examined. The circuits were then transformed to AIG representation (ABC: 'strash') and compared to the results from ABC-only optimization. All of the optimized circuits were formally verified w.r.t their original form (ABC: 'cec').

Many iterations of resyn script were applied in ABC on the original verilog benchmarks as described in [16] in order to obtain the best results for AIG optimization.

The first three columns of Tab. I contain information related to the benchmarks (name, number of PIs and POs). The next two columns contain number of nodes and depth of circuits in the AIG form after application of ABC resyn script. The other two columns show parameters of the mapped circuits and those numbers serve as a baseline for our comparison – the number of gates and logic depth is provided. Then, the achieved results expressed as the relative reduction w.r.t. the baseline are reported for the proposed and global method. For each method, we report the average and the best obtained improvement. These numbers are calculated from five independent runs.

The best results are very close to the average ones which suggests that the both EA-based methods are stable although they are in principle non-deterministic. According to the number of highlighted cases showing the better results, the proposed method performs substantially better considering the average as well as the best results. It wins in 22 out of 28 cases. The average reduction on the IWLS'05 benchmarks is slightly better in favor of the global method, but it is affected mostly by five cases where the global method provides substantially better results. Looking at the arithmetic circuits, the global method is able to slightly improve only two circuits. In other cases, the reduction is negligible. We analyzed the five cases where the global method outperformed the proposed one and concluded that the global method works well especially for small instances (less than 10^4 gates) that have a reasonable depth (10 to 25 levels). The global optimization of circuits with large depth performs unsatisfactory. Compared to ABC, a substantial improvement is achieved on the arithmetic circuits. The number of gates is reduced by nearly 15% in average. The highest reduction, 30.1%, is recorded for hamming benchmark. The detailed analysis revealed that this was possible due to better handling of XORs/XNORs and also by a relatively huge redundancy of the original circuit optimized by ABC. The relative number of AND/OR/NAND/NOR gates remained nearly the same (around 74%). The number of XORs/XNORs increased from 10% to 15%.

Our second experiment evaluates efficiency of the AIG representation. The last two columns of Tab. I show what

[1]All the benchmarks are taken from https://lsi.epfl.ch/MIG

TABLE I: Comparison of the proposed and global method (sec. Impr. proposed, Impr. global) w.r.t. the initial number of mapped gates (sec. ABC(mapped)) and the best resulf of ABC (sec. ABC(AIG)). Section ABC(AIG) / ABC(mapped) contains parameters of the optimized circuits before and after mapping (D is logic depth, G is the number of gates). Last section shows the size of AIG(relative to ABC) when the gate-level circuit is mapped back to AIG.

			ABC(AIG)		ABC(Mapped)		Impr. proposed		Impr. global [11]		Impr. at AIG level	
Benchmark	PIs	POs	AIG	D	gates	D	gates avg	gates best	gates avg	gates best	proposed	global [11]
DSP	4223	3792	39958	41	43491	45	**3.6%**	**3.6%**	0.0%	0.0%	**0.1%**	0.0%
ac97_ctrl	2255	2136	10497	9	11433	10	**2.9%**	**2.9%**	1.4%	1.4%	0.7%	**1.3%**
aes_core	789	532	20632	19	21128	20	**2.9%**	**2.9%**	0.6%	1.7%	-0.8%	-0.3%
des_area	368	70	5043	24	5199	25	**6.0%**	**6.1%**	2.1%	2.3%	**3.6%**	1.0%
des_perf	9042	1654	75561	15	78972	16	**1.8%**	**1.8%**	0.0%	0.1%	-2.7%	-6.8%
ethernet	10672	10452	56882	22	60413	23	**0.5%**	**0.5%**	0.0%	0.0%	**0.1%**	-0.1%
i2c	147	127	1009	10	1161	12	9.2%	9.2%	**10.0%**	**10.7%**	4.8%	**8.1%**
mem_ctrl	1198	959	9351	22	10459	24	7.0%	7.0%	**24.8%**	**25.4%**	2.4%	**26.0%**
pci_bridge32	3519	3136	16812	18	19020	21	**3.5%**	**3.5%**	0.5%	0.6%	0.4%	**0.5%**
pci_spoci_ctrl	85	60	994	13	1136	15	18.3%	18.5%	**34.8%**	**35.7%**	13.4%	**33.0%**
sasc	133	123	657	7	746	8	**6.2%**	**6.2%**	2.4%	2.8%	0.0%	-0.2%
simple_spi	148	132	770	10	822	11	**5.5%**	**5.7%**	4.4%	4.6%	**1.1%**	0.8%
spi	274	237	3430	24	3825	26	5.6%	5.6%	**13.5%**	**20.2%**	1.7%	**16.0%**
ss_pcm	106	90	381	6	437	7	**5.7%**	**6.7%**	2.3%	2.3%	-0.3%	**0.3%**
systemcaes	930	671	11014	31	11352	27	**11.9%**	**12.3%**	0.0%	0.0%	**3.3%**	-0.4%
systemcdes	314	126	2495	21	2601	25	4.8%	5.0%	**9.1%**	**9.9%**	2.2%	**5.2%**
tv80	373	360	7838	35	8738	39	6.6%	6.9%	**11.3%**	**11.1%**	2.9%	**12.4%**
usb_funct	1860	1692	13914	20	15405	23	**5.8%**	**5.9%**	2.6%	2.6%	1.4%	**2.8%**
usb_phy	113	73	380	7	452	9	**13.9%**	**14.0%**	12.2%	12.2%	3.9%	**5.8%**
average (IWLS'05 benchmarks)			14611	18	15620	20	6.4%	6.5%	**7.0%**	**7.6%**	2.0%	**5.5%**
mult32	64	64	8903	40	8225	42	**16.5%**	**16.6%**	0.0%	0.0%	-1.5%	0.0%
sqrt32	32	16	1353	292	1462	307	**22.3%**	**24.3%**	3.0%	3.0%	**4.2%**	-4.2%
diffeq1	354	193	21980	235	20719	218	**11.5%**	**11.5%**	0.0%	0.0%	**0.7%**	-7.3%
div16	32	32	5111	132	5847	152	**15.7%**	**15.8%**	0.0%	0.0%	**2.1%**	-12.0%
hamming	200	7	2607	73	2724	80	**28.6%**	**30.1%**	14.6%	14.6%	**11.0%**	-0.6%
MAC32	96	65	9099	54	7793	55	**7.7%**	**7.8%**	0.0%	0.0%	-9.7%	-13.0%
revx	20	25	7516	162	8131	171	**14.5%**	**14.5%**	0.0%	0.1%	**1.2%**	-13.0%
mult64	128	128	26024	186	21992	190	**7.4%**	**7.4%**	0.3%	0.5%	-5.4%	-1.0%
max	512	130	2964	113	3719	117	**5.3%**	**5.3%**	0.7%	0.8%	**0.8%**	-0.4%
average (arithmetic benchmarks)			9506	143	8956	148	**14.4%**	**14.8%**	2.1%	2.1%	**0.4%**	-5.7%

happens when we convert the optimized gate-level netlists to AIGs. This section contains the relative size improvement for the best results produced by the proposed and global method w.r.t. size of the AIGs produced by ABC. We can see that the average reduction is substantially lower compared to the reduction achieved on the gate level representation. In many cases, the AIG of the optimized circuit is even larger than the original one. However, such a behavior is expectable because this happens if the number of XORs increases but the overall number of removed gates is relative small. On the other hand, when the reduction at the level of gates exceeds a certain level the reduction is visible also on AIGs. This is evident especially on the IWLS benchmarks where the global method produces solutions that clearly dominate. From the perspective of AIGs, the global method completely failed on arithmetic benchmarks. The number of AIG nodes substantially increased in almost all cases. As discussed in the introduction, this simple comparison demonstrates the limited capabilities of otherwise efficient AIG representation.

Tab. II shows the average number of leaves, roots and volume of windows produced by the windowing algorithm on some benchmarks. Despite using a simple selection strategy, the parameters are relatively good. The number of leaves $|L|$ determining the number of primary inputs of the refactored subcircuit is substantially higher compared to the sizes of cuts used during rewriting. Number of cut nodes is also satisfactory. Compared to rewriting, a relatively complex portions of the original circuits are chosen for subsequent optimization. This could explain the reason, why the proposed method is able to achieve such reduction. Detailed analysis revealed that the size of the windows is typically higher for the arithmetic circuits.

TABLE II: Average parameters of all windows and windows that led to a reduction (col. successful windows) generated during the refactoring.

	all windows			successful windows		
Benchmark	$\|L\|$	$\|R\|$	size	$\|L\|$	$\|R\|$	size
mem_ctrl	27	25	38	28	26	44
pci_spoci_ctrl	14	13	21	18	19	32
systemcaes	22	15	35	14	13	26
mult32	20	16	34	26	21	52
sqrt32	33	29	62	20	17	37
diffeq1	30	27	53	28	26	55
div16	32	28	50	25	24	44
hamming	30	26	44	26	24	45

V. Conclusion

Compared to the conventional logic synthesis, state-of-the-art EA-based optimization is able to produce substantially better results at the cost of a higher run time that grows with the increasing complexity of the Boolean networks. This paper addresses this problem by combining the EA-based optimization with refactoring that allows to work on a smaller portions of the original Boolean network. Despite using a very simple strategy of root node selection which may degrade the capabilities of the refactoring, the proposed method is able to outperform the AIG-based as well as the original EA-based optimization applied to the whole Boolean networks.

References

[1] A. Mishchenko, S. Chatterjee, and R. Brayton, “Dag-aware aig rewriting: a fresh look at combinational logic synthesis,” in *2006 43rd ACM/IEEE Design Automation Conference*, July 2006, pp. 532–535.

[2] R. Brayton and A. Mishchenko, “ABC: An academic industrial-strength verification tool,” in *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 24–40.

[3] A. Mishchenko and R. Brayton, “Scalable logic synthesis using a simple circuit structure,” in *Int. Workshop on Logic and Synthesis*, 2006, pp. 15–22.

[4] P. Fiser and J. Schmidt, “The observed role of structure in logic synthesis examples,” in *18th Int. Workshop on Logic and Synthesis*, 2009, pp. 210–213.

[5] P. Fiser, J. Schmidt, Z. Vasicek, and L. Sekanina, “On logic synthesis of conventionally hard to synthesize circuits using genetic programming,” in *13th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems*, 2010, pp. 346–351.

[6] P. Fiser and J. Schmidt, “Small but nasty logic synthesis examples,” in *Proc. 8th Int. Workshop on Boolean Problems*, 2008, pp. 183–190.

[7] C. Yang and M. Ciesielski, “BDS: a BDD-based logic optimization system,” *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 21, no. 7, pp. 866–876, Jul 2002.

[8] N. Vemuri, P. Kalla, and R. Tessier, “Bdd-based logic synthesis for lut-based fpgas,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 7, no. 4, pp. 501–525, Oct. 2002.

[9] L. Amaru, P. E. Gaillardon, and G. D. Micheli, “Mixsyn: An efficient logic synthesis methodology for mixed xor-and/or dominated circuits,” in *2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2013, pp. 133–138.

[10] P. Fiser, I. Halecek, and J. Schmidt, “Sat-based generation of optimum function implementations with xor gates,” in *2017 Euromicro Conference on Digital System Design (DSD)*, 2017, pp. 163–170.

[11] Z. Vasicek, “Cartesian GP in optimization of combinational circuits with hundreds of inputs and thousands of gates,” in *Proceedings of the 18th European Conference on Genetic Programming – EuroGP*, ser. LCNS 9025. Springer International Publishing, 2015, pp. 139–150.

[12] J. D. Lohn and G. S. Hornby, “Evolvable hardware: Using evolutionary computation to design and optimize hardware systems,” *IEEE Computational Intelligence Magazine*, vol. 1, no. 1, pp. 19–27, 2006.

[13] J. Miller and P. Thomson, “Cartesian Genetic Programming,” in *Proc. of the 3rd European Conference on Genetic Programming EuroGP2000*, ser. LNCS, vol. 1802. Springer, 2000, pp. 121–132.

[14] V. Vassilev, D. Job, and J. F. Miller, “Towards the Automatic Design of More Efficient Digital Circuits,” in *Proc. of the 2nd NASA/DoD Workshop on Evolvable Hardware*, J. Lohn, A. Stoica, D. Keymeulen, and S. Colombano, Eds. Los Alamitos, CA, USA: IEEE Computer Society, 2000, pp. 151–160.

[15] Z. Vasicek and L. Sekanina, “Formal verification of candidate solutions for post-synthesis evolutionary optimization in evolvable hardware,” *Genetic Programming and Evolvable Machines*, vol. 12, no. 3, pp. 305–327, 2011.

[16] L. Amaru, P. E. Gaillardon, and G. D. Micheli, “Majority-inverter graph: A new paradigm for logic optimization,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 5, pp. 806–819, 2016.

Appendix C

EA-based Resynthesis: An Efficient Tool for Optimization of Digital Circuits

KOCNOVA Jitka and VASICEK Zdenek

In: *Genetic Programming and Evolvable Machines 21.* Pp 287–319, https://doi.org/10.1007/s10710-020-09376-3.

Genetic Programming and Evolvable Machines manuscript No.
(will be inserted by the editor)

EA-based Resynthesis: An Efficient Tool for Optimization of Digital Circuits

Jitka Kocnova · Zdenek Vasicek

Received: date / Accepted: date

Abstract Scalability of fitness evaluation was the main bottleneck preventing adopting the evolution in the task of logic circuits synthesis since early nineties. Recently, various formal approaches such as SAT and BDD solvers have been introduced to this field to overcome this issue. This made it possible to optimise complex circuits consisting of hundreds of inputs and thousands of gates. Unfortunately, we are facing another problem – scalability of representation. The efficiency of the evolutionary optimization applied at the global level deteriorates with the increasing complexity. To overcome this issue, we propose to apply the concept of local resynthesis in this work. Local resynthesis is an iterative process based on the extraction of smaller sub-circuits from a complex circuit that are optimized locally and implanted back to the original circuit. When applied appropriately, this approach can mitigate the problem of scalability of representation. Two complementary approaches to the extraction of the sub-circuits are presented and evaluated in this work. The evaluation is done on a set of highly optimized complex benchmark problems representing various real-world controllers, logic and arithmetic circuits. The experimental results show that the evolutionary resynthesis provides better results compared to globally operating evolutionary optimization. In more than 85% cases, substantially higher number of redundant gates was removed while keeping the computational effort at the same level. A huge improvement was achieved especially for the arithmetic circuits. On the average, the proposed method was able to remove 25,1% more gates.

Keywords Cartesian Genetic Programming · Evolutionary Resynthesis · Logic optimization

This work was supported by Czech Science Foundation project 19-10137S.

Zdenek Vasicek
Brno University of Technology, Faculty of Information Technology, IT4Innovations Centre of Excellence, Czech Republic
E-mail: vasicek@fit.vutbr.cz

Jitka Kocnova
Brno University of Technology, Faculty of Information Technology, IT4Innovations Centre of Excellence, Czech Republic
E-mail: ikocnova@fit.vutbr.cz

1 Introduction

Logic synthesis, as understood by the hardware community, is a process that transforms a high-level description into a gate-level or transistor-level implementation. Due to the complexity of the problem, the synthesis process is typically broken into a sequence of steps. Among others, logic optimization represents an important part of the whole process. The goal of the logic optimization is to transform a suboptimal solution into an optimal gate-level implementation with respect to given synthesis goals. Due to the scalability issues, the problem is typically represented using a suitable internal representation. Current state-of-the-art logic synthesis tools, such as ABC [1], represent circuits using a directed acyclic graph composed of two-input AND nodes connected by direct or negated edges denoted as and-inverter graph (AIG). The optimization of AIGs is based on *rewriting*, a greedy algorithm which minimizes the size of AIG by iteratively selecting subgraphs rooted at a node and replacing them with smaller precomputed subgraphs, while preserving the functionality of the root node [19]. AIG rewriting is local, however, the scope of changes becomes global by application of rewriting many times. In addition to that, *resubstitution* and *refactoring* can be employed. Resubstitution expresses the function of a node using other nodes present in the AIG [18]. Refactoring iteratively selects large cones of logic rooted at a node and tries to replace them with a more efficient implementation [19]. Refactoring can be seen as a variant of rewriting. The main difference is that rewriting selects subgraphs containing few leaves because the number of leaves determines the number of variables of a Boolean function whose optimal implementation is sought.

The AIG representation is simple and scalable, and leads to simple algorithms but it suffers from an inherent bias in representation. While eight of ten possible two-input logic gates may be represented by means of a single AIG node, XOR and XNOR gate require three AIG nodes each. The efficiency of synthesis is then limited as it mostly fully relies on transformations that disallow an increase of the number of AIG nodes. It has been shown that there exists a huge class of real-world circuits for which the synthesis fails and provides very poor results [21,4,3]. In some cases, the area of the synthesized circuits is of orders of magnitude higher than the known optimum. If a large design is broken down to multiple smaller circuits and such a failure occurs during resynthesis, we obtain an unacceptably large circuit.

Various evolutionary approaches working directly at the level of gates were successfully applied to address this problem [21,27]. Vasicek demonstrated that the evolutionary synthesis using Cartesian Genetic Programming (CGP) conducted directly at the level of common gates is able to provide significantly better results compared to the state-of-the-art synthesis operating on AIGs [27]. On the average, the method enabled a 34% reduction in gate count on an extensive set of benchmark circuits when executed for 15 minutes. It was observed, however, that the efficiency of the evolutionary approach deteriorates with an increasing number of gates. Substantially more generations were required to reduce circuits consisting of more than ten thousands gates. While [27] focuses strictly on the improvement of the scalability of the evaluation, Sekanina et al. employed a divide and conquer strategy to address the problem of scalability of representation [21]. The authors were able to obtain better results than other locally operating methods reported in the literature, however, the performance of this method was significantly worse than the evolutionary global optimization proposed in [27].

Motivated by the problem above, we propose to combine the evolutionary optimization with the principle of so called Boolean network scoping. Boolean network scoping represents a common approach incorporated in the conventional synthesis tools for maintaining the good scalability of the synthesis process. In particular, we propose to use an iterative procedure which extracts sub-circuits that are subsequently optimized by Cartesian Genetic Programming and implanted back into the original circuit provided that there is an improvement at the global level. This approach can be understood as the EA-based resynthesis. The size of the sub-circuits has impact not only on the scalability of the CGP but also on the efficiency of the whole optimization process. Small sub-circuits ensures a good scalability of the evolutionary optimization, but they lead to minor improvements at the global level because we obtained a method which operates mainly locally similarly to the conventional rewriting. Huge sub-circuits, on the other hand, increases possibilities for an improvement but the performance of the CGP deteriorates with increasing the size of the optimized circuit. In order to have a reasonable optimization method, it is necessary to find a good trade-off between the mentioned two extremes.

Several heuristics for Boolean network scoping on the level of AIGs have been proposed in the literature (see Section 2.2). These heuristics have typically been introduced in the context of some more complex algorithms and used as a part of their functionality. It means that they are tailored to the particular scenario and need to be modified to be used for our needs. The rewriting, for example, is designed to work with sub-circuits having at most five inputs and exactly one output. In our case, we do not need to introduce any hard limits on the number of inputs or outputs. Compared to the rewriting, the evolutionary resynthesis profits of a substantially higher number of gates (e.g. low hundreds of gates) that are more like to be further reduced.

1.1 Goals and Contributions

The work in this paper extends the preliminary results presented in [10] where we used a method of Boolean network scoping inspired by the conventional method based on computing so called k-feasible cuts. In this paper, we introduce an alternative method and evaluate its parameters compared to the cut-based method as well as conventional state-of-the-art synthesis. Our goal is to improve the efficiency of the evolutionary optimization and get rid of some parameters and limitations connected with the usage of the cut-based method. In addition to that, a more detailed description and experimental evaluation of both methods is presented.

1.2 Organization

The rest of this paper is organized as follow. Section 2 presents a background in Boolean networks and network scoping and the related work in the area of the evolutionary synthesis of digital circuits. Section 3 introduces the proposed approach to the evolutionary resynthesis of large combinational circuits. Section 4 describes the experimental setup and experiments with the parameter setting.

The obtained results are presented and discussed in Section 5. Finally, Section 6 provides the conclusions and some ideas for future work.

2 Background and Related Work

This section presents relevant background on conventional as well as EA-based optimization of logic circuits and introduces the notation used in the rest of the paper.

2.1 Boolean Networks

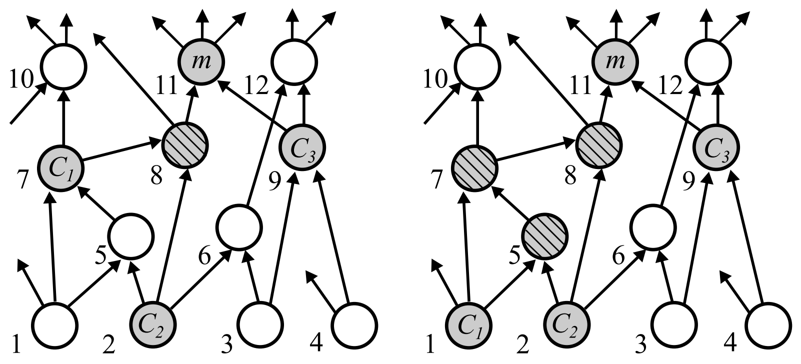
Every circuit can be represented using a Boolean network. A *Boolean network* is a directed acyclic graph (DAG) with nodes represented by Boolean functions [18]. The sources of the graph are the primary inputs (PIs) of the network and the sinks are the primary outputs (POs). The output of a node may be an input to other nodes called *fanouts*. The inputs of a node are called *fanins*. An edge connects two nodes that are in fanin/fanout relationship. Considering this notion, And-Inverter Graph is a Boolean network composed of two-input ANDs and inverters. The network primary inputs are signals that are driven by the environment, there is no node driving these signals in the network. Similarly, the primary outputs are signals that drive the environment and are needed by inner network nodes as well. The size of the network is the number of the nodes (primary inputs and outputs are not considered).

2.2 Limiting the Scope of Boolean Networks

Network scoping represents a key operation to ensure a good scalability of synthesis tools when working with large Boolean networks. In addition, it forms an integral part of rewriting as well as refactoring. Two approaches have been proposed to limit the scope of logic synthesis to work only on a small portion of a Boolean network – *windowing* and *cut computation* [18].

The windowing algorithm determining the window for a given node takes a node and two integers defining the number of logic levels on the fanin/fanout sides of the node to be included in the window. Two sets are produced as the result of windowing – leaf set and root set. The window of a Boolean network is the subset of nodes of the network containing nodes from root set together with all nodes on paths between the leaf set and the root set. The nodes in the leaf set are not included in the window. The main problem of this algorithm is that it is hard to predict how many logic levels have to be traversed to get a window of the desired size and required number of leaves.

A complementary approach based on computing so called k-feasible cuts is usually preferred to avoid determining the required number of logic levels. A cut of a node, called root node, is a set of nodes of the network, called leaves, such that each path from PI to the root node passes through at least one leaf. A cut is k-feasible if the number of nodes (i.e. cut size) in the cut does not exceed k. The volume of a cut is the total number of nodes encountered on all paths between the



(a) Cut $C^{I} = \{7, 2, 9\}$ (b) Cut $C^{II} = \{1, 2, 9\}$

Fig. 1 Example of two possible 3-feasible cuts for root node m and given Boolean network. The cut C^{II} is preferred as its volume is four (root node m and contained nodes 5, 7, and 9). There is only one contained node (node 8) in the case of C^{I}.

root node and the cut leaves. An example of two different 3-feasible cuts is shown in Figure 1. To maximize the cut volume, a reconvergence-driven heuristic is applied in practice. The problem is that the cut computed using a naive bread-first-search algorithm may include only few nodes and leads to tree-like logic structures (see Fig. 1a showing a cut determined by the naive approach and Fig. 1b showing the output of reconvergence-driven heuristic). Such a structure does not lead to any don't cares in the local scope of the node and attempting optimization using such a cut would be wasted time. A simple and efficient cut computation algorithm producing a cut close to a given size while heuristically maximizing the cut volume and the number of reconvergent paths subsumed in the cut has been introduced in [18]. As our work is based on the reconvergence-driven cuts, we briefly discuss this algorithm. The algorithm starts with a set of leaves consisting of a single root node. This set is incrementally expanded by adding one node in each step of a recursive procedure. If the set consists of only PIs, the procedure quits. Otherwise, a non-PI node that minimizes a cost function is chosen from the set of leaves. The chosen node is removed from the leaf set and all its fanins are included instead of it. This causes expansion of the cut. If the cut-size limit is exceeded, the procedure quits and returns the cut before expansion. The cost function returns the number of new nodes that should be added to the leaf set instead of the removed node.

The k-feasible cuts are important not only for the gate-level logic synthesis but also for FPGA-based synthesis as a k-feasible cut can be implemented as a k-input LUT. For resubstitution and FPGA-based mapping, so called maximum fanout free cone (a subnetwork where no node in the cone is connected to a node not in the cone) is requested. It means that the cut-based scoping must always produce a single-output sub-circuits. Otherwise it would be impossible to replace the whole sub-circuit by a precomputed optimal implementation / a single LUT. Typically, 4-feasible and 5-feasible cuts are used for rewriting-based logic synthesis [18,12]. Small k is used not only to make the cut enumeration possible but also to manage memory requirements to store the precomputed optimal implementations of all k-input Boolean functions. For FPGA-based mapping, 5-input and 6-input LUTs are used. Apart from the rewriting, the reconvergence-driven cuts have been applied to refactoring and resubstitution [18]. Typically, k is between 5 and 12 for refactoring depending on the computation effort allowed [18].

2.3 Evolutionary Synthesis of Logic Circuits

Advancements in technology developed in the early nineties enabled researchers to sucessfully apply techniques of evolutionary computation in various problem domains. In the middle nineties, Higuchi and Thompson, two of the most prominent pioneers, demonstrated that evolutionary algorithms are able to solve non-trivial hardware-related problems [9,26]. The achievements presented in the seminal paper of Higuchi et al. [9] motivated other scientists to intensively explore a new and promising research topic. As a consequence of that, new research direction referred to as *Evolvable hardware* has emerged [7] focusing on the use of evolutionary algorithms to create specialized electronics without manual engineering.

The gate-level evolution has been addressed only rarely before the year 2000. The first results in the area of digital circuit synthesis were reported by Koza in 1992, who investigated the evolutionary design of even-parity circuits in his extensive discussions of the standard genetic programming (GP) paradigm [11]. Later, Thompson used a form of direct encoding loosely based on the structure of an FPGA in his experiment with evolution of a square wave oscillator [26]. Genetic algorithm has been employed also by Coello who evolved various 2-bit adders and multipliers [2]. Finally, Miller et al. demonstrated that evolutionary design systems are not only able to rediscover standard designs as it has been shown in the past, but they can, in some cases, improve them [17,14]. The method of evolving digital circuits developed by Miller in 1997 [17] was subsequently revised and a new evolutionary algorithm known as Cartesian genetic programming (CGP) was introduced in 2000 [13]. CGP, which is a general form of genetic programming, was designed to address two issues related to the efficiency of common tree-based genetic programming. Firstly, as GP represents candidate solutions using trees, it does not naturally capture the structure of digital circuits that typically form a directed acyclic graph (DAG). Secondly, GP exhibits the so-called bloat effect enabling the programs to grow uncontrollably until they reach the GP's tree-depth maximum.

Miller is considered as a pioneer in the field of logic synthesis of gate-level circuits. He utilized CGP to demonstrate that evolutionary computing can improve results of conventional circuit synthesis and optimization algorithms. As a proof-of-concept, small arithmetic circuits were considered. A 4-bit multiplier was the most complex circuit evolved in this category [29]. For the next decade, however, the problems addressed by the EHW community remained nearly of the same complexity. The most complex combinational circuits that were directly evolved during the first two decades of EHW consisted of tens of gates and had around 20 inputs [23]. Many novel techniques including decomposition, development, modularization, new problem representations and function level evolution have been proposed [23,22,15,31,20]. The projection-based decomposition approaches such as [24] or [25] helped to increase the complexity of problem instances that can be solved by EAs. Despite of that, the gap between the complexity of problems addressed in industry and EHW continued to widen as the advancements in technology developed. Evolvable hardware found itself in a critical stage around the year 2010 and it was not clear whether there exists a path forward which would allow the field to progress [8]. The scalability problem has been identified as one of the most difficult problems the researchers are faced in this field and that should be, among others, addressed in the future.

In 2011, the scalability of CGP has been significantly improved by introducing a SAT-based CGP. The SAT-based CGP uses a modern SAT solver to avoid an expensive exhaustive circuit simulation commonly used to determine the Hamming distance between a candidate solution and specification [28]. It exploits the fact that the candidate solutions must be functionally equivalent with their parent in logic optimization in order to be further accepted. In addition to that, it exploits the knowledge of differences between parental and candidate circuits. The efficiency of SAT-based method was further improved by combining a SAT solver with an adaptive high-performance circuit simulator used to quickly identify the potential functional non-equivalence [27]. The most advanced SAT-based CGP employs a simulator that is driven by counterexamples produced by the SAT solver as suggested in [27]. Neither the original nor the latter approach rely on a decomposition. The gate-level circuits are optimized directly.

2.4 Cartesian Genetic Programming

Since its introduction, CGP remains the most powerful evolutionary technique in the domain of EA-based logic synthesis and optimization [13]. In this area, a linear form of CGP is preferred today. In this case, CGP models a candidate circuit having n_i PIs and n_o POs as a linear 1D array of n_n configurable nodes. Each node has n_a inputs and corresponds with a single gate with up to n_a inputs. Two-input and single-output nodes are typically used. The inputs can be connected either to the output of a node placed in the previous L columns or directly to PIs. This avoids a feedback. The function of a node can be chosen from a set Γ consisting of $|\Gamma| = n_f$ functions. Depending on the function of a node, some of its inputs may become redundant. In addition to that, some of the nodes may become redundant because they are not referenced by any node connected a PO. It means that the fixed number of nodes n_n does not mean that all the nodes are effective used. The redundant nodes and inputs lead to the presence of non-coding genes in the genotype. This feature makes the search effective [16].

The candidate circuits are encoded as follows. Each PI as well as each node has associated an unique index. Each node is encoded using $n_a + 1$ integers $(x_1, \cdots, x_{n_a}, f)$ where the first n_a integers denote the indices of its fanins and the last integer determines the function of that node. Every candidate circuit is encoded using $n_n(n_a + 1) + n_o$ integers where the last n_o integers specify the indices corresponding with each PO.

The most common search technique used in connection with the CGP is an Evolutionary strategy (ES) [13]. Typically $(1 + \lambda)$-ES is employed, where λ corresponds with the number of new candidate solutions generated from a single parental solution. In the circuit optimization, the initial population is seeded by the original circuit ought to be optimized. Every new population consists of the best circuit chosen from the previous population and its λ offspring created using a mutation operator. Either point or probabilistic mutation is used in the standard

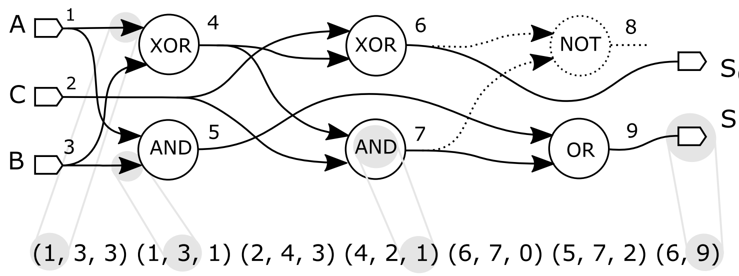


Fig. 2 Example of a CGP individual encoding a logic circuit (one-bit full adder) with $n_i = 3$ inputs and $n_o = 2$ outputs. The individual is encoded using an array of $n_n = 6$ two-input single-output nodes whose functions are chosen from a set of primitive functions $\Gamma = \{\mathrm{NOT}, \mathrm{AND}, \mathrm{OR}, \mathrm{XOR}\}$. Note that the nodes are arranged in a two-dimensional grid for improved readability. Redundant connections and nodes, i.e. those that do not contribute to the outputs, are highlighted using a dotted line.

CGP. Point mutation is typically preferred because it is easier to implement and more efficient than using a probabilistic mutation [16].

The point mutation randomly modifies up to h genes (integers) of a parent genotype to create an offspring. Considering the CGP encoding, a single mutated gene causes either reconnection of a node, reconnection of a primary output or change in function of a node. Due to the presence of redundant genes, the mutation may occur in the redundant part, which means that the mutated genotype has the same phenotype as its parent. Such a mutation is sometimes denoted as neutral since the fitness value remains unchanged. To avoid wasted fitness evaluations, several mutation strategies have been proposed [5,16]. Single Active Mutation strategy, for example, mutates the offspring until one active gene is changed. Another possibility is to detect the neutral mutations and skip the time-consuming fitness evaluation procedure. Considering the usage of CGP in the optimization of logic circuits, the latter approach has been typically used [28,27]. Crossover is not used in the standard CGP because it was found that crossover has little effect on the efficiency of CGP [16].

The main disadvantage of the CGP encoding in connection with the point mutation operator is the presence of a strong length and positional bias that results in large portions of the genotype that are always redundant and never used by any ancestor. To address this issue, several approaches have been proposed [16]. Goldman and Punch, for example, proposed to apply Reorder operation once each generation that shuffles the position of nodes in the parent [6]. Reorder does not semantically change the parent but it allows active nodes to be evenly distributed within the whole genotype. This approach eliminates the length as well as positional bias and improves the efficiency of the search.

The selection of the individuals is typically based on a cost function (e.g. the number of active nodes). In the case that there are more individuals with the same score, the individual that has not served as a parent will be selected as the new parent. This procedure is typically repeated for a predefined number of iterations. The logic synthesis is a complex process that has to consider several aspects that are in principle mutually dependent. Two basic scenarios are typically conducted in practice – optimizing the power and/or area under some delay constraints, or optimizing the delay possibly under some power and/or area constraints. Depending on the goal and required precision, the cost function corresponds either with the number of gates, logic depth or a more precise but computationally more complex measure such as area on a chip or circuit delay.

3 The Proposed Approach

Let $\mathcal{C}$ be a combinational circuit described at the level of common gates represented by a Boolean network N consisting of $|N|$ nodes. Each node corresponds with a single gate in $\mathcal{C}$. The pseudo-code of the proposed optimization procedure based on evolutionary resynthesis is shown in Algorithm 1.

Algorithm 1: Optimization of digital circuits using EA-based resynthesis

```
  Input: A Boolean network $N$
  Output: Optimized network $N'$, $cost(N') \leq cost(N)$
1 $N' \leftarrow N$
2 while terminated condition not satisfied do
3     $W \leftarrow$ GetSubcircuit($N$) ;
4     if $W$ is not a suitable candidate then
5         continue
6     $W' \leftarrow$ OptimizeNetworkUsingEA($W$)
7     if cost(($N' \setminus W) \cup W'$) < cost($N'$) then
8         $N' \leftarrow (N' \setminus W) \cup W'$
9 return $N'$
```

We propose to apply an iterative process which consists of a sequence of three steps that are executed in a loop. A working area (Boolean network W) is extracted from the Boolean network N' in the first step. The goal is to obtain a smaller, preferably compact, circuit which is easier to manipulate with. In the next step, each W that is not suitable for the subsequent optimization is skipped. The motivation is to eliminate execution of a relatively time-consuming resynthesis for the windows that are unlikely to lead to any improvement. The identification of the suitable windows can be based on the size of W (small windows are filtered out) or a more advanced metric which reflect, for example, the number of inputs and depth (thin windows are filtered out). In the third step, resynthesis is applied to the extracted Boolean network. The resynthesis is performed by an evolutionary algorithm which produces an optimized version of W denoted as W'. Depending on the success of the optimization, the cost of W' can be either better or the same as the cost of W. Finally, the optimized logic network W' is evaluated with respect to N' and if it exhibits a better parameters, it replaces W in N'. The whole optimization algorithm is terminated when a predefined number of iterations or a given runtime is exhausted.

3.1 Working Area Extraction

Two different approaches to the identification and extraction of a suitable subcircuit corresponding with the procedure GetSubcircuit in Algorithm 1 are pro-

Algorithm GS1: Cut-based procedure GetSubcircuit

```
Input: A Boolean network $N$,
minimum ($c_{min}$) and maximum ($c_{max}$) volume of cut $C$,
minimum ($k_{min}$) and maximum ($k_{max}$) size of cut $C$
Output: A working area $W$
1 $m \leftarrow$ identify the best candidate root node $m \in N$
2 $C \leftarrow$ ReconvergenceDrivenCut($m$, $c_{min}$, $c_{max}$, $k_{min}$, $k_{max}$)
3 $W \leftarrow$ ExpandCutToWindow($m$, $C$)
4 return $W$
```

posed and evaluated. The first implementation is based on the computation of the reconvergence-driven cuts which is the preferred approach applied during logic synthesis. This method, however, may produce subcircuits with a relatively small volume. To avoid this, we propose an alternative approach loosely inspired by the windowing introduced in Section 2.2.

The pseudo-code of the cut-based approach is shown in Algorithm GS1. Firstly a node which may potentially lead to the best improvement of N is determined. Since the identification of this node itself is a nontrivial problem, some heuristic needs to be implemented. The size of transitive fanin cone, level of the node or a more complex information can be used to determine the most suitable candidate. Then, a working area is extracted from the Boolean network. This procedure starts with computation of the reconvergence-driven cut C as described in Section 2.2. From the practical reasons, is also beneficial to limit the size of C to be able to enumerate a large number of sub-circuits in a reasonable time. Hence, we can define four parameters: c_{min} and c_{max} restricting the volume of C ($c_{min} \leq |C| \leq c_{max}$), and k_{min} and k_{max} ($k_{min} \leq k_{max}$) limiting the size of cut (feasibility).

This step is followed by expansion of the cut C into a window W, i.e. expansion of the set of leaf nodes to a set of contained nodes. In addition to the nodes

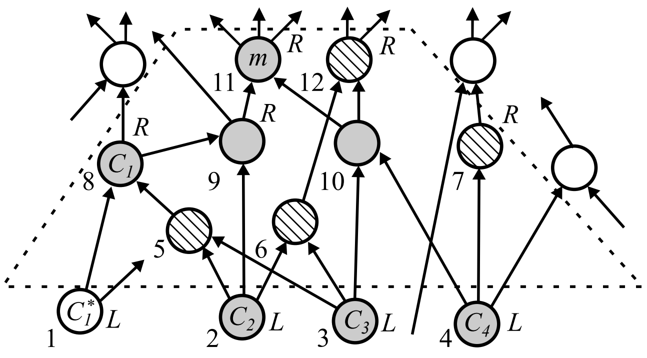


Fig. 3 Example of the window created using the cut-based algorithm GS1. The set of contained nodes of a 4-feasible cut C = $\{C_1, C_2, C_3, C_4\}$ rooted in node m is highlighted using the filled nodes. The hatched nodes are added to the window during the expansion of the cut. As a consequence of that, leave C_1 is replaced by C_1^*. The root and leaves of the window are denoted as R and L, respectively. The nodes in the window have assigned an index (the number located below a particular node) used to uniquely identify each node in the CGP. One of the many possibilities how to encode the window using CGP is for example: (2,3,f_5) (2,3,f_6) (4,1,f_7) (1,5,f_8) (8,2,f_9) (3,4,f_{10}) (9,10,f_{11}) (6,10,f_{12}) (7,8,9,11,12), where $f_i \in \{\text{NOT}, \text{AND}, \text{OR}, \text{XOR}, ...\}$ is the function of the node with index i.

inside the cut, we should consider also all nodes that are not contained in the cut but have fanins inside the cut. Our expansion is similar to that employed in the resubstitution [18] where transitive fanout of C is considered, however, we do not impose any limit on the number of included nodes or their maximum level. The process of cut identification and the subsequent expansion is illustrated in Figure 3.

During the expansion, three set of nodes are created: the set of internal nodes I, the set of leaves L and the set of root nodes R. L contains nodes that will serve as PIs of the temporary network used in the subsequent optimization. Similarly R contains nodes whose outputs have to be connected to POs. Note that R contains not only the root node m but also other nodes whose fanouts are outside of the window (see Fig. 3). It holds that $C \subseteq L$ since the expansion may cause that some leaves of C become a fanout of a node inside the window. Two situations can happen for a leaf node. If all fanins are inside the window, the leaf can be simply removed from L. Otherwise, all fanins of the original leaf node need to be added to L (the case of C_1 in Figure 3). This procedure has to be repeated iteratively to ensure that there are no leaves having a fanin already included the window.

Algorithm GS2: Window-based procedure GetSubcircuit

```
Input: A Boolean network $N$,
minimum ($w_{min}$) and maximum ($w_{max}$) size of $W$
Output: A working area $W$, $w_{min} \leq |W| \leq w_{max}$
$m \leftarrow$ select a random node $m \in N$
init queue $q$ with $m$
$W \leftarrow \emptyset$
while $q$ not empty $\wedge |W| < w_{max}$ do
    $m \leftarrow$ pop a node from $q$
    $W \leftarrow W \cup \{m\}$
    $X \leftarrow fanin(m) \cup fanout(m)$
    push all nodes from $X \setminus W$ that are not already in $q$ into $q$
if $|W| < w_{min}$ then
    $W \leftarrow \emptyset$
$W \leftarrow \bigcup_{m \in W} fanin(m)$
return $W$
```

The pseudo-code of the second approach is given in Algorithm GS2. The process starts with the selection of a node $m \in N$ that will serve as a pivot. The pivot serves as an initial point for the expansion that iteratively marks neighboring nodes of already processed and marked nodes. By neighboring nodes of a node n we mean those belonging to fanin or fanout of that node. This mechanism enables the window to grow to both directions, i.e. towards PIs as well as POs. After a finite number of steps, we obtain a subcircuit W of the required size consisting of the pivot node and its neighbourhood.

To implement the expansion efficiently, we use a queue q whose content is initialized to m. In each iteration, one node is dequeued from q and included in W. Then, the neighboring nodes X (those that are directly connected to m) are identified. Finally, nodes that have not yet been processed and are not already in the queue are enqueued. Two parameters are used to restricting the size of W – w_{min} and w_{max}. The process ends when w_{max} nodes are included in W or no more nodes remain (all nodes surrounding m have been processed and included in W). Subcircuits smaller than w_{min} are ignored. In the final step (line 11 in Algorithm GS2), all the fanins of the nodes included in W are added into W. Then, the leaves of W serve as inputs and roots of W as outputs.

The whole process is illustrated in Figure 4. The procedure starts with node m. In the first iteration, three nodes are pushed into queue, namely q_1, q_2 and q_3. In the second iteration q_1 is enqueued and three additional nodes are queued: q_4, q_5 and q_6. Node m also belongs to $fanout(q_1)$ but this node is already included in W and is thus ignored. In the third iteration, q_2 is enqueued and processed which gives also three new nodes q_7, q_8, and q_9. The process ends when q_{10} is dequeued and included in W. During the finalization phase, nodes having the index 5 and 4 are added into W because these nodes has to serve as new primary inputs. We received a subcircuit with five inputs (nodes denoted with L) and five outputs (output of the nodes denoted as R).

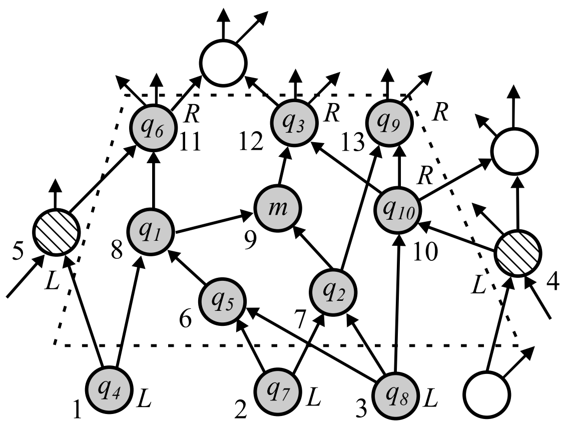


Fig. 4 Example of the window consisting of 10 nodes ($w_{max} = 10$) created using the proposed alternative windowing algorithm GS2. The neighboring nodes added into W are highlighted using the filled nodes. The hatched nodes are those added during the final step. The nodes at the bottom are primary outputs. The root and leaves of the window are denoted as R and L, respectively. The nodes in the window have assigned an index (the number located below a particular node) used to uniquely identify each node in the CGP. The labels q_i inside the nodes denote the order i in which the nodes were chosen.

Both approaches are complementary and have their own advantages and disadvantages. The cut-based windowing algorithm GS1 is in general very sensitive to the root node selection. In some cases, small windows can be produced. This can happen especially when the root node is located close to the primary inputs. The reason is that the cut-based algorithm allows the window grow only towards the primary inputs. Unfortunately, selection of the best root node represents a hard problem. Depending on the structure of the circuits to be optimized, the obtained windows can be narrow and tall.

Identification of the best pivot node in the alternative windowing approach GS2 is also a non-trivial problem but its selection is not as critical as in GS1 because bi-directional expansion is applied in this case. The algorithm allows the window to grow not only towards to the primary inputs but also to the primary outputs. Despite of that, it can easily happen that the iterative procedure produces also unsatisfactory results. This can happen when we select a node with a high number of fanout nodes. In such a case, we receive the required number of nodes in the first iteration because the queue is filled with the necessary number of nodes when visiting the pivot node. Hence depending on the structure of the circuits to be optimized, the windows can be wide but with small depth.

3.2 Evolutionary Optimization

The procedure OptimizeNetworkUsingEA is implemented as follows. At the beginning, the extracted subcircuit (window) is encoded using the 1D CGP encoding. The received chromosome is used to seed the initial population. The evolutionary optimization is then executed for a limited number of iterations (evaluations). The goal is to optimize the initial solution with respect to a chosen cost function. The number of iterations should be determined heuristically according to the size of the initial circuit. The more iterations are allowed, the higher improvement can be achieved. On the other hand, many iterations on a small circuit wastes time. At the end, the best obtained circuit is returned and implanted back into the original Boolean network instead of the original window.

The extracted window is encoded using CGP encoding as follows. All nodes $n \in W$ contained in the window W are sorted in the topological order. We receive a list of nodes having the leaf nodes located at the beginning of the list. Each node is assigned an unique index which is equal to its position in the list. One to one mapping is then used to encode the nodes using CGP encoding. Only non-leaf nodes are encoded in the chromosome because the leaf nodes serve as inputs. It means that the size of the CGP grid is $n_n = |W \setminus L|$. There is no need to introduce any redundancy at this level as shown in [27]. To illustrate the principle, let us consider the window depicted in Figure 3 consisting of 12 nodes. The window is mapped to a 1D array of eight CGP nodes ($n_n = 8$). The inputs are numbered 1 to 4 because four leaf nodes are present. The contained nodes have associated indices 5 to 12. To encode the first node associated with the index 5, for example, the following three genes are used: (2, 3, AND). The first gene encodes the connection of the first input (the node 5 is connected to the output of the leaf node 2), the second gene encodes the connection of the second output and the third gene encodes the function of the node assuming that the node 5 is AND gate. Five genes are used at the end of the chromosome to encode the output connections corresponding with five root nodes denoted as R. In summary, the window is encoded using a string of $8 \times 3 + 5 = 29$ genes.

Let C be a candidate solution (circuit) created by mutating a parental solution P. The fitness of the candidate solution $fitness(C)$ is determined as

$$fitness(C) = \begin{cases} cost(C), & \text{if } f(C) \equiv f(P). \\ \infty, & \text{otherwise,} \end{cases} \tag{1}$$

where $cost(C)$ is a cost function to be minimized, $f(C)$ is a Boolean function representing C and $f(P)$ is a Boolean function corresponding with P. Candidate circuits violating the requirement for the functional equivalence, i.e. those for that $f(C) \equiv f(P)$ is violated, are assigned a high positive value and are discarded. Depending on the scenario, the cost function can reflect the number of gates, area on a chip, logic depth, delay or power consumption.

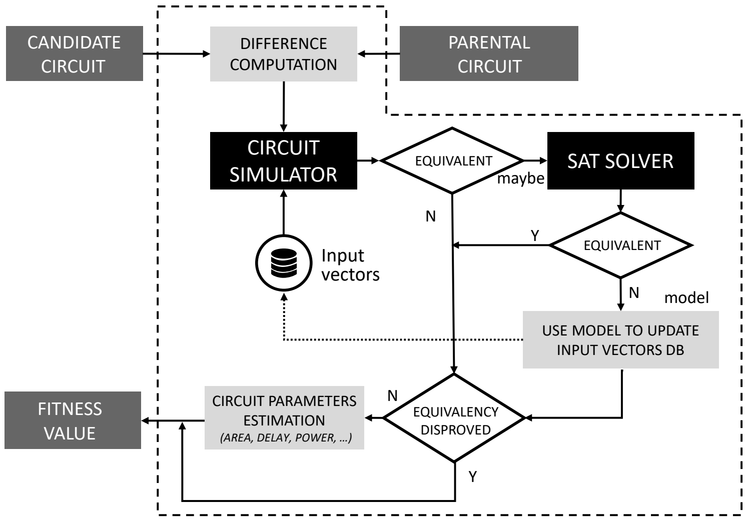


Fig. 5 Principle of the fitness score computation using the hybrid approach combining a circuit simulator with a SAT solver.

The computation of the fitness score is implemented as suggested in [27]. The overall principle is illustrated in Figure 5. The process begins with the computation of the difference between a candidate and parental circuit. The difference is computed at the level of the phenotypes, i.e. Boolean networks, and its purpose is to enable equivalence checking, i.e. to check whether the candidate solution is functional equivalent with its parent. Only the functionally equivalent solution is further analysed to determine its cost. In order to perform the equivalence checking as quick as possible, we combine a SAT solver with a circuit simulator to avoid excessive runtimes caused by some hard-to-solve SAT instances. The key idea is to use a small number of input vectors to disprove the equivalence using a fast circuit simulator. If the candidate circuit produces a different output value compared to the parental circuit serving as a reference, we can terminate the fitness calculation because the candidate circuit violates the specification. If the output values are the same, we have to use a SAT solver to prove that there is no input assignment that produces different output values. Randomly generated input vectors have been used in [27]. In this work, we use a slightly advanced version where we feed the simulation engine with counter examples produced by the SAT solver. This mechanism helps to further improve the overall efficiency.

4 Experimental Setup

The proposed method was implemented in C++ as a part of Yosys open synthesis suite [30]. The advantage of this tool, among others, is that it allows us to directly manipulate with Verilog files and that it integrates ABC [1], a state-of-the-art academic tool for hardware synthesis and verification.

The goal of this work is to evaluate the performance of the proposed approach and compare the results with the state-of-the-art evolutionary as well as conventional method for optimization of digital circuits. In particular, we consider two variants of Algorithm 1 that differ in the implementation of the procedure GetSubcircuit. The first one (denoted as GS1) is based on Algorithm GS1 and the second one (denoted as GS2) is based on Algorithm GS2. The state-of-the-art is represented by the EA-based optimization technique that optimizes the whole Boolean network at once [27]. This approach will be denoted as global. To represent the conventional tools, we chose ABC synthesis tool which is considered to be the best academia tool implementing the state-of-the-art synthesis algorithms.

The methods are evaluated on a recent set of benchmark circuits coming from the logic synthesis community. The benchmark set consists of 28 real-world circuits available in the form of Verilog netlists[1]. Nineteen instances are various controllers taken from IWLS'05 Open Cores benchmarks. The remaining nine instances represent common arithmetic circuits. At the beginning, all the instances were deeply optimized by ABC (hundred iterations of 'resyn' script) to make sure that our optimization algorithms start with the best results produced by the conventional synthesis. The optimized circuits were then mapped to gates (ABC command 'map') using a library of common 2-input gates including XORs/XNORs gates and exported back to Verilog. The mapped Verilog netlists then served as input to the EA-based methods. Compared to the ABC, the EA-based methods operate directly at the level of gates. The gate-level representation was chosen intentionally because it enables to avoid the bias of the AIG representation and better exploit the XOR decomposition.

Area-optimization is targeted in this work. It means that the only criterion in the fitness function considered in this paper is the area on a chip expressed as the number of gates. It means that the improvement is measured in terms of the number of removed gates. The other electrical parameters such as delay or power consumption are not reflected. The line 7 of Algorithm 1 thus reduces to $|W'| < |W|$ which is much simpler to evaluate. For each method and each benchmark, five independent runs were executed to obtain statistically valid results. All of the optimized circuits were formally verified with respect to their original form (ABC command 'cec') to avoid any error in the evaluation.

The procedure OptimizeNetworkUsingEA is based on the CGP implemented as described in Section 2.3 and Section 3.2 with the following parameters: $n_a = 2$, $\lambda = 1$, $h = 2$, $n_n = |W|$, $\Gamma = \{\mathrm{BUF}, \mathrm{NOT}, \mathrm{AND}, \mathrm{OR}, \mathrm{XOR}, \mathrm{NAND}, \mathrm{NOR}, \mathrm{XNOR}\}$. The CGP parameters were chosen in accordance with [27]. The termination conditions are designed as follows. The proposed method is allowed to execute n_{iters} iterations. Each iteration corresponds with a single execution of the OptimizeNetworkUsingEA procedure. This procedure terminates either when a given number of evaluations (n_{evals}) is exhausted or when a predefined amount of time (t_{max}) has

[1] The Verilog netlists of the benchmark circuits are taken from https://lsi.epfl.ch/MIG

elapsed. The latter condition helps to ensure a good scalability and predictability of the worst-case CPU time of the optimization which could be enormous especially in those cases when many hard-to-solve candidate solutions are generated during the evolution. The global method terminates either when $n_{evals} \times n_{iters}$ evaluations are exhausted or when the CPU time reaches $t_{max} \times n_{iters}$ seconds. The strategy with the fixed number of evaluations is relatively naïve because it supposes that the computation effort does not depend on the size of the window. On the other hand, it helps to fairly evaluate all evolutionary methods because they are allowed to evaluate the same number of candidate solutions. We chose $n_{iters} = 2 \times 10^4$, $n_{evals} = 5 \times 10^5$, and $t_{max} = 10$ seconds in this work. This setup ensures that 10^{10} candidate solutions are generated and evaluated.

4.1 Parameter Setting

As both algorithms for sub-circuit extraction contain parameters that may have a huge impact on the efficiency of the optimization process, we need to ensure proper parameter configuration. To perform a fair evaluation, we ran experiments that help us to identify a suitable parameter setting. Due to the increased computational complexity, we conducted the experiments on a limited set of benchmark circuits. We selected three benchmarks from each class of circuits to have a small yet representative set of circuits[2].

Table 1 Impact of c_{min} and c_{max} parameters on the performance of the evolutionary optimization based on GS1 algorithm evaluated on a subset of six benchmark circuits. The best results in each row are highlighted.

	c_{min} / c_{max}						
	5/10	10/20	20/35	35/50	50/75	75/100	5/1000
achieved improvement	5.5%	6.2%	6.5%	6.5%	6.4%	4.7%	*8.2%*
controllers & logic	3.6%	4.2%	4.3%	4.2%	4.0%	2.3%	*5.5%*
arithmetic circuits	7.4%	8.2%	8.8%	8.8%	8.8%	7.1%	*10.9%*
iterations caused reduction	2.8%	3.0%	3.3%	3.3%	3.2%	2.4%	*4.1%*
controllers & logic	2.6%	2.7%	2.8%	2.6%	2.5%	1.7%	*3.4%*
arithmetic circuits	3.0%	3.2%	3.8%	4.0%	3.9%	3.2%	*4.8%*
iterations when EA time-outed	*0.0%*	*0.0%*	*0.0%*	*0.0%*	0.7%	10.5%	1.8%
controllers & logic	*0.0%*	*0.0%*	*0.0%*	*0.0%*	0.5%	13.4%	2.0%
arithmetic circuits	*0.0%*	*0.0%*	*0.0%*	*0.0%*	0.9%	7.6%	1.6%

Four parameters are present in Algorithm GS1. Parameters k_{min} and k_{max} control the feasibility of the cuts. These parameters are fixed to 1 and 10000, respectively, because our SAT-based CGP optimizer does not need to put any restriction on the number of circuit inputs. The next two parameters c_{min} and c_{max} determine the size (i.e. the number of gates) of the extracted sub-circuits. We hypothesize that larger sub-circuits may lead to higher number of reduced gates in the sub-circuits and better improvement at the global level. To confirm

[2] The following circuits were used to determine the best parameter setting: dsp, mem_ctrl, tv80, diffeq1, max, revx

this hypothesis and identify a suitable setting, we run many experiments with different values of c_{min} and c_{max}. Results for some settings are summarized in Table 1. Three efficiency indicators were established and analysed. The first three rows report the average number of removed gates calculated over all benchmark circuits (first row) and for each class separately (second and third row). The next three rows report the average number of iterations that caused a reduction in the number of gates. The last three rows show the average number of iterations that produced a sub-circuit whose optimization by CGP time-outed. We firstly tried to restrict the size of the sub-circuits to a relative narrow range. The numbers shown in the first six columns, however, suggest that this strategy does not offer any advantage. The average improvement stagnates and does not increase with increasing the c_{min} and c_{max}. The achieved reduction in the number of gates is around 8% for the arithmetic benchmarks and 4% for the logic benchmarks. For higher values ($c_{min} = 75$, $c_{max} = 100$), we can observe 1.7% drop in the performance (average improvement is 4.7% vs 6.4%). More than 10% iterations, on the average, were terminated prematurely due to the t_{max} restriction for this setting. This behavior is caused by the fact that many hard-to-solve instances were generated. It means that the computationally expensive SAT solver needed to be used to decide equivalence of many complex candidate solutions. As a consequence of that, less than $1 \cdot 10^{10}$ candidate solutions were generated and evaluated in those cases. Interestingly, the most advantageous setting was the least restrictive one where we chosen $c_{min} = 5$ and $c_{max} = 1000$. The lower bound prevents the cut-based algorithm to generate too small sub-circuits. The upper bound was chosen to be a value higher than the largest volume that was ever observed on the reduced benchmark set across all experiments. This setting in practice means that no restrictions are applied at all.

Table 2 Impact of w_{min} and w_{max} parameters on the performance of the evolutionary optimization based on GS2 algorithm evaluated on a subset of six benchmark circuits. The best results in each row are highlighted.

	w_{min} / w_{max}					
	5/10	5/20	5/50	5/100	5/1000	5/10000
achieved improvement	7.4%	9.0%	12.6%	*14.4%*	12.9%	12.2%
controllers & logic	3.9%	5.1%	6.9%	8.8%	13.5%	*23.5%*
arithmetic circuits	10.8%	12.9%	18.3%	*19.9%*	12.2%	0.8%
iterations caused reduction	18.7%	19.1%	*23.2%*	17.8%	5.1%	2.2%
controllers & logic	32.4%	31.0%	*37.5%*	27.9%	5.1%	4.3%
arithmetic circuits	4.9%	7.1%	*8.9%*	7.8%	5.0%	0.1%
iterations when EA time-outed	5.6%	16.2%	*5.3%*	13.2%	76.2%	95.8%
controllers & logic	*0.0%*	7.7%	5.8%	13.1%	66.5%	92.3%
arithmetic circuits	11.2%	24.7%	*4.9%*	13.2%	85.9%	99.4%

Note that the root node m is chosen randomly. This strategy simplifies the problem but it may lead to degradation of the performance especially if many unacceptable windows are produced. If this happens in 10% cases, for example, the total number of effective generations is in fact reduced to 90%. Interestingly, we didn't observed such degradation. This situation happened only in less than ten iterations.

Algorithm GS2 has only two parameters, namely w_{min} and w_{max}, that have the same meaning as c_{min} and c_{max} in Algorithm GS1. Similarly to the cut-based algorithm, we tried to identify the best values of these parameters. The results of the experiments on a reduced set of benchmark circuits are summarized in Table 2. Only the cases where w_{min} is fixed to the lower bound are listed. Compared to Algorithm GS1, however, much larger windows has to be accepted because of the construction of the sub-circuits. The method produces natively larger windows because all fanins and fanouts are included in the list of potential nodes in each iteration of the windowing algorithm. As shown in the first row of Table 2, the efficiency of the optimization increases with increasing w_{max} and it culminates for $w_{max} = 100$. For sub-circuits having ten times higher number of gates, i.e. $w_{max} = 1000$, the average number of removed gates drops down to 12.9%. In this case, majority of the CGP runs timed out. The results presented in the last three rows suggests that Algorithm GS1 produces sub-circuits that are more complex compared to the cut-based method which tends to produce structures having a tree-like shape. The choice of the best setting is not as apparent as for GS1 because it depends on the preferred criteria. As we are primarily interested in the best gate improvement, we decided to use $w_{min} = 5$ and $w_{max} = 100$ for the following experiments.

According to the obtained results, it can be concluded that GS2 performs significantly better even though there is a relative high amount of premature terminated CGP runs. The best result was obtained for $w_{max} = 100$. In this case, the method was able to reduce the optimized netlists by 14.4% in average. The best reduction for the cut-based approach is 8.2% and it was achieved when $c_{max} = 1000$.

5 Results

The results from running each method on each problem with the best parameter setting identified in the previous section are summarized by Table 3. The first three columns contain information related to the benchmarks: circuit name, the number of circuit inputs (PIs), and the number of circuit outputs (POs). The next two columns show parameters of the optimized and mapped circuits produced by ABC. In particular, the number of gates and logic depth are given and those numbers serve as a baseline for our comparison. Then, the achieved improvement expressed as the relative reduction with respect to the baseline is reported for the global and both proposed methods. For each method, we report not only the median (section average improvement) but also the best obtained results (section best improvement). The statistics is based on all five independent runs. For each group of circuits, the mean improvement is provided. The values in the sixth, seventh and eight column are calculated from all runs. The values in the remaining columns are calculated from the data in the table.

All the evolutionary approaches were able to further reduce the size of the benchmark circuits despite that they were highly optimized by the ABC synthesis tool. On the average, the evolutionary resynthesis achieved 8.9% circuit size reduction on controllers and 21.4% reduction on arithmetic circuits. The best results obtained by a particular method are relatively close to the average ones which suggests that the evolutionary methods are quite stable although they are in principle non-deterministic. According to the number of highlighted cases showing the

Table 3 Comparison of the evolutionary methods (global and both proposed) against ABC. The columns 'improvement' report the relative improvement in the number of gates compared to the optimized circuits obtained using ABC whose parameters are shown in column 'ABC'. The median is used to determine the average improvement.

			ABC		**Average improvement**			**Best improvement**		
Benchmark	**PIs**	**POs**	**gates**	**delay**	global	GS1	GS2	global	GS1	GS2
ac97_ctrl	2255	2136	11433	10	1.2%	2.7%	*3.1%*	1.8%	2.9%	*4.0%*
aes_core	789	532	21128	20	0.1%	2.9%	*5.3%*	*5.6%*	2.9%	5.5%
des_area	368	70	5199	25	2.0%	*5.5%*	4.8%	2.6%	*6.0%*	5.2%
des_perf	9042	1654	78972	16	0.0%	1.8%	*4.2%*	0.1%	1.8%	*5.8%*
dsp	4223	3792	43491	45	0.0%	*3.4%*	1.8%	0.0%	*3.6%*	3.5%
ethernet	10672	10452	60413	23	0.0%	0.4%	*1.5%*	0.0%	0.6%	*1.7%*
i2c	147	127	1161	12	10.3%	8.3%	*17.9%*	10.7%	9.1%	*18.3%*
mem_ctrl	1198	959	10459	24	*25.4%*	6.9%	10.0%	*26.1%*	7.0%	12.2%
pci_bridge32	3519	3136	19020	21	0.6%	3.4%	*3.6%*	1.3%	3.5%	*4.6%*
pci_spoci_ctrl	85	60	1136	15	36.9%	17.0%	*37.1%*	38.0%	18.3%	*39.1%*
sasc	133	123	746	8	2.5%	5.8%	*6.8%*	2.8%	6.4%	*7.2%*
simple_spi	148	132	822	11	3.9%	4.7%	*6.6%*	4.4%	5.2%	*7.2%*
spi	274	237	3825	26	*13.9%*	5.5%	8.7%	*25.1%*	6.7%	9.6%
ss_pcm	106	90	437	7	2.1%	*5.0%*	4.8%	2.3%	*5.5%*	5.3%
systemcaes	930	671	11352	27	0.0%	*11.0%*	10.6%	0.0%	11.9%	*14.7%*
systemcdes	314	126	2601	25	10.6%	4.4%	*15.5%*	11.6%	4.7%	*16.3%*
tv80	373	360	8736	39	10.3%	6.0%	*14.2%*	*14.7%*	6.5%	14.3%
usb_funct	1860	1692	15405	23	2.6%	5.6%	*9.6%*	4.8%	5.8%	*11.3%*
usb_phy	113	73	452	9	11.9%	13.4%	*16.8%*	12.2%	13.7%	*17.7%*
average (controllers & logic)			15620.4	20.3	7.5%	5.7%	*8.9%*	8.6%	6.4%	*10.7%*
diffeq1	354	193	20719	218	0.0%	11.3%	*26.5%*	0.0%	11.5%	*28.6%*
div16	32	32	5847	152	0.0%	15.2%	*27.4%*	0.0%	16.0%	*64.2%*
hamming	200	7	2724	80	11.8%	28.3%	*58.6%*	14.6%	32.9%	*59.9%*
mac32	96	65	7793	55	0.0%	7.8%	*10.1%*	0.0%	9.9%	*10.6%*
max	512	130	3719	117	0.6%	*7.2%*	5.1%	0.9%	*7.4%*	5.2%
mul32	64	64	8225	42	0.0%	16.2%	*20.9%*	0.0%	16.5%	*21.4%*
mult64	128	128	21992	190	0.0%	5.5%	*6.3%*	0.0%	5.9%	*8.4%*
revx	20	25	8130	171	0.0%	13.9%	*22.8%*	0.1%	14.5%	*27.1%*
sqrt32	32	16	1462	307	3.0%	*21.7%*	16.3%	5.1%	*22.8%*	20.9%
average (arithmetic circuits)			8956.8	148.0	1.8%	12.6%	*21.4%*	2.3%	15.3%	*27.4%*

best results in each section of Table 3, the method GS2 introduced in this paper is the clear winner. Nevertheless, both methods mentioned in this work perform substantially better considering the average as well as the best results compared to the global method. Method GS1 won in 21 out of 28 cases. Method GS2 won in 24 cases. There are even cases, when the global method provided none or nearly none improvement (see benchmarks 'des_perf', 'dsp', 'ethernet', 'systemcaes'). Looking at the arithmetic circuits, the global method was able to slightly improve only two circuits – 'hamming' and 'sqrt32'. In other cases, the reduction is negligible. There are, however, two problem instances (controller 'mem_ctrl' and 'spi') for that the global method provided very competitive results. In addition there are three cases ('aes_core', 'pci_spoci_ctrl', 'tv80') where the global method produced results that are very close to the best one obtained by the proposed methods. The common feature of these five cases is a very steep convergence curve (see Figure 6 which contains the convergence curve for 'spi' controller). We tried to identify the exact reason for that but it looks that such a behaviour is a result of the combination of several factors. It can be concluded, in general, that the global method works

Table 4 The average number of generated and evaluated candidate solutions needed to achieve 1%, 5%, and 10% reduction. The median is used to determine the average number of evaluations as well as the average in the summary rows provided for each class of circuits. The average value in the summary is determined only from the successful runs, i.e. those that leaded to the required reduction.

	1% improvement			**5% improvement**			**10% improvement**		
Benchmark	global	GS1	GS2	global	GS1	GS2	global	GS1	GS2
ac97_ctrl	4.8×10^8	9.6×10^8	1.2×10^8	–	–	–	–	–	–
aes_core	2.1×10^8	2.1×10^9	6.8×10^8	1.6×10^9	$> 10^{10}$	8.8×10^9	–	–	–
des_area	5.3×10^7	9.7×10^8	3.9×10^8	$> 10^{10}$	7×10^9	8.6×10^9	–	–	–
des_perf	$> 10^{10}$	3.4×10^9	1×10^9	$> 10^{10}$	$> 10^{10}$	7.6×10^9	–	–	–
dsp	$> 10^{10}$	8×10^8	4.6×10^8	–	–	–	–	–	–
ethernet	$> 10^{10}$	$> 10^{10}$	2.4×10^9	–	–	–	–	–	–
i2c	3×10^5	6.7×10^7	2.5×10^6	8.6×10^6	6.8×10^8	2.9×10^7	2.9×10^9	$> 10^{10}$	1.6×10^8
mem_ctrl	1.4×10^4	2.6×10^8	1.5×10^8	8.1×10^4	4.5×10^9	1.6×10^9	2.4×10^5	$> 10^{10}$	6.4×10^9
pci_bridge32	1.4×10^9	3.1×10^8	2.3×10^8	–	–	–	–	–	–
pci_spoci_ctrl	1.2×10^4	3×10^7	2.5×10^6	1.7×10^5	2.3×10^8	1.7×10^7	7.3×10^5	6.4×10^8	4.9×10^7
sasc	2.2×10^7	2.7×10^7	1.8×10^7	$> 10^{10}$	8.6×10^8	1.7×10^9	–	–	–
simple_spi	6.6×10^6	3.9×10^7	1.1×10^7	$> 10^{10}$	2.8×10^9	7.6×10^8	–	–	–
spi	5.5×10^6	1×10^8	5×10^7	7.3×10^7	3.9×10^9	9.4×10^8	2.1×10^8	$> 10^{10}$	$> 10^{10}$
ss_pcm	9.3×10^6	1.1×10^8	2.2×10^7	$> 10^{10}$	2.1×10^9	6.6×10^9	–	–	–
systemcaes	$> 10^{10}$	2.1×10^8	1.1×10^8	$> 10^{10}$	1.7×10^9	1.1×10^9	$> 10^{10}$	6.5×10^9	3.5×10^9
systemcdes	5.8×10^6	2.4×10^8	4.2×10^7	5.9×10^7	$> 10^{10}$	4.5×10^8	1.7×10^9	$> 10^{10}$	1.8×10^9
tv80	4.2×10^4	2.3×10^8	8.9×10^7	1.9×10^7	4.8×10^9	6.4×10^8	2.2×10^8	$> 10^{10}$	2.8×10^9
usb_funct	9.7×10^7	3×10^8	7.5×10^7	$> 10^{10}$	6.6×10^9	8.4×10^8	$> 10^{10}$	$> 10^{10}$	5.1×10^9
usb_phy	6.2×10^4	4.3×10^6	1.5×10^6	2.2×10^6	5.6×10^7	4×10^6	6.3×10^8	3.8×10^8	3.8×10^7
average	5.4×10^6	2.2×10^8	8.7×10^7	7.5×10^6	1.7×10^9	7.4×10^8	2.1×10^8	6.3×10^8	1.6×10^9
success rate	78%	94%	*100%*	53%	80%	*100%*	77%	33%	*88%*
diffeq1	$> 10^{10}$	2.2×10^8	6.2×10^7	$> 10^{10}$	1.7×10^9	3.4×10^8	$> 10^{10}$	6.8×10^9	8.1×10^8
div16	$> 10^{10}$	8.5×10^7	1.9×10^7	$> 10^{10}$	6.3×10^8	9.3×10^7	$> 10^{10}$	2.6×10^9	2.3×10^8
hamming	3.6×10^4	1.9×10^7	5×10^6	4×10^5	1.8×10^8	2.5×10^7	1.5×10^6	6.1×10^8	4.8×10^7
mac32	$> 10^{10}$	6.7×10^7	7.2×10^7	$> 10^{10}$	7×10^8	8.5×10^8	$> 10^{10}$	$> 10^{10}$	7.2×10^9
max	$> 10^{10}$	1.1×10^8	1×10^8	$> 10^{10}$	1.2×10^9	6.2×10^9	–	–	–
mul32	$> 10^{10}$	7.7×10^7	2.3×10^7	$> 10^{10}$	4.9×10^8	1.4×10^8	$> 10^{10}$	1.6×10^9	5.6×10^8
mult64	$> 10^{10}$	3.7×10^8	1.2×10^8	$> 10^{10}$	6.8×10^9	1.4×10^9	–	–	–
revx	$> 10^{10}$	1×10^8	2.3×10^7	$> 10^{10}$	8.6×10^8	1.4×10^8	$> 10^{10}$	3.3×10^9	3.8×10^8
sqrt32	3.4×10^5	2.5×10^7	5×10^6	3.8×10^6	1.4×10^8	3×10^7	$> 10^{10}$	4.8×10^8	9.1×10^7
average	4.9×10^4	8.4×10^7	2.5×10^7	4.3×10^5	6.9×10^8	1.4×10^8	1.5×10^6	1.7×10^9	3.8×10^8
success rate	22%	*100%*	*100%*	22%	*100%*	*100%*	14%	85%	*100%*

well especially for small instances that are compact (do not contain many independent sub-circuits) and that have a reasonable depth (10 to 25 levels). On the other hand, the optimization of circuits having a large depth, many gates or many independent sub-parts performs unsatisfactory when the global method is applied.

All the evolutionary approaches were able to improve the original circuit substantially. A significant improvement was recorded for the arithmetic circuits. The number of gates was reduced by 27.4% using GS2 (15.3% for GS1) on the average. The highest improvement, 59.9%, was recorded for the 'hamming' benchmark. The detailed analysis revealed that this was possible due to better handling of XORs/XNORs compared to the conventional synthesis. The relative number of AND/OR/NAND/NOR gates remained nearly the same (around 74%) but the absolute number of XORs/XNORs increased from 10% to 15% for GS1 and 18% for GS2.

A more detailed analysis is provided in Table 4 showing the computational effort required to reduce the benchmark circuits by 1%, 5% and 10%. The computation effort is expressed as the average number of generations that have to be evaluated to obtain a circuit whose number of gates is reduced by a given level. The number of evaluations corresponds with the real number of evaluated candidate solutions. It means that we reflected the fact that the CGP may be prematurely terminated due to the time limit. The empty cells in the table mean that none of the evolutionary runs produced a circuit satisfying the required condition. This can happen either because it is in principle impossible to obtain such a circuit (we are already at the optimum or close to the optimum) or because of the insufficient number of evaluations (n_{evals}) or iterations (n_{iters}). The cells containing the value $> 10^{10}$ indicate that it was impossible to reduce the number of gates to the required level within the allowed number of evaluations but it may happen that the required reduction can be achieved when more than 10^{10} evaluations are used.

If we compare the computation effort required for reduction by 1% shown in the first section of Table 4, we can easily identify that the global method converges faster compared to GS1 and GS2. On the other hand, the globally applied CGP has tendency to stuck at a local optima especially when complex benchmarks are optimized. The global method applied to the controllers and logic benchmarks was successful in 78% cases. In the remaining cases, no result was obtained within the allowed number of evaluations. A complete different situation can be observed for the arithmetic circuits. Nearly none improvement was achieved in this category of circuits. The benchmark circuits 'hamming' and 'sqrt32' represent the only exception where the evolution ended successfully. The proposed GS1 and GS2 exhibit a slow convergence but the iterative principle makes them more robust and less likely to converge prematurely to local optima. If we compare the success rate, it is evident that the EA-based resynthesis exhibits better overall performance. Method GS2 achieved the required reduction in all cases. Method GS1 performs similarly. The only failure is in the case of 'ethernet' benchmark circuit. Considering the computation effort, the proposed GS2 typically requires lower number of generations than GS1. The superiority of GS2 over GS1 is more evident in the last section of Table 4 showing the computation effort required for reduction by 10%. GS1 significantly outperforms the other methods on logic as well as arithmetic circuits.

The performance of the evolutionary methods can also be investigated by comparing the corresponding convergence curves. Figure 6 shows the exemplary convergence curves. The first row illustrates the situation typical for the majority of the benchmarks. It corresponds with the situation when the proposed method GS2 clearly outperforms the remaining two methods; it converges faster and achieves better reduction. Global method exhibits a quick convergence but the search mostly ends at a local optima. This is the case of 'usb_phy'. For arithmetic circuits, no improvement was achieved due to the complex circuit structure. The second row illustrates what usually happened for instances where GS1 provided better results than GS2. We identified two different causes. Optimization based on Algoritm GS1 performs better because it profits from the usage of smaller subcircuits. The smaller sub-circuits require less computational effort to be optimized compared to the larger ones. Such a behavior was observed for 'max', 'des_area', 'dsp', 'ss_pcm' and 'max' benchmark. A different situation happened in case of

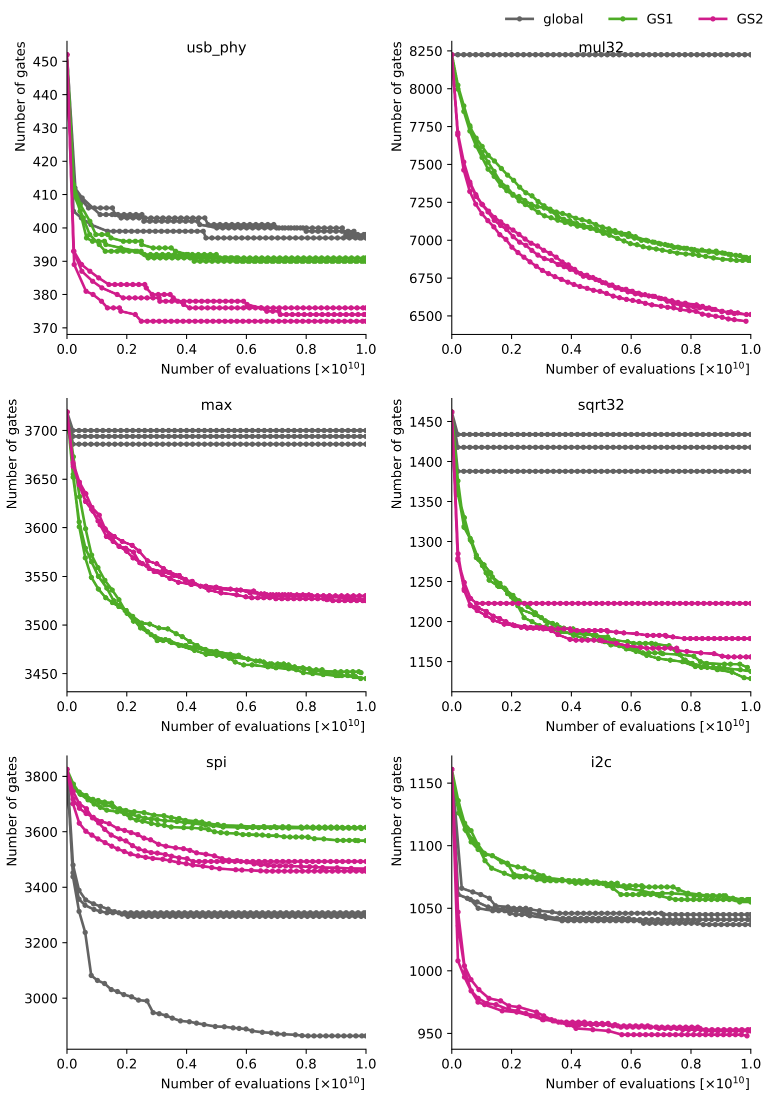


Fig. 6 The exemplary convergence curves representing the typical progress of the fitness score observed during the evolutionary optimization of digital circuits. Records from three independent evolutionary runs are shown in each figure. The lower number of gates, the better result. The data are downsampled to improve the readability. Each curve consists of up to 50 points.

'sqrt32' benchmark. We suppose that GS2 modified the original circuit in such a way that it was hard to further improve it. Considering the space of all valid circuit structures, the method probably reached a local optima that is hard to overcome. The last row in Figure 6 shows two examples where the global method achieved better results than at least single proposed method. The left part shows the typical progress observed in the case of the 'spi', 'aes_core', 'mem_ctrl', and

Table 5 The average number of inputs, outputs and size of the sub-circuits produced by the implemented windowing algorithms. The numbers are reported separately for sub-circuits that were successfully optimized (column 'improved sub-circuits') and the remaining ones where the CGP was not successful (column 'unchanges sub-circuits'). The median is reported for all columns entitled 'avg' prefix.

	improved sub-circuits								**unchanged sub-circuits**							
	avg PIs		avg POs		avg $\|W\|$		max $\|W\|$		avg PIs		avg POs		avg $\|W\|$		max $\|W\|$	
Benchmark	GS1	GS2	GS1	GS2	GS1	GS2	GS1	GS2	GS1	GS2	GS1	GS2	GS1	GS2	GS1	GS2
ac97_ctrl	6	48	6	60	11	100	63	100	5	54	7	67	9	100	62	100
aes_core	6	53	8	98	12	100	185	100	6	55	12	97	14	100	271	100
des_area	13	80	16	99	27	100	282	100	14	89	15	104	29	100	351	100
des_perf	10	39	11	84	19	100	90	100	13	40	14	85	22	100	88	100
dsp	17	74	16	87	32	100	282	100	18	74	17	90	29	100	496	100
ethernet	20	60	24	89	29	100	114	100	19	63	23	95	27	100	429	100
i2c	9	40	9	69	14	100	61	100	7	40	8	58	10	100	58	100
mem_ctrl	20	83	17	97	35	100	280	100	15	83	15	90	24	100	366	100
pci_bridge32	11	61	11	82	19	100	274	100	11	63	12	80	18	100	287	100
pci_spoci_ctrl	19	58	19	88	31	100	92	100	18	58	16	89	24	100	100	100
sasc	4	26	5	30	8	90	27	100	5	30	6	32	7	100	27	100
simple_spi	7	47	7	61	12	100	41	100	6	52	6	59	7	100	57	100
spi	14	68	15	87	27	100	161	100	10	70	11	85	19	100	175	100
ss_pcm	5	17	4	17	7	41	24	100	6	22	5	23	7	52	26	100
systemcaes	11	76	11	78	18	100	154	100	9	67	10	78	15	100	156	100
systemcdes	20	66	20	96	38	100	122	100	17	66	18	92	31	100	139	100
tv80	15	73	21	98	33	100	244	100	15	79	18	96	26	100	239	100
usb_funct	9	52	9	77	14	100	156	100	9	64	9	75	15	100	206	100
usb_phy	7	17	6	28	11	63	33	100	6	25	6	44	8	89	32	100
diffeq1	26	64	25	98	51	100	268	100	26	67	24	99	47	100	302	100
div16	25	51	23	100	42	100	181	100	26	69	23	100	40	100	207	100
hamming	24	60	22	95	40	100	216	100	26	60	22	93	39	100	255	100
mac32	14	64	16	102	29	100	487	100	15	70	18	109	31	100	679	100
max	13	71	8	72	20	100	208	100	16	74	8	72	23	100	209	100
mul32	17	63	16	91	36	100	435	100	17	63	15	91	31	100	422	100
mult64	31	71	23	94	52	100	264	100	14	74	18	93	33	100	402	100
revx	33	73	29	109	55	100	227	100	33	77	29	112	52	100	257	100
sqrt32	23	70	19	96	41	100	136	100	26	66	22	99	45	100	167	100

'tv80'. The common feature is the steep convergence of the global method. The chosen 'spi' benchmark represents, however, a bit exceptional case because we can observe how the global method can stuck at a local optima. As evident also from Table 3, there is a huge difference between the best and the average result. This is caused by the fact that only one run ended in the global optima (less than 2900 gates). We assume that the remaining four runs followed a bad direction in the search space and stuck at a local optima (see the divergence around 3400 gates). The right part of the last row shows the convergence curves that were observed for the following benchmarks: i2c, pci_spoci_ctrl, systemcdes. In this case the global method provided results that are better than those obtained by GS1 but worse than those obtained by GS2.

As we already mentioned in the previous part, the evolutionary resynthesis converges sometimes slowly compared to the CGP working at the global level. We assume that the slow convergence is caused by the fact that each sub-circuit produced by the proposed windowing algorithm is optimized for a fixed number

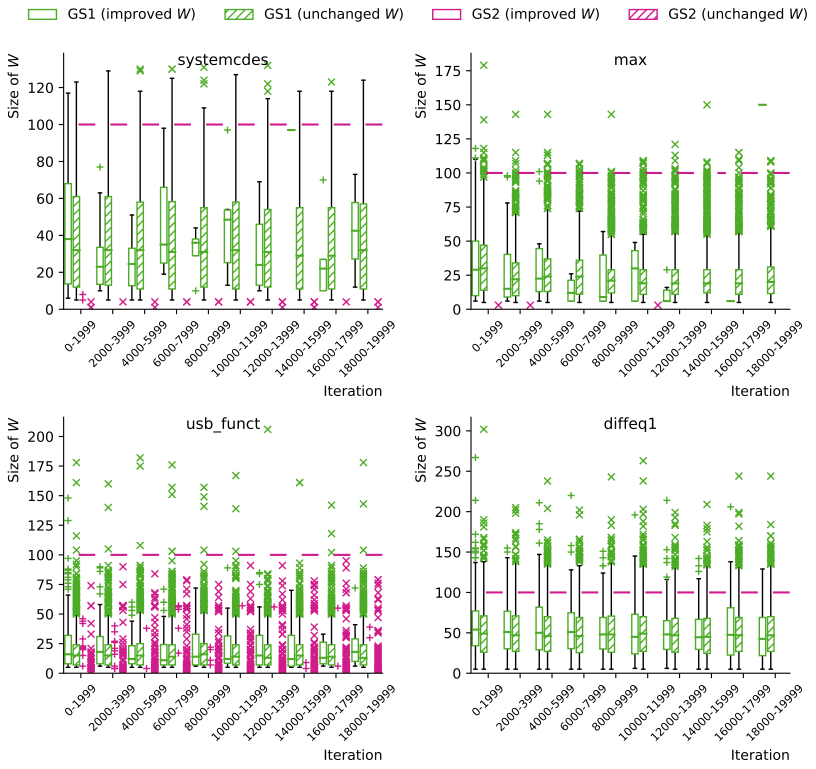


Fig. 7 Size of the sub-circuits extracted from the benchmark circuits in course of the optimization. Data from a single evolutionary run are plotted for each benchmark circuits. The boxes visualize distribution of $|W|$ for sub-circuits generated in 2000 consecutive iterations. Outliers are plotted as individual points ($+$ for successfully optimized sub-circuits, $\times$ for the sub-circuits that left unchanged). Note that the boxes are reduced to a single line and outliers in case of GS2.

of generations independently on its parameters such as the size or the number of PIs. This simplifies the problem but it may lead to a potential inefficiency. Many generations can be wasted to optimize small circuits. In order to elaborate on this problem, we logged all created sub-circuits (W in Algorithm 1) and analyzed their size and other parameters. The parameters of the sub-circuits produced by the proposed sub-circuit extraction algorithms are given in Table 5. The table contains the average number of inputs and outputs, and the average as well as the maximum size of the sub-circuits produced by the proposed windowing algorithms. Note that the leave nodes are not considered in the size. These numbers are provided separately for the case when $|W'| < |W|$ (CGP reduced the sub-circuit) and for the case when $|W'| = |W|$ (CGP kept the sub-circuit unchanged considering the number of gates). Method GS2 mostly produces windows having their size equal to w_{max}. Depending on the circuits structure, however, it may be impossible to create such a large working window because there may be independent parts that consist of the smaller number of gates. This was observed massively during the optimization of the following three benchmark circuits: 'sasc', 'ss_pcm', 'usb_phy'.

According to Table 5, windows having less than 100 nodes were generated in more than half of the total number of iterations for those cases (please refer to the column 'avg $|W|$'). This does not mean, however, that this situation did not occurred for the remaining benchmarks. Figure 7 shows boxplots of $|W|$ for four selected evolutionary runs. The smaller windows, represented by the outliers in the boxplots, were generated in many cases also for 'usb_funct'.

Surprisingly, even GS1 produces sub-circuits of a reasonable volume despite of the usage of the cut-based method with a simple root node selection strategy. On the average, the size of the windows is much smaller than the chosen limit c_{max}. We can also observe that many windows consisting of less than 10 gates were generated. This is valid for 'ac97_ctrl', 'sasc', 'ss_pcm' and 'usb_phy'. Much larger windows are generated for the arithmetic circuits than for the controllers and logic, on average. On the other hand, we can also see that the cut-based method is able to extract sub-circuits having significantly more than 100 gates but we never hit c_{max}. The number of inputs and outputs positively correlates with the size of W. The larger the number of gates in the window, the higher number of inputs and outputs. This observation is valid for both methods.

The number of inputs of the sub-circuits optimized by the evolution is substantially higher compared to the numbers used by the rewriting algorithm which is applied in the conventional synthesis. Compared to the rewriting and other techniques mentioned in Section 2.2, a relatively complex portions of the original circuits are chosen for subsequent optimization. This could explain the reason, why the proposed EA-based method is able to achieve such reduction compared to the conventional state-of-the-art synthesis.

6 Conclusion

Compared to the conventional logic synthesis, state-of-the-art EA-based optimization is able to produce substantially better results but at the cost of a higher run time. Unfortunately, the run time increases with the increasing complexity of the Boolean networks. This work addressed this problem by combining the EA-based optimization with the principle of the so called Boolean network scoping. Our method extracts smaller sub-circuits from a complex circuit that are optimized locally and implanted back to the original circuit. This concept can be understood as the evolutionary resynthesis. This approach helps to improve the scalability because the evolution is applied on smaller portions of the original Boolean network.

We implemented and evaluated two different techniques to the sub-circuit extraction. One method is based on the computation of the so called reconvergence-driven cuts. This approach is used in the state-of-the-art logic synthesis algorithms but in a different scenario. Despite of many advantageous properties, the cut-based method has some limitations regarding our application. To avoid that, we proposed an alternative approach loosely inspired by a conventional windowing technique.

Even though we used a simple setting which may degrade the capabilities of the resynthesis (e.g. the fixed number of evaluations of EA or random root node selection), the proposed approach was able to outperform the EA-based optimization applied to the whole Boolean networks. The proposed sub-circuit extraction inspired by windowing exhibit significantly better compared to the cut-based alternative. On the average, the evolutionary resynthesis achieved 8.9% circuit size improvement on controllers and 21.4% improvement on arithmetic circuits. The globally applied evolution was able to improve the circuits belonging to the mentioned groups by 7.5% and 1.8%, respectively. Even though only the area was targeted in this study, the depth of the optimized circuits is comparable with the original circuits.

The capability of exploration of the evolutionary resynthesis is higher but at the cost of slower convergence. There are few instances where the EA-based optimization applied to the whole circuit produced better results. In our future work, we would like to implement an adaptive strategy that modifies the maximum number of evaluations according to the size of the optimized logic circuit. We suppose that this mechanism helps us to improve the convergence. In addition to that, we would like to focus on improvement of root node selection strategy. The question here is whether the result would be better if the cut is built from a node near to the previously chosen one.

References

1. Brayton, R., Mishchenko, A.: ABC: An academic industrial-strength verification tool. In: Computer Aided Verification, pp. 24–40. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
2. Coello, C.C.A., Christiansen, A.D., Aguirre, A.H.: Automated design of combinational logic circuits by genetic algorithms. In: Artificial Neural Nets and Genetic Algorithms: Proceedings of the International Conference in Norwich, U.K., 1997, pp. 333–336. Springer Vienna, Vienna (1998). DOI 10.1007/978-3-7091-6492-1_73
3. Fiser, P., Schmidt, J.: Small but nasty logic synthesis examples. In: Proc. 8th Int. Workshop on Boolean Problems, pp. 183–190 (2008)
4. Fiser, P., Schmidt, J., Vasicek, Z., Sekanina, L.: On logic synthesis of conventionally hard to synthesize circuits using genetic programming. In: 13th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems, pp. 346–351 (2010)
5. Goldman, B., Punch, W.: Reducing wasted evaluations in cartesian genetic programming. Lecture Notes in Computer Science **7831 LNCS**, 61–72 (2013). DOI 10.1007/978-3-642-37207-0_6
6. Goldman, B.W., Punch, W.F.: Analysis of cartesian genetic programming's evolutionary mechanisms. IEEE Transactions on Evolutionary Computation **19**(3), 359–373 (2015)
7. Gordon, T.G.W., Bentley, P.J.: On evolvable hardware. In: Soft Computing in Industrial Electronics, pp. 279–323. Physica-Verlag, London, UK (2002)
8. Haddow, P.C., Tyrrell, A.: Challenges of evolvable hardware: past, present and the path to a promising future. Genetic Programming and Evolvable Machines **12**, 183–215 (2011)
9. Higuchi, T., Niwa, T., Tanaka, T., Iba, H., de Garis, H., Furuya, T.: Evolving Hardware with Genetic Learning: A First Step Towards Building a Darwin Machine. In: Proc. of the 2nd International Conference on Simulated Adaptive Behaviour, pp. 417–424. MIT Press (1993)
10. Kocnova, J., Vasicek, Z.: Towards a scalable ea-based optimization of digital circuits. In: Genetic Programming 22nd European Conference, EuroGP 2019, pp. 81–97. Springer International Publishing (2019). DOI 10.1007/978-3-030-16670-0_6
11. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge, MA (1992)
12. Li, N., Dubrova, E.: AIG rewriting using 5-input cuts. In: Proc. of the 29th Int. Conf. on Computer Design, pp. 429–430. IEEE CS (2011)
13. Miller, J., Thomson, P.: Cartesian Genetic Programming. In: Proc. of the 3rd European Conference on Genetic Programming EuroGP2000, *LNCS*, vol. 1802, pp. 121–132. Springer (2000)
14. Miller, J.F.: Digital filter design at gate-level using evolutionary algorithms. In: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 1999, pp. 1127–1134. Morgan Kaufmann (1999)
15. Miller, J.F.: Cartesian Genetic Programming. Springer-Verlag (2011)

16. Miller, J.F.: Cartesian genetic programming: its status and future. Genetic Programming and Evolvable Machines (2019). DOI 10.1007/s10710-019-09360-6
17. Miller, J.F., Thomson, P., Fogarty, T.: Designing electronic circuits using evolutionary algorithms. arithmetic circuits: A case study. In: Genetic Algorithms and Evolution Strategies in Engineering and Computer Science, pp. 105–131. Wiley (1997)
18. Mishchenko, A., Brayton, R.: Scalable logic synthesis using a simple circuit structure. In: Int. Workshop on Logic and Synthesis, pp. 15–22 (2006)
19. Mishchenko, A., Chatterjee, S., Brayton, R.: DAG-aware AIG rewriting: a fresh look at combinational logic synthesis. In: 2006 43rd ACM/IEEE Design Automation Conference, pp. 532–535 (2006). DOI 10.1145/1146909.1147048
20. Sekanina, L.: Evolvable Components: From Theory to Hardware Implementations. Natural Computing Series, Springer Verlag (2004)
21. Sekanina, L., Ptak, O., Vasicek, Z.: Cartesian genetic programming as local optimizer of logic networks. In: 2014 IEEE Congress on Evolutionary Computation, pp. 2901–2908. IEEE CIS (2014)
22. Shanthi, A.P., Parthasarathi, R.: Practical and scalable evolution of digital circuits. Applied Soft Computing **9**(2), 618–624 (2009)
23. Stomeo, E., Kalganova, T., Lambert, C.: Generalized disjunction decomposition for evolvable hardware. IEEE Transaction Systems, Man and Cybernetics, Part B **36**(5), 1024–1043 (2006)
24. Stomeo, E., Kalganova, T., Lambert, C.: Generalized disjunction decomposition for the evolution of programmable logic array structures. In: First NASA/ESA Conference on Adaptive Hardware and Systems (AHS'06), pp. 179–185 (2006)
25. Tao, Y., Zhang, L., Zhang, Y.: A projection-based decomposition for the scalability of evolvable hardware. Soft Computing **20**(6), 2205–2218 (2016). DOI 10.1007/s00500-015-1636-2
26. Thompson, A.: Silicon evolution. In: Proceedings of the First Annual Conference on Genetic Programming, GECCO '96, pp. 444–452. MIT Press, Cambridge, MA, USA (1996)
27. Vasicek, Z.: Cartesian GP in optimization of combinational circuits with hundreds of inputs and thousands of gates. In: Proceedings of the 18th European Conference on Genetic Programming – EuroGP, LCNS 9025, pp. 139–150. Springer International Publishing (2015)
28. Vasicek, Z., Sekanina, L.: Formal verification of candidate solutions for post-synthesis evolutionary optimization in evolvable hardware. Genetic Programming and Evolvable Machines **12**(3), 305–327 (2011)
29. Vassilev, V., Job, D., Miller, J.F.: Towards the Automatic Design of More Efficient Digital Circuits. In: J. Lohn, A. Stoica, D. Keymeulen, S. Colombano (eds.) Proc. of the 2nd NASA/DoD Workshop on Evolvable Hardware, pp. 151–160. IEEE Computer Society, Los Alamitos, CA, USA (2000)
30. Wolf, C., Glaser, J., Kepler, J.: Yosys-a free verilog synthesis suite. In: Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip) (2013)
31. Zhao, S., Jiao, L.: 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)

Appendix D

Resynthesis of logic circuits using machine learning and reconvergent paths

KOCNOVA Jitka and VASICEK Zdenek

In: *24th Euromicro Conference on Digital System Design (DSD)*. Palermo, Italy, 2021, pp. 69-76, doi: 10.1109/DSD53832.2021.00020.

Resynthesis of logic circuits using machine learning and reconvergent paths

Jitka Kocnova
Brno University of Technology,
Faculty of Information Technology,
IT4Innovations Centre of Excellence
Brno, Czech Republic
ikocnova@fit.vutbr.cz

Zdenek Vasicek
Brno University of Technology,
Faculty of Information Technology,
IT4Innovations Centre of Excellence
Brno, Czech Republic
vasicek@fit.vutbr.cz

***Abstract*—Boolean network scoping represents a common approach incorporated in conventional synthesis tools for maintaining good scalability of the synthesis process. Recently, an approach to the local resynthesis based on combination of evolutionary optimization with the principle of Boolean network scoping has been proposed. Local resynthesis is an iterative process based on the extraction of smaller sub-circuits from a complex circuit that are optimized locally and implanted back to the original circuit. The main advantage of the local resynthesis is that it can mitigate the problem of scalability of representation which is typical to the evolutionary algorithms as the efficiency of the evolutionary optimization applied at the global level deteriorates with the increasing circuit complexity. Unfortunately, the efficiency of local resynthesis depends on the efficiency of the sub-circuit extraction process. We propose an alternative method, based on the reconvergent paths. The evaluation is performed on a set of highly optimized benchmark problems representing various real-world controllers, logic and arithmetic circuits. The method provides better results compared to the state-of-the-art logic synthesis tool and evolutionary optimization techniques operating locally and globally. A substantially higher number of redundant gates was removed in more than 70% cases, while keeping the computational effort at the same level. A huge improvement was achieved especially for the controllers. On average, the proposed method was able to remove more than 14.3% of gates. The highest achieved gate reduction was more than 45% of gates.**

***Index Terms*—Logic optimization, Cartesian Genetic Programming, Evolutionary Resynthesis**

I. Introduction

Logic synthesis transforms a high-level description into a gate-level or transistor-level implementation. Due to the complexity of the current digital systems, the synthesis process is typically broken into a sequence of steps. Among others, logic optimization represents an important part of the whole process. The goal of the logic optimization is to transform a suboptimal solution into an optimal gate-level implementation with respect to given synthesis goals. The problem is typically represented using a suitable internal representation due to the scalability issues. Current state-of-the-art logic synthesis tools, such as ABC, represent circuits using a directed acyclic graph composed of two-input AND nodes connected by direct or negated edges denoted as and-inverter graph (AIG). The AIG representation is simple and scalable, and leads to simple algorithms but it suffers from an inherent bias in representation. While eight of ten possible two-input logic gates may be represented by a single AIG node, XOR and XNOR gates require three AIG nodes each. The efficiency of synthesis is then limited as it mostly fully relies on transformations that disallow an increase in the number of AIG nodes. However, the ability to capture XOR gates is essential for efficient representation of arithmetic and XOR-intensive circuits. It has been shown that there exists a huge class of real-world circuits for which the synthesis fails and provides very poor results [1, 2, 3]. In some cases, the area of the synthesized circuits is orders of magnitude higher than the known optimum. If a large design is broken down to multiple smaller circuits and such a failure occurs during resynthesis, we obtain an unacceptably large circuit. To address this problem, various approaches have been proposed. E.g. binary decision diagrams (BDDs) can be employed [4, 5]. Due to their limited scalability, Amaru et al. employed a two step synthesis process based on a selective and distinct manipulation of AND/OR and XOR-intensive portions of the logic circuit [6]. In the first phase, XOR-intensive regions are identified in the input Boolean network. These regions are then optimized in the second phase independently of the rest of the network. On average, the method outperforms the AIG-based ABC by 18% when the number of transistors is considered. Fiser et al. introduced XOR-AIGs to explicitly support XOR gates [7]. The synthesis is based on a modified rewriting selecting subgraphs with four leaves. Unfortunately, no significant improvement has been reported in the paper. Haaswijk et al. adopted a different approach [8]. Instead of AIG, XOR majority graphs (XMGs) have been employed to extend the capabilities of the synthesis oriented on area optimization. To summary, the conventional methods rely on circuit preprocessing or circuit decomposition [6], precomputation of optimal solutions [7] or presence of advanced technology cells such as XMG [8].

A different strategy is to avoid intermediate representation. Various machine-learning approaches working directly at the level of gates were successfully applied to address this problem [1, 9]. In [9], for example, Vasicek demonstrated that the evolutionary synthesis using Cartesian Genetic Programming

(CGP) conducted directly at the level of common gates is able to provide significantly better results compared to the state-of-the-art synthesis operating on AIGs. On average, the method enabled a 34% reduction in gate count on an extensive set of benchmark circuits when executed for 15 minutes. A similar approach was successfully applied even to the synthesis of conventionally hard to synthesize circuits [2]. It was observed, however, that the efficiency of the evolutionary approach deteriorates with the increasing circuit complexity, i.e. the increasing number of gates. Motivated by this fact, a combination of evolutionary optimization with the principle of so-called Boolean network scoping has been proposed in [10, 11]. Boolean network scoping represents a common approach incorporated in conventional synthesis tools for maintaining the good scalability of the synthesis process. The key idea is to use an iterative procedure which extracts sub-circuits that are subsequently optimized by Cartesian Genetic Programming and implanted back into the original circuit provided that there is an improvement at the global level. This approach can be understood as the EA-based resynthesis. As it has been demonstrated in [11], the size of the sub-circuits has an impact on the scalability of the CGP and also on the efficiency of the whole optimization process. Small sub-circuits ensure a good scalability of the evolutionary optimization, but they lead to minor improvements at the global level because this method operates mainly locally similarly to the conventional rewriting. Large sub-circuits, on the other hand, increase a chance for improvement but the performance of the CGP deteriorates with increasing the size of the optimized circuit. In order to obtain a reasonable optimization method, it is necessary to find a good trade-off between the mentioned two extremes. Two methods of Boolean network scoping were proposed in the literature. The first was inspired by the conventional method based on computing so-called k-feasible cuts [10]. The other one was based on a so-called windowing method [11]. Both methods achieved significant reduction w.r.t. the number of gates in the benchmark circuits but the sub-circuit selection introduces some overhead. This overhead is noticeable especially if many sub-circuits that lack any form of redundancy are selected. In this case, the evolutionary optimization will spend a significant amount of time by trying to search for a better solution that simply does not exist.

Our goal is to improve the performance of the sub-circuit selection – mainly to focus on particular areas in the circuits that may potentially boost the efficiency of the evolutionary optimization. In order to achieve this goal, we propose to optimize sub-circuits containing so-called reconvergent paths.

II. Background

This section presents relevant background on conventional as well as EA-based optimization of logic circuits and introduces the notation used in the rest of the paper.

A. Boolean networks

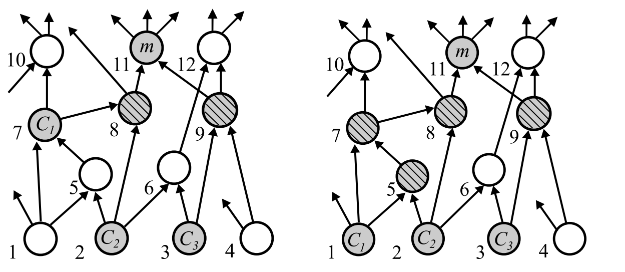
Every circuit can be represented using a Boolean network. A *Boolean network* is a directed acyclic graph (DAG) with nodes represented by Boolean functions [12]. The sources of the graph are the primary inputs (PIs) of the network and the sinks are the primary outputs (POs). The output of a node may be an input to other nodes called *fanouts*. The inputs of a node are called *fanins*. An edge connects two nodes that are in fanin/fanout relationship. Considering this notion, And-Inverter Graph is a Boolean network composed of two-input ANDs and inverters. The network primary inputs are signals that are driven by the environment, there is no node driving these signals in the network. Similarly, the primary outputs are signals that drive the environment and are needed by inner network nodes as well. The size of the network is the number of the nodes (primary inputs and outputs are not considered).

B. Limiting the scope of Boolean networks

Network scoping represents a key operation to ensure a good scalability of synthesis tools when working with large Boolean networks. It forms an integral part of rewriting as well as refactoring. Two approaches have been proposed to limit the scope of logic synthesis to work only on a small portion of a Boolean network – *windowing* and *cut computation* [11, 12].

The windowing algorithm determines the working area denoted as window by computing transitive fanin and transitive fanout. The algorithm takes a node (typically referred to as pivot node) and two integers *m* and *n* defining the number of logic levels on the fanin/fanout sides of the node to be included in the resulting window. The transitive fanin is a set of nodes on the fanin side that are distance-*m* or less from the pivot node. Similarly, the transitive fanout is a set of nodes on the fanout side that are distance-*n* or less from the pivot node. These two sets are then used to obtain the leaf and root sets that uniquely determine the window. The window of a Boolean network N is a connected subnetwork $N\prime \subseteq N$ that corresponds to the subset of nodes of the network containing nodes from root set together with all nodes on paths between the leaf set and the root set. The nodes in the leaf set are not included in the window. The complete algorithm can be found in [11, 12]. The main problem of this algorithm is that it is hard to predict how many logic levels have to be traversed to get a window of the desired parameters.

The second approach based on computing so-called k-feasible cuts is usually preferred to avoid determining the required number of logic levels. A cut of a node, called root node, is a set of nodes of the network, called leaves, such that each path from PI to the root node passes through at least one leaf. A cut is k-feasible if the number of nodes (i.e. cut size) in the cut does not exceed k. The volume of a cut is the total number of nodes encountered on all paths between the root node and the cut leaves. An example of two different 3-feasible cuts is shown in Fig. 1. The problem is that the cut computed using a naive breadth-first-search algorithm may include only a few nodes and leads to tree-like logic structures. Such a structure does not lead to any don't cares in the local scope of the node and attempting optimization using such a cut would be wasted time. A simple and efficient cut computation algorithm producing a cut close to a given



(a) Cut $\mathrm{C}^{\mathrm{I}} = \{7, 2, 3\}$ (b) Cut $\mathrm{C}^{\mathrm{II}} = \{1, 2, 3\}$

Fig. 1: Example of two possible 3-feasible cuts for root node m and given Boolean network. The cut C^{II} is preferred as its volume is five (root node m and contained nodes 5, 7, 8 and 9). There are only two contained nodes (node 8 and 9) in the case of C^{I}.

size while heuristically maximizing the cut volume has been introduced in [12]. The k-feasible cuts are important not only for the gate-level logic synthesis but also for FPGA-based synthesis as a k-feasible cut can be implemented as a k-input LUT.

C. Evolutionary Synthesis of Logic Circuits

Evolutionary algorithms (EAs) have been used to synthesize logic circuits since the late nineties [13, 14]. Miller et al., the author of Cartesian Genetic Programming (CGP) [15], is considered a pioneer in the field of logic synthesis of gate-level circuits. He utilized his own variant of genetic programming to synthesize compact implementations of multipliers described by means of a behavioral specification [16]. Despite the many advantages of this unconventional technique, only small problem instances were typically addressed. To tackle the limited scalability, various decomposition strategies have been proposed. A good survey of the existing techniques is provided, for example, in [17]. In 2011, the scalability of CGP has been significantly improved by introducing a SAT-based CGP. It exploits the fact that the candidate solutions must be functionally equivalent with their parent in logic optimization in order to be further accepted. It also exploits the knowledge of differences between parental and candidate circuits. The efficiency of the SAT-based method was further improved by combining a SAT solver with an adaptive high-performance circuit simulator used to quickly identify the potential functional non-equivalence. The most advanced SAT-based CGP employs a simulator that is driven by counterexamples produced by the SAT solver [9]. Neither the original nor the latter approach rely on a decomposition of the optimized circuits. Many other machine learning techniques have been recently used for circuit synthesis, see e.g. [18].

Since its introduction, CGP remains the most powerful evolutionary technique in the domain of logic used in the domain of digital circuit synthesis and optimization [14]. In this area, a linear form of CGP is preferred today. CGP models a candidate circuit having n_i PIs and n_o POs as a linear 1D array of n_n configurable nodes, as can be seen in the Figure 2. Each node has n_a inputs and corresponds to a single gate with up to n_a inputs. The inputs can be connected either to the output of a node placed in the previous L columns or directly to PIs to avoid feedback. The function of a node can be chosen from a set of n_f functions. Depending on the function of a node, some of its inputs may become redundant. Also, the fixed number of nodes n_n does not mean that all the nodes contribute to the POs. These key features allow redundancy and flexibility of CGP.

The candidate circuits are encoded as follows. Each PI as well as each node has associated an unique index. Each node is encoded using $n_a + 1$ integers $(x_1, \cdots, x_{n_a}, f)$ where the first n_a integers denote the indices of its fanins and the last integer determines the function of that node. Every candidate circuit is encoded using $n_n(n_a+1)+n_o$ integers. The last n_o integers specify the indices corresponding to each PO.

CGP is a population oriented approach which operates with $1 + \lambda$ candidate solutions. The initial population is seeded by the original circuit ought to be optimized. Every new population consists of the best circuit chosen from the previous population and its λ offspring created using a mutation operator that randomly modifies up to h integers. Considering the CGP encoding, a single mutation causes either reconnection of a gate, reconnection of primary outputs or change in function of a gate. The selection of the individuals is typically based on a cost function (e.g. the number of active nodes). In the case that there are more individuals with the same score, the individual that has not served as a parent will be selected as the new parent. This procedure is typically repeated for a predefined number of iterations.

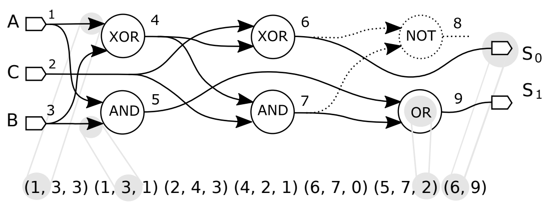


Fig. 2: Example of a CGP individual encoding a logic circuit (one-bit full adder) with $n_i = 3$ inputs and $n_o = 2$ outputs. The individual is encoded using an array of $n_n = 6$ two-input single-output nodes whose functions are chosen from a set of primitive functions $\Gamma = \{\mathrm{NOT}, \mathrm{AND}, \mathrm{OR}, \mathrm{XOR}\}$. The nodes are arranged in a two-dimensional grid for improved readability. Redundant connections and nodes, (those that do not contribute to the outputs) are highlighted by dotted line.

D. EA-based resynthesis

Let $\mathcal{C}$ be a combinational circuit described at the level of common gates represented by a Boolean network N consisting of $|N|$ nodes. Each node corresponds to a single gate in $\mathcal{C}$. The pseudo-code of the proposed optimization procedure based on evolutionary resynthesis presented in [11] is shown in Algorithm 1.

An iterative process which consists of a sequence of three steps that are executed in a loop is applied. A working area

(Boolean network W) is extracted from the Boolean network N' in the first step. The goal is to obtain a smaller circuit which is easier to manipulate with. Each W that is not suitable for the subsequent optimization is skipped in the next step in order to eliminate execution of a relatively time-consuming resynthesis for the windows that are unlikely to lead to any improvement. Identification of the suitable windows can be based on the size of W (small windows are filtered out) or a more advanced metric which reflect, for example, the number of inputs and depth (thin windows are filtered out). In the third step, resynthesis by means of the CGP is applied to the extracted Boolean network. At the beginning, each node in the window is assigned an unique index and a netlist corresponding with the nodes in the window is created. This netlist is then used to seed the initial population. The evolutionary optimization is executed for a limited number of iterations. The number of iterations should be determined heuristically. The more iterations are allowed, the higher improvement can be achieved. On the other hand, many iterations on a small window wastes time. Finally, the optimized logic network W' is evaluated w.r.t. N' and if it performs better, it replaces all non-leaf nodes included in W. The whole optimization algorithm is terminated when a predefined number of iterations or a given runtime is exhausted.

Algorithm 1: Optimization of digital circuits using EA-based resynthesis

```
Input: A Boolean network $N$
Output: Optimized network $N'$, $cost(N') \leq cost(N)$
1 $N' \leftarrow N$
2 while terminated condition not satisfied do
3     $W \leftarrow$ GetSubcircuit()
4     if $W$ is a suitable candidate then
5         $W' \leftarrow$ OptimizeNetworkUsingEA($W$)
6         if cost$((N' \setminus W) \cup W') <$ cost$(N')$ then
7             $N' \leftarrow (N' \setminus W) \cup W'$
8 return $N'$
```

In [11], two different approaches to the extraction of W were proposed. The first one was based on the cut computed using a naive breadth-first-search algorithm. The problem here was that W extracted a tree-like logic structure and consisted of only a few nodes, mainly when working with networks of a small depth. In order to maximize the volume of W, another approach based on the windowing algorithm was presented. Instead of predicting how many logic levels were needed to traverse in order to get the W of a desired volume, the W was cumulatively expanded with all of the neighbouring nodes of the nodes already present in the W, so that W was expanded in every possible direction. This approach successfully overcame the issues connected with the earlier mentioned breadth-first-search-based algorithm. The W is always extracted randomly in these two methods, without incorporating any further knowledge about the circuit (e.g. number of nodes, PIs, POs, depth, atc.). The optimization step itself may then lead to an inefficiency when trying to optimize a W, that can not be optimized any more or, when the W does not include sufficient interconnection between its nodes. As shown in [11], the main progress in optimization of a circuit was achieved mostly in the beginning of the optimization; after that, the node reduction seemed to be not very significant even though the circuits still contained redundant nodes. Also, only a part of the CGP generations computed for each sub-circuit had an effect on reduction or at least modification of the sub-circuit. In conclusion, many iterations of the optimization were executed without actually making an impact on the overall result.

III. THE PROPOSED METHOD

To overcome limitations mentioned in Section II-D, we propose to increase the redundancy of the nodes in W. Before selecting the W itself, we try to find a so-called reconvergence path. Reconvergent paths lead from one source node through two different areas of the network and meet again at the inputs of a single node located in fanout cone of the source node. Such a path may increase a chance of a good optimization result, as it may contain more redundant nodes than the other areas of the network N [12].

Algorithm 2: Procedure GetSubcircuit based on reconvergence paths

```
Input: A Boolean network $N$,
minimum ($rw_{min}$) and maximum ($rw_{max}$) volume of $W$,
maximum $rp_{max}$ volume of reconvegent path $rp$
Output: A working area $RW$, $rw_{min} \leq |W| \leq rw_{max}$
1 $RW \leftarrow \emptyset$
2 $rp \leftarrow \emptyset$
3 $init\_roots \leftarrow$ randomly select $K$ nodes from $N$
4 $rp \leftarrow$ identify a reconvergence path starting from a node
    $n \in init\_roots$ containing at least $rp_{max}$ nodes
5 if $rp$ found then
6     push all nodes from $rp$ to $RW$
7 else
8     select a node $n \in init\_roots$ randomly
9     push $n$ to $RW$
10 init queue $q$ with $rp$
11 while $q$ not empty $\wedge$ $|RW| < rw_{max}$ do
12     $rm \leftarrow$ pop a node from $q$
13     $RW \leftarrow RW \cup \{rm\}$
14     $X \leftarrow fanin(rm) \cup fanout(rm)$
15     push all nodes from $X \setminus RW$ that are not already in $q$
         into $q$
16 if $|RW| < rw_{min}$ then
17     $RW \leftarrow \emptyset$
18 $W \leftarrow \bigcup_{rm \in RW} fanin(rm)$
19 return $RW$
```

The principle of the proposed selection strategy is given in Algorithm 2. The input is a boolean network N and the output is a set of nodes (selected sub-circuit) denoted as RW. At the beginning, an attempt to identify a reconvergent path rp of a desired volume (in terms of the number of gates along the path) is executed. This step is repeated at most K times for $k = 10$ randomly selected nodes denoted as $init\_roots$. The first successfully identified rp proceeds to the next step of the algorithm. All the nodes from rp are copied to RW (see line 6) and root of rp becomes the rm node. If the reconvergent path is not found in the given amount of tries, the RW is initiated with a randomly chosen root node from the $init\_roots$.

If the volume of the RW is not already at its upper limit rw_{max}, the RW is expanded with the nodes connected to the nodes already present in the RW. The expansion starts from the root node rm. If the RW contains less than rw_{min} nodes at the end of the selection, it is declined from the optimization process and search for a more suitable RW starts again. An example of the outcome of our algorithm can be seen in the Figure 3.

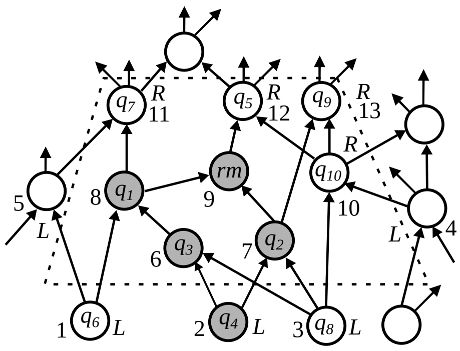


Fig. 3: Example of the window consisting of 13 nodes created using the reconvergence path selection Algorithm 2. The reconvergence path rp starting in the root node rm is highlighted using the gray nodes. The nodes q_5–q_{12} are those added during the final expansion of the selection RW. The nodes at the bottom are primary inputs. The root and leaves of the window are denoted as R and L, respectively. The nodes in the window have assigned an index (the number located below a particular node) used to uniquely identify each node in the CGP. The labels q_i inside the nodes denote the order i in which the nodes were chosen.

IV. Experimental evaluation

A. Experimental setup

The proposed method was implemented in C++ as a part of Yosys open synthesis suite [19]. This tool allows us to directly manipulate with Verilog files and it integrates ABC [20], a state-of-the-art academic tool for hardware synthesis and verification. The reconvergence path selection was done with the help of the Mockturtle C++ logic network library [21]. The goal of this paper is to evaluate performance of the proposed method (further denoted as GSRW) and compare the results to those presented in [11], i.e. to the method using the windowing algorithm for the sub-circuit extraction (denoted as GSW), and also the EA-based method (denoted as global) applied to the whole Boolean network. In addition to that, the results from ABC are included to establish a baseline. Each of the three evolutionary methods operate at the level of optimized and mapped Boolean networks to avoid the bias of AIG representation. To provide a fair evaluation, we used the same set of benchmark circuits as in [11]. This set includes 28 highly optimized real-world circuits to evaluate all of the methods. Nineteen Verilog netlists are taken from IWLS'05 Open Cores benchmarks, the remaining nine netlists represent various arithmetic circuits[1]. The circuits were optimized by ABC (several iterations of ABC command 'resyn') and mapped to gates using a library of common 2-input gates including XORs/XNORs gates (ABC command 'map'). After mapping, optimization by the three observed methods was executed and the final number of mapped gates in circuits was examined. All of the optimized circuits were formally verified w.r.t their original form (ABC command 'cec').

We target area-optimization. It means that the only criterion considered in the fitness function is the area on a chip expressed as the number of gates. The improvement is measured in terms of the number of removed gates. The line 6 of Algorithm 1 thus reduces to $|W'| < |W|$ which is much simpler to evaluate. For each method and each benchmark, five independent runs were executed to obtain statistically valid results. All of the optimized circuits were formally verified with respect to their original form (ABC command 'cec') to avoid any error in the evaluation.

The procedure OptimizeNetworkUsingEA is based on the CGP implemented as described in Section II-C with the following parameters: $n_a = 2$, $\lambda = 1$, $h = 2$, $n_n = |W|$, $\Gamma = \{\mathrm{BUF}, \mathrm{NOT}, \mathrm{AND}, \mathrm{OR}, \mathrm{XOR}, \mathrm{NAND}, \mathrm{NOR}, \mathrm{XNOR}\}$. The CGP parameters were chosen in accordance with [9]. A single call of this procedure is executed for the global method (the procedure takes the whole Boolean network and returns its optimized version). On the contrary, several calls of this procedure are executed both in the proposed GSRW method and the GSW method. The global method terminates when n_{iters} iterations are exhausted. One iteration corresponds to evaluation of a single candidate solution. In the case of the proposed method a simple divide-and-conquer strategy is employed. The termination conditions are designed as follows. The GSRW and GRW methods are allowed to execute n_{iters} iterations. Each iteration corresponds to a single execution of the OptimizeNetworkUsingEA procedure. This procedure terminates either when a given number of evaluations (n_{evals}) is exhausted or when a predefined amount of time (t_{max}) has elapsed. The latter condition helps to ensure a good scalability and predictability of the worst-case CPU time of the optimization which could be enormous especially in those cases when many hard-to-solve candidate solutions are generated during the evolution. The GSRW method is allowed to select a reconvergent path of a rp_{max} volume. In [11], the global method was terminated either when $n_{evals} \times n_{iters}$ evaluations were exhausted or when the CPU time reached $t_{max} \times n_{iters}$ seconds. To set up all the necessary parameters of the optimization, we used the same experimental settings as were used in [11]. This helps to fairly evaluate all evolutionary methods because they are allowed to evaluate the same number of candidate solutions. To match the setup with that used

[1]The benchmarks can be found at https://lsi.epfl.ch/MIG

in the already available works, we chose $n_{iters} = 2 \times 10^4$, $n_{evals} = 5 \times 10^5$, and $t_{max} = 10$ seconds in this work. The volume of the reconvergent path in the GSRW method is set to $rp_{max} = (10, 20, 50, 100)$ gates and the minimal and maximal volume of the selection $rw_{min} = 5$, $rw_{max} = 100$ nodes respectively. This setup ensures that 10^{10} candidate solutions are generated and evaluated for every method.

B. Experimental results

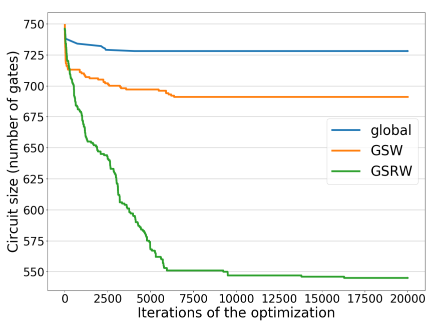
The overall results showing the performance of the considered approaches are summarized in Tab. I. The first three columns contain information about the benchmarks (name, number of PIs and POs). The next two columns show parameters of the optimized and mapped circuits produced by ABC; the number of gates and logic depth are given. These numbers serve as a baseline for our comparison. Then, the achieved improvement expressed as the relative reduction with respect to the baseline is reported for the global and both local methods. For each method, we report the average improvement and also the best obtained results (section best improvement). The statistics are based on all five independent runs for every circuit and every method. The results presented for the proposed GSRW method are in the form of an average improvement obtained from the average improvements of the five independent runs for every desired reconvergent path volume as mentioned in IV-A. The average results for the different reconvergent path volumes are quite similar for each circuit (the average difference between them is 2% at maximum). The similar reduction result is caused by the total volume of the selected sub-circuits ($rw_{max} = 100$ nodes). However, the presence of the reconvergent paths in the sub-circuits had the main impact on the gate reduction process. Hence, we decided to present an overall average of all of the obtained results for every circuit for the GSRW.

The proposed method was able to reduce the size of every circuit even though the circuits have already been optimized by ABC. On average, the GSRW method achieved 13.4% circuit size reduction on the IWLS'05 benchmarks and 14.5% reduction on arithmetic circuits. The highest improvement, 45.9%, was recorded for the 'hamming' benchmark. Although the GSRW method is in principle stochastic (similarly to the GSW), the best results obtained by it are relatively close to the average ones which suggests that this evolutionary method is quite stable. Compared to the global method, the GSRW method performs substantially better considering the average as well as the best results. It won in 26 out of 28 cases. The GSW method won in 24 cases against the global method. So, the global method comes out as a loser in this comparison, except for the two cases ('mem_ctrl' and 'spi'). It can be concluded, in general, that the global method works well especially for small instances that are compact (do not contain many independent sub-circuits) and that have a reasonable depth (10 to 25 levels). When compared to the GSW method, the GSRW is a winner in reducing the IWLS'05 Open Cores benchmarks – it won in 18 out of 19 cases. However, the GSW method achieved better results in the reduction of the arithmetic set of benchmarks. It was better in 5 out of 9 cases.

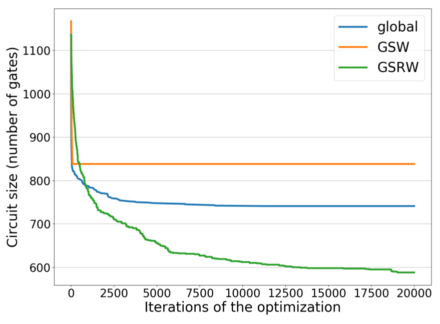
We also investigated and compared the corresponding convergence curves of the performance of the evolutionary methods. Global method converges quickly but the reduction process typically gets stuck at a local optima. Both the GSW and the proposed GSRW method profit from the usage of smaller sub-circuits, that require less computational effort to be optimized compared to the whole circuits. As can be clearly seen in the convergence graphs in Figure 4, the GSW method reaches its solution earlier than the GSRW method. However, the GSRW compensates for the slower convergence with better utilization of the computation time thanks to the suitable structure of the sub-circuits. Hence, a substantially higher number of the total 10^{10} candidate solutions generated during the optimization successfully participated in the final circuit reduction.

Considering the arithmetic circuits, the GSRW performs worse compared to the GSW method. In five cases ('hamming', 'diffeq1', 'div16', 'MAC32', 'revx'), the performance of the GSRW was surpassed by the GSW. We analysed the convergence curves and parameters of the produced windows to investigate this issue and we identified that the performance is connected with the internal structure of the circuits. Figure 5 shows the numeral IDs of root nodes of the sub-circuit identified using the GSRW method. The higher ID typically implies a node located close to the outputs and a higher volume of the corresponding sub-circuit. The blue boxplots represent cases for which the desired reconvergent path was found while the orange ones are those for which the reconvergent path does not exist and the same window-like selection as in the GSW was applied. It can be seen that for the 'hamming' benchmark (see Figure 5c), the reconvergent path was selected in almost every iteration of the optimization algorithm, which is good. However, we can see that the root nodes (and thus the reconvergent paths) were always selected from a limited set of nodes, as the blue-boxed IDs are concentrated around quite a narrow area across all of the computed iterations. This brings us to a conclusion that in this case (and in the four others as well, as the scattering of the root node locations was pretty similar for them) the selection algorithm was stuck to a limited set of the reconvergent paths present in the particular circuits. Therefore, the optimization was performed on a relatively small part of the boolean networks for the whole time and the remaining parts of the circuits were left unnoticed, which caused worse reduction in comparison with the GSW method, which selects the sub-circuits randomly. Despite not reaching the best results in those five cases because of this issue, the GSRW method was still able to reduce those five circuits by a decent amount of gates.

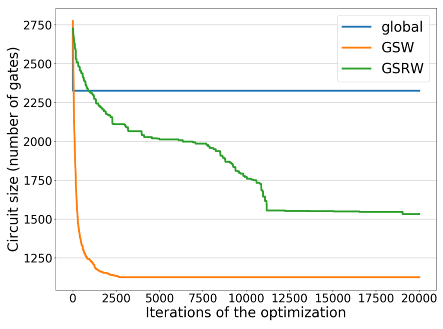
This statement could be supported with the scattering of the root node locations for the cases where the GSRW method outperformed the other methods. As an example, we present the 'pci_spoci' and 'sasc' benchmarks. It can be seen that the root nodes (majority of them rooting a reconvergent path) were selected from a much wider area compared to the total number



(a) Sasc

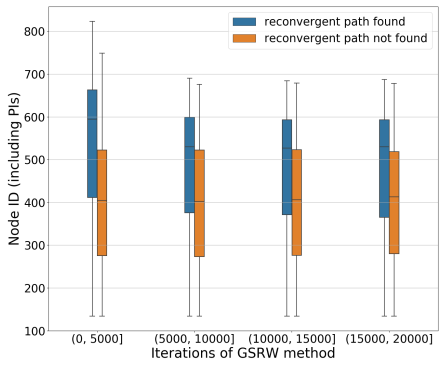


(b) Pci_spoci

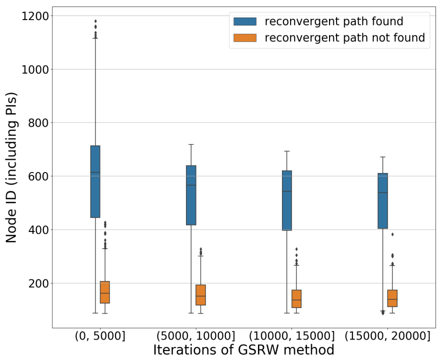


(c) Hamming

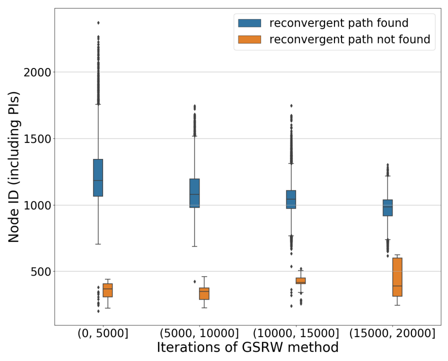
Fig. 4: Convergence through all of the generations



(a) sasc



(b) pci_spoci



(c) Hamming

Fig. 5: Root location through the iterations.

of nodes in the circuit. That caused the good performance of the GSRW method not only in this case, but also in all of the other winning cases reported in the Table I. Presence of the blue-marked root nodes in Figures 5a, 5b shows, that the GSRW was not always able to select a window containing a reconvergent path, but it was able to modify the circuit in a way that new suitable reconvergent paths appeared and were further selected for optimization. So, being highly successful in the reconvergent path selection does not necessarily imply the best final circuit reduction, mainly when there is a small number of reconvergent paths available.

V. Conclusion

State-of-the-art EA-based optimization is able to produce substantially better results at the cost of a higher run time compared to the conventional logic synthesis. However, the run time increases with the increasing complexity of the Boolean networks. Previous works addressed this problem by combining the EA-based optimization with the principle of the so-called Boolean network scoping. However, the methods used for sub-circuit selection were causing an inefficiency resulting in a waste of computational time, when trying to reduce sub-curcuits, that could not be optimized any further. Our work addressed this problem by selecting sub-circuits that contain so-called reconvergent paths. This allowed us to focus the computational effort on the parts of the original circuits that have the high chance of reduction in means of number of gates. The proposed method outperformed the earlier presented works focused on EA-based optimization combined with sub-circuit selection in 22 out of 28 cases. When compared to the globally working EA-based optimization, the proposed method won in 26 out of 28 cases. The overall average reduction was 13.4% for the IWLS'05 benchmarks and 14.5% for the arithmetic benchmarks. In our future work, we would like to further improve the sub-circuit selection so that it does not stick to a limited set of reconvergent paths as it had in some of our experiments.

Acknowledgment

This work was supported by Czech Science Foundation project GA19-10137S.

References

[1] L. Sekanina, O. Ptak, and Z. Vasicek, "Cartesian genetic programming as local optimizer of logic networks," in *2014 IEEE Congress on Evolutionary Computation*. IEEE CIS, 2014, pp. 2901–2908.

[2] P. Fiser, J. Schmidt, Z. Vasicek, and L. Sekanina, "On logic synthesis of conventionally hard to synthesize circuits using genetic programming," in *13th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems*, 2010, pp. 346–351.

[3] P. Fiser and J. Schmidt, "Small but nasty logic synthesis examples," in *Proc. 8th Int. Workshop on Boolean Problems*, 2008, pp. 183–190.

[4] C. Yang and M. Ciesielski, "BDS: a BDD-based logic optimization system," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 21, no. 7, pp. 866–876, Jul 2002.

TABLE I: Comparison of the proposed method against ABC, the method using the windowing algorithm and the CGP working globaly. The columns 'GSRW', 'GSW' and 'global' report the relative improvement in the number of gates compared to the optimized circuits obtained using ABC. Column 'ABC' contains parameters of the optimized circuits after mapping ('gates' is the number of gates, 'depth' is logic depth).

			ABC		GSRW		GSW [11]		global [9]	
Benchmark	PIs	POs	gates	depth	avg	best	average	best	average	best
DSP	4223	3792	43491	45	**4.9%**	**5.3%**	1.4%	2.13%	0.0%	0.0%
ac97_ctrl	2255	2136	11433	10	**4.6%**	**5.6%**	3.0%	4.0%	1.4%	1.4%
aes_core	789	532	21128	20	**8.5%**	**9.8%**	4.2%	5.5%	0.6%	1.7%
des_area	368	70	5199	25	**6.4%**	**7.4%**	4.5%	5.2%	2.1%	2.3%
des_perf	9042	1654	78972	16	**7.3%**	**9.5%**	2.8%	4.2%	0.0%	0.1%
ethernet	10672	10452	60413	23	**1.9%**	**2.6%**	1.6%	1.7%	0.0%	0.0%
i2c	147	127	1161	12	**24.7%**	**25.7%**	18.3%	18.5%	10.0%	10.7%
mem_ctrl	1198	959	10459	24	9.6%	10.9%	6.2%	10.0%	**24.8%**	**25.4%**
pci_bridge32	3519	3136	19020	21	**6.7%**	**7.1%**	3.4%	4.7%	0.5%	0.6%
pci_spoci_ctrl	85	60	1136	15	**43.0%**	**46.9%**	31.5%	36.7%	34.8%	35.7%
sasc	133	123	746	8	**21.0%**	**24.9%**	7.4%	7.4%	2.4%	2.8%
simple_spi	148	132	822	11	**11.9%**	**14.0%**	6.6%	7.4%	4.4%	4.6%
spi	274	237	3825	26	9.3%	10.4%	5.0%	8.4%	**13.5%**	**20.2%**
ss_pcm	106	90	437	7	**12.4%**	**13.0%**	4.8%	5.5%	2.3%	2.3%
systemcaes	930	671	11352	27	**12.1%**	**19.8%**	11.7%	12.7%	0.0%	0.0%
systemcdes	314	126	2601	25	**19.5%**	**21.4%**	15.7%	15.9%	9.1%	9.9%
tv80	373	360	8738	39	10.5%	11.6%	**13.5%**	**14.2%**	11.1%	11.3%
usb_funct	1860	1692	15405	23	**10.4%**	**14.0%**	10.2%	11.3%	2.6%	2.6%
usb_phy	113	73	452	9	**29.1%**	**30.3%**	17.7%	18.0%	12.2%	12.2%
average (IWLS'05 benchmarks)			15620	20	**13.4%**	**15.3%**	6.4%	6.5%	7.0%	7.6%
mult32	64	64	8225	42	**21.6%**	**24.6%**	19.5%	20.9%	0.0%	0.0%
sqrt32	32	16	1462	307	**17.1%**	**26.2%**	6.6%	9.5%	3.0%	3.0%
diffeq1	354	193	20719	218	8.7%	11.4%	**25.5%**	**28.6%**	0.0%	0.0%
div16	32	32	5847	152	20.6%	25.1%	**29.5%**	**42.7%**	0.0%	0.0%
hamming	200	7	2724	80	45.9%	52.9%	**58.8%**	**58.9%**	14.6%	14.6%
MAC32	96	65	7793	55	5.5%	6.4%	**9.5%**	**10.5%**	0.0%	0.0%
revx	20	25	8131	171	7.9%	9.0%	**18.0%**	**21.2%**	0.0%	0.1%
mult64	128	128	21992	190	**12.6%**	**12.9%**	5.0%	6.2%	0.3%	0.5%
max	512	130	3719	117	**5.2%**	**5.6%**	5.1%	5.2%	0.7%	0.8%
average (arithmetic benchmarks)			8956	148	14.5	17.4	**19.7%**	**22.6%**	2.1%	2.1%

[5] N. Vemuri, P. Kalla, and R. Tessier, "BDD-based logic synthesis for LUT-based FPGAs," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 7, no. 4, pp. 501–525, Oct. 2002.

[6] L. Amaru, P. E. Gaillardon, and G. D. Micheli, "MIXSyn: An efficient logic synthesis methodology for mixed XOR-AND/OR dominated circuits," in *2013 18th ASP-DAC Conference*, 2013, pp. 133–138.

[7] P. Fiser, I. Halecek, and J. Schmidt, "SAT-based generation of optimum function implementations with XOR gates," in *2017 Euromicro Conference on Digital System Design (DSD)*, 2017, pp. 163–170.

[8] W. Haaswijk, M. Soeken, L. Amaru, P. E. Gaillardon, and G. D. Micheli, "A novel basis for logic rewriting," in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2017, pp. 151–156.

[9] Z. Vasicek, "Cartesian GP in optimization of combinational circuits with hundreds of inputs and thousands of gates," in *Proc. of the 18th European Conference on Genetic Programming – EuroGP*, ser. LCNS 9025. Springer, 2015, pp. 139–150.

[10] J. Kocnova and Z. Vasicek, "Towards a scalable ea-based optimization of digital circuits," in *Genetic Programming*. Cham: Springer International Publishing, 2019, pp. 81–97.

[11] ——, "Ea-based resynthesis: An efficient tool for optimization of digital circuits," *Genetic Programming and Evolvable Machines*, vol. 21, no. 3, pp. 287–319, 2020.

[12] A. Mishchenko and R. Brayton, "Scalable logic synthesis using a simple circuit structure," in *Int. Workshop on Logic and Synthesis*, 2006, pp. 15–22.

[13] J. D. Lohn and G. S. Hornby, "Evolvable hardware: Using evolutionary computation to design and optimize hardware systems," *IEEE Computational Intelligence Magazine*, vol. 1, no. 1, pp. 19–27, 2006.

[14] J. Miller and P. Thomson, "Cartesian Genetic Programming," in *Proc. of the 3rd European Conference on Genetic Programming EuroGP2000*, ser. LNCS, vol. 1802. Springer, 2000, pp. 121–132.

[15] J. F. Miller, *Cartesian Genetic Programming*. Springer-Verlag, 2011.

[16] V. Vassilev, D. Job, and J. F. Miller, "Towards the Automatic Design of More Efficient Digital Circuits," in *Proc. of the 2nd NASA/DoD Workshop on Evolvable Hardware*, J. Lohn, A. Stoica, D. Keymeulen, and S. Colombano, Eds. Los Alamitos, CA, USA: IEEE Computer Society, 2000, pp. 151–160.

[17] Y. Tao, L. Zhang, and Y. Zhang, "A projection-based decomposition for the scalability of evolvable hardware," *Soft Computing*, vol. 20, no. 6, pp. 2205–2218, Jun 2016.

[18] S. Rai, W. L. Neto *et al.*, "Logic synthesis meets machine learning: Trading exactness for generalization," *arXiv preprint arXiv:2012.02530*, 2020.

[19] C. Wolf, J. Glaser, and J. Kepler, "Yosys-a free verilog synthesis suite," in *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, 2013.

[20] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 24–40.

[21] M. Soeken, H. Riener, W. Haaswijk, E. Testa, B. Schmitt, G. Meuli, F. Mozafari, and G. De Micheli, "The EPFL logic synthesis libraries," Nov. 2019, arXiv:1805.05121v2.

Appendix E

Delay-aware evolutionary optimization of digital circuits

KOCNOVA Jitka and VASICEK Zdenek

In: *2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. Nicosia, Cyprus, 2022, pp. 188-193, doi: 10.1109/ISVLSI54635.2022.00045.

Delay-aware Evolutionary Optimization of Digital Circuits

Jitka Kocnova
Brno University of Technology,
Faculty of Information Technology,
Brno, Czech Republic
ikocnova@fit.vutbr.cz

Zdenek Vasicek
Brno University of Technology,
Faculty of Information Technology,
Brno, Czech Republic
vasicek@fit.vutbr.cz

***Abstract*—In the recent years, machine learning techniques have successfully been applied in various areas of digital circuit design including logic synthesis. Evolutionary resynthesis, among others, represents one of the machine learning approaches. This technique is based on local iterative optimization of parts of the original circuit. Even though the local optimization could be inefficient compared to the optimization conducted on the whole circuits, it has been shown that the resynthesis performs extremely well. It produces more compact solutions compared to the state-of-the art synthesis methods. In addition, it scales significantly better compared to the evolutionary optimization performed at the level of the original circuit. The previous methods have been focused solely on the optimization of the number of gates. In this paper, we analyse how the local optimization affects the delay of the resulting circuits and based on that, we propose a modified approach that considers the delay in the course of the optimization process. The proposed modification enables to maintain the delay of the optimized circuit at a reasonable level without a significant overhead. The evaluation conducted on a set of non-trivial highly optimized benchmark circuits representing various real-world circuits demonstrated that the proposed method is able to remove a significant number of gates while preserving the delay within the requested bounds.**

***Index Terms*—Logic optimization, Cartesian Genetic Programming, Evolutionary Resynthesis**

I. Introduction

Logic synthesis is a complex process consisting of a sequence of steps that transform a high-level description into a gate-level or transistor-level implementation. Even though there is a large body of research related to the logic synthesis, it has been shown recently that the results can be significantly improved by incorporating various machine learning techniques to the synthesis flow [1, 2, 3, 4].

Recently, an unconventional method inspired by the success of evolutionary approaches such as [3, 5] has been proposed in [6] and further elaborated in [7, 8]. The method, denoted as *evolutionary resynthesis*, combines the evolutionary optimization with the principle of Boolean network scoping. The combination of both techniques is used for maintaining the good scalability of the evolutionary synthesis process. The key idea is to use an iterative procedure which extracts sub-circuits that are subsequently optimized by evolutionary algorithm and implanted back into the original circuit provided that there is an improvement at the global level. The method is based on so called Cartesian Genetic Programming that manipulates with circuits represented using DAGs. Compared to the state-of-the-art synthesis tools such as ABC, the optimization is performed directly at the level of gates. This means that the nodes of DAG representing an optimized circuit are common gates. This eliminates the need of an intermediate representation such as and-inverter graph (AIG) that could negatively affect the performance of the logic optimization. AIG, for example, suffers from inherent bias in representation because every 2-input gate can be expressed using a single AIG node with except of the XOR/XNOR gates represented using three AIG nodes each. This bias is the main limitation of the common synthesis tools as they mostly rely on transformations disallowing an increase in the number of AIG nodes. To address the limitation of AIG representation, various approaches were proposed, see e.g. [1, 1, 1, 9]. The evolutionary resynthesis avoids the bias in representation, but for the cost of a much complicated optimization algorithm.

It has been shown that the efficiency of the evolutionary resynthesis depends mainly on efficiency of the method used for Boolean network scoping. A good network scoping method should produce sub-circuits that have a high chance to be further optimized. Three approaches have been investigated in the past. The first approach is based on computing k-feasible cuts [6]. The second approach uses the concept of windowing [7]. Both methods achieved significant reduction in terms of the number of removed gates, but the best results have been achieved with a more complex approach that is based on the identification of so called reconvergent paths [8]. On average, the latter method was able to remove more than 14.3% of gates when evaluated on a set of highly optimized benchmark problems representing various real-world controllers, logic and arithmetic circuits [8].

The previous works have focused exclusively on optimizing the number of gates, which is a legitimate requirement, but in practice it is usually necessary to control not only the area but also the depth of the circuits. If not constrained, the local optimization typically causes that the depth of the optimized circuits is increasing with the number of removed gates. Our goal is to modify the evolutionary resynthesis so that it can reduce the number of gates while keeping the depth of the optimized circuit within a predefined bound.

II. Background

This section presents relevant background on conventional as well as evolutionary optimization of logic circuits and introduces the notation used in the rest of the paper.

A. Boolean networks

The combinational circuits can be modelled as a directed acyclic graph (DAG) with nodes represented by Boolean functions [1]. Such a DAG is called *Boolean network*. Sources of the graph are the primary inputs (PIs) of the network and the sinks are the primary outputs (POs). The output of a node may be an input to other nodes called *fanouts*. The inputs of a node are called *fanins*. An edge connects two nodes in a fanin/fanout relationship.

B. Limiting the Scope of Boolean Networks

Network scoping ensures a good scalability of synthesis tools when working with large Boolean networks. It is a part of rewriting and refactoring. Three approaches were proposed in to limit the scope of logic synthesis to work only on a small portion of a Boolean network – *windowing*, *reconvergent paths selection* and *cut computation* [1]. We will further discuss only the first two approaches, as they produce significantly better results than the k-feasible cut based one.

The windowing algorithm determines the working area (a window) by taking a node (a root node) and two integers *m* and *n* defining the number of logic levels on the fanin/fanout sides of the node to be included in the resulting window. The transitive fanin is a set of nodes on the fanin side with distance-*m* or less from the root node. The transitive fanout is a set of nodes on the fanout side with distance-*n* or less from the root node. These sets are used to obtain the leaf and root sets determining the window. The window of a Boolean network N is a subnetwork $\mathrm{N}' \subseteq \mathrm{N}$ corresponding to the subset of nodes of the network containing nodes from root set and all nodes on paths between the leaf set and the root set. Nodes in the leaf set are not included in the window. The complete algorithm can be found in [1, 7].

The second approach is similar to the windowing. At first, so called reconvergent path is identified. Then, the reconvergent path is expanded to a window [8]. Reconvergent paths are those paths that lead from one source node through two different portions of the network and meet again in a single node n. According to the definition, the node n must be located in the fanout cone of the source node. Including reconvergent paths in a window increases a chance of a further optimization, as the window may contain more redundant nodes than other areas of the network.

C. Evolutionary Optimization of Logic Circuits

Evolutionary algorithms (EAs) have been used to synthesize and optimize logic circuits since the late nineties [1, 1]. Miller et al. proposed a special variant of genetic programming coined as Cartesian Genetic Programming (CGP) [1] used to synthesize novel implementations of arithmetic circuits [1]. Despite many advantages, the method inherently suffered from various scalability issues and only small problem instances were addressed. The scalability of CGP has been significantly improved by incorporating a SAT-based combinational equivalence checking engine in [3]. In the latest improvement, the equivalence checking is driven by counterexamples produced by SAT solver and the method can directly handle instances having hundreds of inputs and thousands of gates [8].

A candidate circuit is typically represented as a string of integers which fully specifies a Boolean network having n_i PIs and n_o POs containing up to n_n nodes. Each node has n_a inputs and represents a single gate with up to n_a inputs that can be connected either to the output of a node placed in the previous columns or directly to a PI. All PIs and node outputs are assigned with a unique index. This enables us to specify the circuit encoding. Let $n_a = 2$, which is the common configuration for gate-level circuits. Then, each node is associated with three integers (a, b, f); a and b defining the input connection and f for function code. The function of a node can be chosen from a set of n_f functions Γ. In addition to that, n_o integers specify where the POs are connected. An example of CGP encoding can be seen in Figure 1. Note that the fixed number of nodes n_n does not mean that all the nodes contribute to the POs. These features allow redundancy and flexibility of CGP. For more details, please refer to as [7].

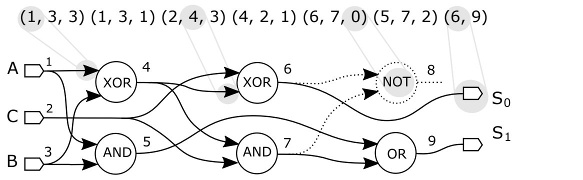


Fig. 1: A CGP encoding of one-bit full adder consisting of five active and one redundant gate. The circuit having $n_i = 3$ inputs and $n_o = 2$ outputs is encoded using a set of $n_n = 6$ two-input single-output nodes whose functions are chosen from a set of primitive functions $\Gamma = \{\mathrm{NOT}, \mathrm{AND}, \mathrm{OR}, \mathrm{XOR}\}$. Redundant node and connections that do not contribute to the outputs are highlighted using dotted line.

CGP is a population oriented approach operating with $1 + \lambda$ candidate solutions. The initial population is seeded by the original circuit ought to be optimized. Every new population contains the best circuit from the previous population and its λ offsprings created using so called mutation operator randomly modifying up to h integers. A single mutation causes either a) reconnection of a gate, or b) primary outputs, or c) change in function of a gate. Selection of the individuals is based on a cost function (e.g. the number of active nodes). If there are more individuals with the same score, the individual that has not served as a parent will be selected as the new parent. This procedure is repeated for a predefined number of iterations.

III. The proposed method

The pseudo-code of the proposed delay-aware optimization procedure based on the evolutionary resynthesis is shown in Algorithm 1. The algorithm expects a Boolean network N and produces its optimized version. The algorithm consists of three steps executed in a loop. At the beginning, N is assigned to N' (see line 1). In the first step, a working area (Boolean network W) is extracted from N' (see line 4). The goal is to obtain a smaller circuit which is easier to manipulate with. To extract a suitable sub-circuit, windowing [7] or windowing combined with identification of reconvergent paths [8] can be used. In the next step, W is optimized by means of the evolutionary algorithm (CGP) (see line 9). A constrained optimization reflecting the maximum allowed depth of N' specified by parameter D_{max} is performed in this step. The goal is to minimize the number of gates contained in W provided that the delay at the primary outputs of W stays within the required bounds. The bounds are determined according to the delay of the original circuit and the parameter τ. To avoid wasting computational resources, any W that is unlikely to lead to any improvement (e.g. small or thin windows) is discarded. Finally, the optimized logic network W' replaces W in N' which yields a new Boolean network N''. This network is evaluated with respect to the N' and if it receives better reward, it replaces the N' (see line 12). The optimization loop is terminated when a predefined number of iterations is exhausted.

Algorithm 1: Delay-aware EA-based resynthesis

```
   Input: Boolean network $N$, maximum allowed delay
          increase $1 \leq \tau < \infty$
   Output: Optimized network $N'$, $cost(N') \leq cost(N)$,
           $depth(N') \leq \tau \cdot depth(N)$
1  $N' \leftarrow N$
2  $D_{max} \leftarrow \tau \cdot depth(N')$
3  while terminated condition not satisfied do
4      $W \leftarrow$ ExtractSubcircuit($N'$)
5      if $W$ is not suitable candidate then
6          continue
7      $D_{PI} \leftarrow$ GetPIDelays($W, N'$)
8      $D_{PO} \leftarrow$ GetMaxAllowedPODelays($W, N', D_{max}$)
9      $W' \leftarrow$ OptimizeNetworkUsingCGP($W, D_{PI}, D_{PO}$)
10     $N'' \leftarrow (N' \setminus W) \cup W'$
11     if $cost(N'') \leq cost(N')$ then
12         $N' \leftarrow N''$
13 return $N'$
```

Cartesian Genetic Programming expects the Boolean network in the form of a netlist encoded using string of integers. This is achieved as follows. The nodes in W are topologically sorted and indexed. Then, every node is converted to the corresponding triplet of integers as described in Section II-C. Once provided, the string is then optimized by means of EA. The search procedure is driven by a cost function provided by the user. In the case of area optimization, the cost function simply reflects the number of active gates in CGP representation. One possibility how to to support the delay-aware optimization is to introduce a depth constraint into the cost function. This, however, leads to wasted evaluations caused by violating the maximum depth requirement due to the unconstrained changes by mutation operator. To avoid this kind of inefficiency, we decided to modify the mutation operator itself. Instead of allowing to connect a mutated gate to any node topologically preceding the mutated one, we restrict the possibilities to those nodes guaranteeing that the maximum output delay does not violate the predefined constraints. To achieve that, the mutation operator needs to know delay at the inputs of a mutated gate and maximum allowed output delay. The input delay is calculated using the delay at primary inputs (denoted as D_{PI}). The maximum allowed output delay is determined by the maximum allowed delay on the primary outputs (denoted as D_{PO}) that is propagated from the outputs to the mutated node. Based on these informations, list of suitable nodes for connection is determined.

IV. Experimental evaluation

A. Experimental Setup

The proposed method was implemented a part of Yosys framework [1]. Two approaches for subcircuit extraction are supported: common windowing (further denoted as GSW) and reconvergent-paths based one (further denoted as GSRW). The evaluation was carried out on 28 highly optimized real-world circuits (IWLS'05 Open Cores benchmarks and a set of arithmetic circuits[1]). As in [7], the benchmarks were optimized by ABC and mapped to common 2-input gates. The goal of the optimization process is to reduce the number of gates while preserving the delay at the required level. For every benchmark, six levels of τ were considered: $\tau = 1$ which means that delay at any PO should not be worse than the maximum delay of the original circuit (i.e. the maximum delay must be preserved), $\tau = \{1.1, 1.2, 1.5, 1.75\}$ which allows delay increase by 10, 20, 50 and 75%, and $\tau = \infty$ which does not put any restrictions on the delay. The latter case corresponds to the original counterparts of GSW and GSRW presented in [7] and [8].

Five independent runs were executed for every benchmark to obtain statistically sound results. All of the optimized circuits were formally verified with respect to their original form (ABC command 'cec') to avoid any error in the evaluation. The procedure OptimizeNetworkUsingCGP is implemented as described in Section II-C. The parameters are: $n_a = 2$, $\lambda = 1$, $h = 2$, $n_n = |W|$, $\Gamma = \{\text{BUF}, \text{NOT}, \text{AND}, \text{OR}, \text{XOR}, \text{NAND}, \text{NOR}, \text{XNOR}\}$. To match the setup with that used in our previous works, the procedure is executed $n_{iters} = 2 \cdot 10^4$ times and $n_{evals} = 5 \cdot 10^5$ candidate solutions are generated in each iteration. Four volume settings for GSRW are considered: $rp_{max} = (10, 20, 50, 100)$ gates. The minimal and maximal volume of the selection is set to $rw_{min} = 5$, $rw_{max} = 100$ nodes. For $\tau = 1$, the gates on the longest path were omitted during the sub-circuit selection in order to minimize the chance of extending the delay during the optimization.

B. Experimental Results

The overall statistics calculated from the all considered experiments are summarized in Tab. I (GSW) and Tab. II

[1]The benchmarks can be downloaded from https://lsi.epfl.ch/MIG

TABLE I: The achieved gate reduction for the proposed method GSW. Average relative improvement in the number of gates compared to the optimized circuits obtained using ABC are provided. Column 'ABC' contains parameters of the optimized circuits after mapping, namely $|N|$ denoting the number of utilized gates and $D = depth(N)$. Column D' is depth of N' obtained for $\tau = \infty$.

			ABC		Relative gate reduction with respect to $\|N\|$							
Benchmark	PIs	POs	$\|N\|$	D	$\tau = 1.0$	$\tau = 1.1$	$\tau = 1.2$	$\tau = 1.5$	$\tau = 1.75$	$\tau = \infty$ [8]	D'	D'/D
DSP	4223	3792	43491	45	0.2%	0.4%	0.3%	0.2%	0.1%	1.4%	62	1.4
ac97_ctrl	2255	2136	11433	10	2.2%	2.1%	2.9%	2.9%	2.2%	3.0%	13	1.3
aes_core	789	532	21128	20	0.5%	1.1%	0.3%	0.4%	0.3%	4.2%	28	1.4
des_area	368	70	5199	25	0.3%	0.6%	0.5%	0.6%	0.8%	4.5%	35	1.4
des_perf	9042	1654	78972	16	0.1%	0.1%	0.1%	0.1%	0.1%	2.8%	24	1.5
ethernet	10672	10452	60413	23	0.3%	0.1%	0.1%	0.1%	0.1%	1.6%	33	1.4
i2c	147	127	1161	12	7.5%	11.8%	15.0%	18.4%	17.8%	18.3%	21	1.8
mem_ctrl	1198	959	10459	24	0.7%	2.0%	0.9%	2.7%	1.9%	6.2%	44	1.8
pci_bridge32	3519	3136	19020	21	0.9%	0.5%	1.3%	1.2%	1.1%	3.4%	47	2.2
pci_spoci_ctrl	85	60	1136	15	1.3%	14.3%	12.7%	15.3%	17.3%	31.5%	23	1.5
sasc	133	123	746	8	4.5%	4.5%	4.9%	6.1%	6.1%	7.4%	7	0.9
simple_spi	148	132	822	11	2.5%	3.2%	4.5%	4.4%	4.6%	6.6%	18	1.6
spi	274	237	3825	26	1.2%	2.9%	4.0%	1.2%	4.0%	5.0%	32	1.2
ss_pcm	106	90	437	7	1.2%	1.5%	5.1%	2.8%	2.7%	4.8%	11	1.6
systemcaes	930	671	11352	27	0.5%	0.2%	0.2%	0.4%	1.5%	11.7%	32	1.2
systemcdes	314	126	2601	25	2.2%	5.7%	10.4%	11.1%	13.3%	15.7%	31	1.2
tv80	373	360	8738	39	1.6%	0.4%	2.6%	2.3%	3.1%	13.5%	57	1.5
usb_funct	1860	1692	15405	23	0.5%	1.3%	1.0%	1.2%	1.3%	10.2%	47	2.0
usb_phy	113	73	452	9	13.2%	11.1%	16.8%	15.4%	13.8%	17.7%	12	1.3
average (IWLS'05 benchmarks)			15620	–	2.2%	3.4%	4.4%	4.6%	4.8%	8.9%	–	1.5
mult32	64	64	8225	42	0.7%	0.7%	0.6%	0.7%	1.3%	19.5%	85	2.0
sqrt32	32	16	1462	307	0.7%	3.4%	2.9%	3.4%	3.3%	6.6%	370	1.2
diffeq1	354	193	20719	218	2.0%	2.0%	1.4%	1.2%	0.7%	25.5%	250	1.1
div16	32	32	5847	152	0.5%	4.4%	4.0%	5.7%	2.0%	29.5%	280	1.8
hamming	200	7	2724	80	0.9%	12.9%	9.0%	10.3%	12.1%	58.8%	83	1.0
MAC32	96	65	7793	55	1.0%	2.0%	1.4%	3.6%	2.6%	9.5%	142	2.6
revx	20	25	8131	171	1.7%	1.2%	4.7%	4.2%	5.1%	18.0%	221	1.3
max	512	130	3719	117	0.4%	4.6%	3.2%	2.4%	3.6%	5.1%	127	1.1
average (arithmetic benchmarks)			8956	–	1.0%	3.9%	3.4%	3.9%	3.8%	21.6%	–	1.5

(GSRW). The first three columns of both tables contain information about the benchmarks (name, number of PIs and POs). The next two columns show parameters of the optimized and mapped circuits produced by ABC; the number of gates and logic depth are given. These numbers serve as a baseline for our comparison. Then, the achieved improvement expressed as the relative reduction with respect to the baseline is reported for the windowing and reconvergent paths based methods. For each method, we report the average improvement in means of reduction of the gates for every benchmark circuit. The average is calculated from all five independent runs. For GSRW method, however, twenty runs are used because of various values of rp_{max} parameter. The last three columns show parameters of the best optimized circuit obtained for $\tau = \infty$, i.e. the case when only the area is minimized. In particular, we included the relative gate reduction, delay of the resulting circuit, and ratio between the delay of the original and optimized circuit are provided. Due to the limited space, we do not report the delay for other configurations, but the delay is in fact typically equal or close to τD.

When we compare the average results of constrained ($\tau < \infty$) and unconstrained ($tau = \infty$) optimization, we can see that the limitation of the maximum delay (logic depth) has a substantial impact on the size of the obtained circuits. This observation is valid for GSW as well as GSRW method. The achieved gate reduction of the constrained optimization is much lower compared to the unconstrained one and it increases with increasing τ. It is clear that we can't achieve the same results in cases where the ratio D'/D is higher than τ. This is the case of pci_bridge32, for example. When optimized with the unconstrained setup, the benchmark size decreased by 3.4% on the average. However, the logical depth increased by a factor of 2.2. Given that the maximum allowable τ is 1.75, it is clear that the same reduction cannot be achieved. There are also cases, however, where the ratio D'/D is lower than 1.75 and yet the constrained optimization did not achieve the same or even a comparable reduction. This is particularly noticeable in the case of arithmetic instances such as mult32, div16, diffeq1 or revx.

We analysed the parameters of the circuits in the course of the optimization when $\tau = \infty$. We found that the depth of the optimized circuits usually increases in each iteration as the number of removed gates increases. At some point it reaches a maximum and then starts to decrease. After a certain number of iterations, the decreasing phase stops and the number of gates usually start to oscillate around a certain value. This behaviour is typical for controller circuits. For arithmetic benchmarks which are much more complex, the

TABLE II: The achieved gate reduction for the proposed method using windowing based on the reconvergent paths (GSRW). Average relative improvement in the number of gates compared to the optimized circuits obtained using ABC are provided. Column 'ABC' contains parameters of the optimized circuits after mapping, namely $|N|$ denoting the number of utilized gates and D=depth(N). Column D' is depth of N' obtained for $\tau = \infty$.

			ABC		Relative gate reduction with respect to $\|N\|$							
Benchmark	PIs	POs	$\|N\|$	D	$\tau = 1.0$	$\tau = 1.1$	$\tau = 1.2$	$\tau = 1.5$	$\tau = 1.75$	$\tau = \infty$ [8]	D'	D'/D
DSP	4223	3792	43491	45	0.2%	0.5%	0.1%	0.2%	0.1%	4.9%	71	1.6
ac97_ctrl	2255	2136	11433	10	1.0%	2.6%	2.7%	2.5%	2.5%	4.6%	14	1.4
aes_core	789	532	21128	20	0.6%	0.8%	0.3%	0.2%	0.3%	8.5%	32	1.6
des_area	368	70	5199	25	0.2%	0.3%	0.1%	0.4%	0.4%	6.4%	35	1.4
des_perf	9042	1654	78972	16	0.1%	0.1%	0.1%	0.1%	0.1%	7.3%	26	1.6
ethernet	10672	10452	60413	23	0.2%	0.1%	0.1%	0.1%	0.1%	1.9%	30	1.3
i2c	147	127	1161	12	5.8%	8.6%	13.2%	15.3%	16.4%	24.7%	27	2.2
mem_ctrl	1198	959	10459	24	0.6%	0.5%	1.2%	2.5%	2.6%	9.6%	45	1.9
pci_bridge32	3519	3136	19020	21	0.1%	0.6%	0.4%	0.9%	0.8%	6.7%	46	2.2
pci_spoci_ctrl	85	60	1136	15	6.8%	4.6%	9.1%	17.5%	35.1%	43.0%	23	1.5
sasc	133	123	746	8	5.1%	1.9%	6.2%	7.3%	5.2%	21.0%	10	1.2
simple_spi	148	132	822	11	1.2%	4.3%	4.4%	5.0%	5.9%	11.9%	28	2.5
spi	274	237	3825	26	0.9%	0.3%	0.8%	2.3%	4.6%	9.3%	32	1.2
ss_pcm	106	90	437	7	2.8%	2.1%	6.2%	6.2%	5.7%	12.4%	12	1.7
systemcaes	930	671	11352	27	1.2%	0.9%	0.3%	0.6%	0.2%	12.1%	36	1.3
systemcdes	314	126	2601	25	2.9%	13.9%	11.9%	10.1%	8.3%	19.5%	32	1.3
tv80	373	360	8738	39	1.0%	3.2%	3.1%	1.7%	1.6%	10.5%	61	1.6
usb_funct	1860	1692	15405	23	0.6%	0.3%	0.2%	0.3%	2.4%	10.4%	31	1.3
usb_phy	113	73	452	9	15.6%	17.7%	15.5%	17.1%	16.2%	29.1%	13	1.4
average (IWLS'05 benchmarks)			15620	20	2.5%	3.3%	4.0%	4.8%	5.7%	13.4%	–	1.6
mult32	64	64	8225	42	1.8%	2.6%	3.6%	3.6%	1.4%	21.6%	61	1.5
sqrt32	32	16	1462	307	0.2%	1.7%	2.3%	1.9%	1.1%	17.1%	339	1.1
diffeq1	354	193	20719	218	0.1%	0.5%	0.1%	0.4%	0.2%	8.7%	283	1.3
div16	32	32	5847	152	0.7%	1.0%	1.2%	0.9%	2.2%	20.6%	247	1.6
hamming	200	7	2724	80	1.9%	12.4%	13.9%	17.5%	21.3%	45.9%	87	1.1
MAC32	96	65	7793	55	2.8%	1.5%	1.1%	1.4%	1.7%	5.5%	60	1.1
revx	20	25	8131	171	0.6%	2.7%	2.3%	2.8%	2.1%	7.9%	128	0.7
max	512	130	3719	117	1.1%	2.7%	2.0%	2.2%	1.3%	5.2%	132	1.1
average (arithmetic benchmarks)			8956	148	1.1%	3.1%	3.3%	3.8%	3.9%	16.6%	–	1.2

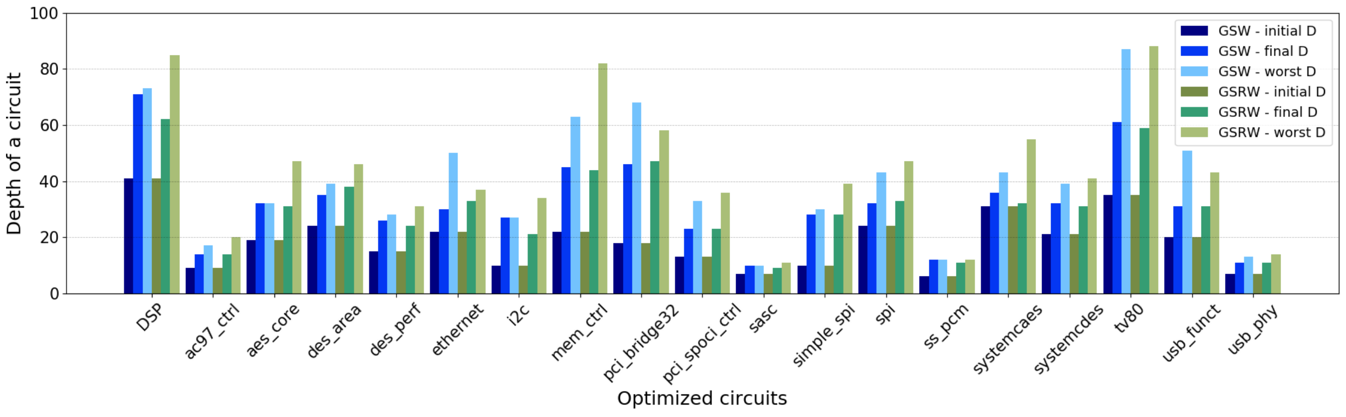
decreasing phase is much shorter. The relationships between the initial depth, the final depth and the worst observed depth are shown in Figure 2. We assume that this behavior is due to the nature of evolutionary optimization. As the optimized circuit becomes smaller, the chance of gate removal in the extracted subcircuits decreases. However, this does not mean that the subcircuit cannot be modified and restructured. Since the depth is not under control when $\tau = \infty$, the depth can grow indefinitely. Interestingly, there is an implicit regulatory mechanism that prevents from growing indefinitely.

Comparing the overall results across different values of τ, we can observe that GSRW performs slightly better for controller circuits. For arithmetic circuits, however, there is no significant difference between the performance of GSW and GSRW method. The success of the optimization is highly dependent on the structure of a particular benchmark. There are benchmarks that offer a high degree of freedom with respect to redundancy. On the other hand, there are cases that can be reduced only slightly. This suggests that the initial circuit optimized by ABC has probably no redundancy. Despite of the fact that the constrained optimization achieves worse results compared to the unconstrained one, the results are still encouraging because in all the cases we were able to improve the initial highly optimized circuits even for $\tau = 1$, a setting that requires keeping the worst-case delay at the same value as the initial circuit. When we enable to increase the the circuit's depth, the improvement increases. In the case of the GSRW method, for example, the maximum number of removed gates was around 35% (pci _spoci_ctrl) and more than 21% (hamming), while the circuit's depth was increased by 75% of the original value.

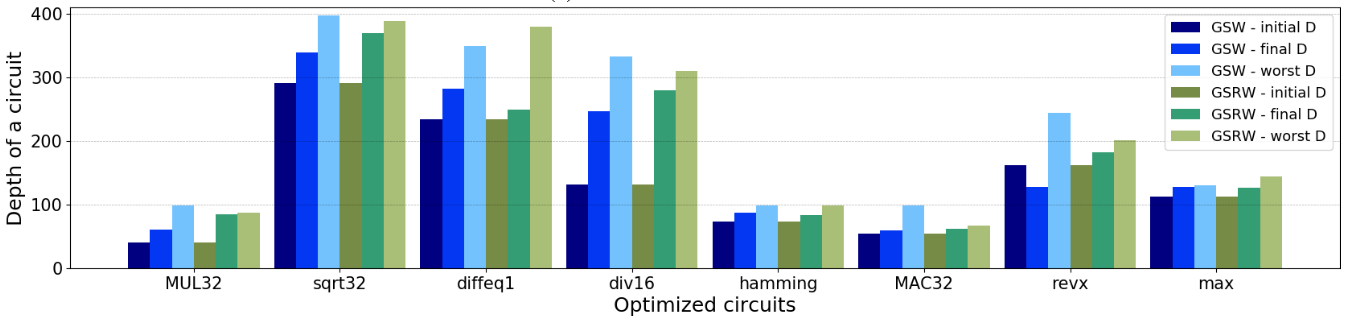
V. Conclusion

EA-based optimization is able to improve the results of the conventional logic synthesis. However, its performance deteriorates with increasing circuit complexity. Previous works successful addressed this problem by combining the EA-based optimization with the principle of the Boolean network scoping. However, this approach could lead to a significant increase in the depth of the circuits. We solved this problem by constraining the optimization to force the EA to explore only candidate solutions satisfying a depth limit. In order to do that, we modified the mutation operator.

The evaluation performed on a set of complex benchmark circuits confirmed that it is possible to constrain the depth according to the user requirements and still achieve a good reduction in the number of gates. However, the limitation of the circuit's depth has a significant effect on the achieved reduction.



(a) Controller benchmarks



(b) Arithmetic benchmarks

Fig. 2: The initial, final and worst depth observed during optimization using GSW and GSRW methods for $\tau = \infty$.

ACKNOWLEDGMENT

This work was supported by the Czech science foundation project GA22-02067S.

REFERENCES

[1] G. Huang, J. Hu *et al.*, "Machine learning for electronic design automation: A survey," vol. 26, no. 5, 2021.

[2] P. Fiser, J. Schmidt, Z. Vasicek, and L. Sekanina, "On logic synthesis of conventionally hard to synthesize circuits using genetic programming," in *13th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems*, 2010, pp. 346–351.

[3] Z. Vasicek, "Cartesian GP in optimization of combinational circuits with hundreds of inputs and thousands of gates," in *Proceedings of the 18th European Conference on Genetic Programming*, ser. LCNS 9025. Springer International Publishing, 2015, pp. 139–150.

[4] S. Rai, W. L. Neto, Y. Miyasaka, X. Zhang, M. Yu, Q. Y. M. Fujita, G. B. Manske, M. F. Pontes, L. S. Junior, M. S. de Aguiar *et al.*, "Logic synthesis meets machine learning: Trading exactness for generalization," *arXiv preprint arXiv:2012.02530*, 2020.

[5] L. Sekanina, O. Ptak, and Z. Vasicek, "Cartesian genetic programming as local optimizer of logic networks," in *2014 IEEE Congress on Evolutionary Computation*. IEEE CIS, 2014, pp. 2901–2908.

[6] J. Kocnova and Z. Vasicek, "Towards a scalable EA-based optimization of digital circuits," in *Genetic Programming*. Cham: Springer International Publishing, 2019, pp. 81–97.

[7] ——, "EA-based resynthesis: An efficient tool for optimization of digital circuits," *Genetic Programming and Evolvable Machines*, vol. 21, no. 3, pp. 287–319, 2020.

[8] J. Kocnová and Z. Vasicek, "Resynthesis of logic circuits using machine learning and reconvergent paths," in *2021 24th Euromicro Conference on Digital System Design (DSD)*, 2021, pp. 69–76.

[9] C. Yang and M. Ciesielski, "BDS: a BDD-based logic optimization system," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 21, no. 7, pp. 866–876, Jul 2002.

[10] N. Vemuri, P. Kalla, and R. Tessier, "BDD-based logic synthesis for LUT-based FPGAs," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 7, no. 4, pp. 501–525, Oct. 2002.

[11] L. Amaru, P. E. Gaillardon, and G. D. Micheli, "Mixsyn: An efficient logic synthesis methodology for mixed xor-and/or dominated circuits," in *2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2013, pp. 133–138.

[12] P. Fiser, I. Halecek, and J. Schmidt, "Sat-based generation of optimum function implementations with xor gates," in *2017 Euromicro Conference on Digital System Design (DSD)*, 2017, pp. 163–170.

[13] A. Mishchenko and R. Brayton, "Scalable logic synthesis using a simple circuit structure," in *Int. Workshop on Logic and Synthesis*, 2006, pp. 15–22.

[14] J. D. Lohn and G. S. Hornby, "Evolvable hardware: Using evolutionary computation to design and optimize hardware systems," *IEEE Computational Intelligence Magazine*, vol. 1, no. 1, pp. 19–27, 2006.

[15] J. Miller and P. Thomson, "Cartesian Genetic Programming," in *Proc. of the 3rd European Conference on Genetic Programming EuroGP2000*, ser. LNCS, vol. 1802. Springer, 2000, pp. 121–132.

[16] J. F. Miller, *Cartesian Genetic Programming*. Springer, 2011.

[17] V. Vassilev, D. Job, and J. F. Miller, "Towards the Automatic Design of More Efficient Digital Circuits," in *Proc. of the 2nd NASA/DoD Workshop on Evolvable Hardware*, J. Lohn, A. Stoica, D. Keymeulen, and S. Colombano, Eds. Los Alamitos, CA, USA: IEEE Computer Society, 2000, pp. 151–160.

[18] C. Wolf, J. Glaser, and J. Kepler, "Yosys-a free verilog synthesis suite," in *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, 2013.