Resynthesis of logic circuits using machine learning and reconvergent paths

1st Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

2nd Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

***Abstract*—Boolean network scoping represents a common approach incorporated in conventional synthesis tools for maintaining the 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 is highly influenced by the efficiency of the sub-circuit extraction process. We propose a new method, based on the reconvergent paths identification. The evaluation is done on a set of highly optimized complex benchmark problems representing various real-world controllers, logic and arithmetic circuits. It provides better results compared to the state-of-the-art logic synthesis tool and both locally and globally operating evolutionary optimizations presented earlier. 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 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. 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 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. 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 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. 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 on the rest of the network. In 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. employed XOR majority graphs (XMGs) to extend the capabilities of exact synthesis oriented on area optimization. To summary, the methods perform either a preprocessing or circuit decomposition [6] or precomputation of ideal solutions [7]; other methods rely on XOR transformations or presence technology cells (eg. XMGs).

Other authors tried to avoid intermediate representation and the need for circuit preprocessing. Various machine-learning approaches working directly at the level of gates were successfully applied to address this problem [1, 8]. 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 [8]. 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 synthesis of conventionally hard to synthesize circuits [2]. Many other machine learning techniques have been recently used for circuit synthesis, see e.g. [9] which lists various methods used to the synthesis of incompletely-specified functions. 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, 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 has been shown, 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. Huge sub-circuits, on the other hand, increase 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. 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. The other one was based on so called windowing method. Both methods achieved significant reduction w.r.t. the number of gates in the benchmark circuits. However, the authors found out that their approach to the sub-circuit selection may cause an inefficiency in the optimization. It happens when there is an attempt to optimize a sub-circuit that lacks the redundancy nodes or the interconnection between its nodes are poor. Therefore the evolutionary optimization spends a significant ammount of time searching for a solution that has already been reached instead of moving on to the optimization of another sub-circuit.

Our goal is to improve the performance of the the sub-circuit selection – mainly to focus on particular areas in the circuits that may potentionaly 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

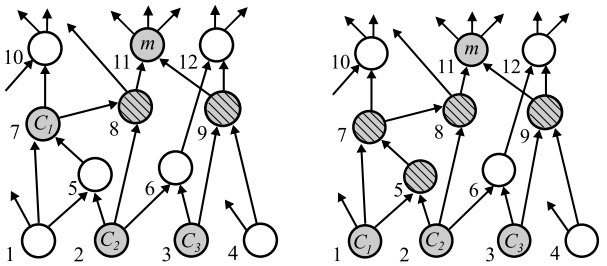
Every circuit can be represented using a Boolean network. A *Boolean network* is a directed acyclic graph (DAG) with nodes represented by Boolean functions [12]. The sources of the graph are the primary inputs (PIs) of the network and the sinks are the primary outputs (POs). The output of a node may 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 $\mathrm{N}\prime \subseteq \mathrm{N}$ 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 few nodes and leads to tree-like logic structures. Such a structure does not lead to any



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

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

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 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 late nineties [13, 14]. Miller et al., the author of Cartesian Genetic Programming (CGP) [15], 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 [16]. 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 [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 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 [8]. Neither the original nor the latter approach rely on a decomposition of the optimized circuits.

Since its introduction, CGP remains the most powerful evolutionary technique [14]. 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 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 to avoid 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. 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 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.

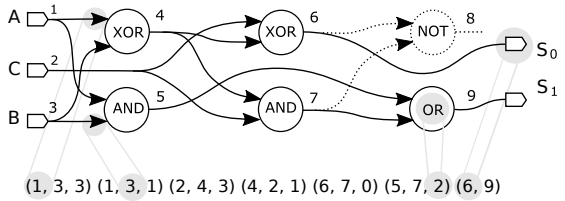


Fig. 2: Example of a CGP individual encoding a logic circuit (one-bit full adder) with $n_i = 3$ inputs and $n_o = 2$ outputs. The individual is encoded using an array of $n_n = 6$ two-input single-output nodes whose functions are chosen from a set of primitive functions $\Gamma = \{\mathrm{NOT}, \mathrm{AND}, \mathrm{OR}, \mathrm{XOR}\}$. 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 with 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 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 [11]

Input: A Boolean network N
Output: Optimized network N', $cost(N') \leq cost(N)$

$N' \leftarrow N$
while *terminated condition not satisfied* **do**
  $m \leftarrow$ identify the best candidate root node $m \in N'$
  $W \leftarrow$ GetSubcircuit(m)
  if W *is a suitable candidate* **then**
    $W' \leftarrow$ OptimizeNetworkUsingEA(W)
    if $\text{cost}((N' \setminus W) \cup W') < \text{cost}(N')$ **then**
      $N' \leftarrow (N' \setminus W) \cup W'$
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 few nodes, mainly when working with networks of a small depth. In order to maximize the volume of W, the 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 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 limitation 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 in a receiving node that is in the 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: Reconvergence path-based procedure GetSubcircuit

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, $rw_{min} \leq |W| \leq rw_{max}$

$RW \leftarrow \emptyset$
$rp \leftarrow \emptyset$
$init\_roots \leftarrow$ randomly select 10 nodes from N
foreach $rm \in init\_roots$ **do**
  $rp \leftarrow$ select a reconvergence path starting from rm containing rp_{max} nodes
if rp *is empty* **then**
  $rp \leftarrow$ select random node from $init\_roots$
push all nodes from rp to RW
init queue q with rp
while q *not empty* $\wedge$ $|RW| < rw_{max}$ **do**
  $rm \leftarrow$ pop a node from q
  $RW \leftarrow RW \cup \{rm\}$
  $X \leftarrow fanin(rm) \cup fanout(rm)$
  push all nodes from $X \setminus RW$ that are not already in q into q
if $|RW| < rw_{min}$ **then**
  $RW \leftarrow \emptyset$
$W \leftarrow \bigcup_{rm \in RW} fanin(rm)$
return RW

The principle of the proposed selection strategy is shown in Algorithm 2. The input is a boolean network N and the output is a set of nodes (selection) denoted as RW. At first, a search for a reconvergence path rp of a desired volume (in means of number of gates within the path) is done. This search is done for a 10 randomly selected nodes from the circuit - subset $init\_roots$. The first rp that is found proceeds to the next step of the algorithm and its root node becomes the rm node. If the reconvergent path is not found for any node from $init\_roots$, the rp is initiated only with a randomly chosen

root node from the $init\_roots$. Then, all the nodes from rp are copied to RW (see line 8). 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.

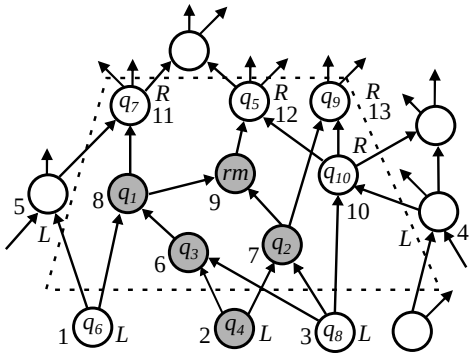


Fig. 3: Example of the window consisting of 10 nodes created using the reconvergence path selection algorithm 2. The reconvergence path rp starting in the root node rm is highlighted using the filled nodes. The nodes q_5–q_{10} are those added during the final expansion of the selection RW. The nodes 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.

IV. Experimental evaluation

A. Experimental setup

The proposed method was implemented in C++ as a part of Yosys open synthesis suite [18]. This tool allows us to directly manipulate with Verilog files and it integrates ABC [19], 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 [20]. 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] – 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, we will include the results from ABC 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 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 the area-optimization. The only criterion in the fitness function considered in this paper is the area on a chip expressed as the number of gates. Thus the improvement is measured in terms of the number of removed gates. The other 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 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 [8]. 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 both in the proposed GSRW method and the GSW method. 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 termination conditions are designed as follows. The GSRW and GRW methods are 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 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 reaches $t_{max} \times n_{iters}$ 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 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

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

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

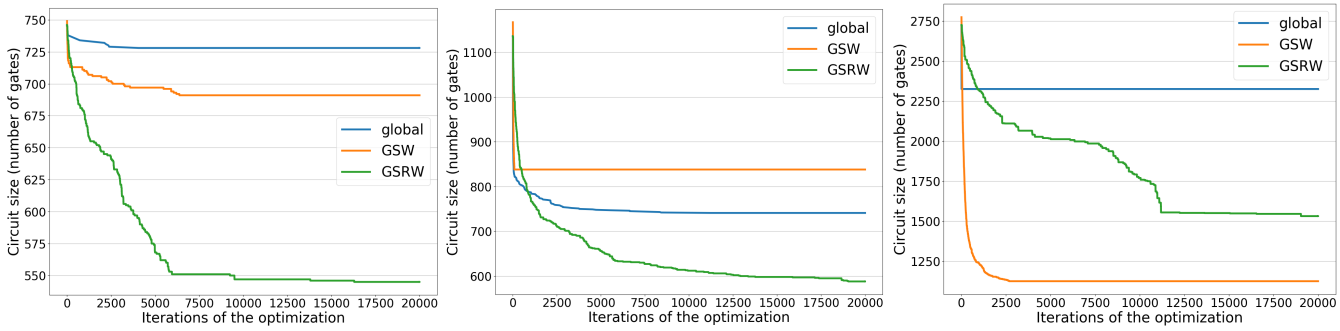
The overall results obtained from all of the considered approaches are summarized in Tab. I. The first three columns contain information about the benchmarks (name, number of PIs and POs). The next two columns show parameters of the optimized and mapped circuits produced by ABC; the number of gates and 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 is 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 the 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 non-deterministic (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 ends at a local optima. Both the 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 the Figure 4, the GSW method reaches its solution earlier than the GSRW method. However, the GSRW compensates the slower convergence with better utilization of the computation time thanks to the suitable structure of the sub-circuits. Hence, substantially higher number of the total 10^{10} candidate solutions generated during the optimization successfully participated in the final circuit reduction.

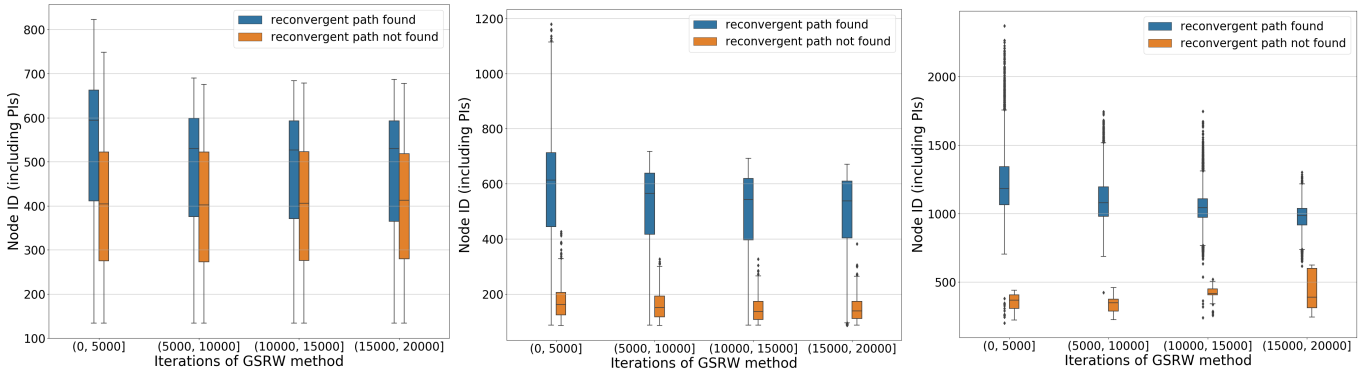
Considering the arithmetic circuits, the GSRW performs worse compared to the GSW method. In five cases ('hamming', 'diffeq1', 'div16', 'MAC32', 'revx'), the performance of the GSRW was surpassed by the GSW. When examining the convergence curves and parameters of the produced windows, we came to a circumstance which is responsible for those results and is connected to the structure of the circuits. The Figure 5 shows the numeral IDs of root nodes of the every one of the sub-circuit identified using the GSRW method. The blue ones represent those, for which the desired reconvergent path was found and the orange ones are those for which the reconvergent path does not exist and the same window-like selection as in the GSW was established. 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 clearly 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 thin 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 root node dissipation 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 root node dissipation 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 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



(a) Sasc (b) Pci_spoci (c) Hamming

Fig. 4: Convergence trough all of the generations



(a) sasc (b) pci_spoci (c) Hamming

Fig. 5: Root location through the iterations.

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 stuck to a limited set of reconvergent paths as it had in some of our experiments.

Acknowledgment

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.

[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

TABLE I: Comparison of the proposed method against ABC, the method using the windowing algorithm and the CGP working globaly. The columns 'Impr. GSRW(proposed)', 'Impr. GSW' 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. GSRW (proposed)		Impr. GSW		Impr. global [8]	
Benchmark	PIs	POs	gates	D	avg	best	avg	best	avg	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%

circuits," in *2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 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] 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.

[9] 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.

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

[19] 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.

[20] 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.