EA-based resynthesis: an efficient tool for optimization of digital circuits

Jitka Kocnova & Zdenek Vasicek

Genetic Programming and Evolvable Machines

ISSN 1389-2576
Volume 21
Number 3

Genet Program Evolvable Mach (2020)
21:287-319
DOI 10.1007/s10710-020-09376-3

Springer

Your article is protected by copyright and all rights are held exclusively by Springer Science+Business Media, LLC, part of Springer Nature. This e-offprint is for personal use only and shall not be self-archived in electronic repositories. If you wish to self-archive your article, please use the accepted manuscript version for posting on your own website. You may further deposit the accepted manuscript version in any repository, provided it is only made publicly available 12 months after official publication or later and provided acknowledgement is given to the original source of publication and a link is inserted to the published article on Springer's website. The link must be accompanied by the following text: "The final publication is available at link.springer.com".

Springer

Author's personal copy

Genetic Programming and Evolvable Machines (2020) 21:287–319
https://doi.org/10.1007/s10710-020-09376-3

EA-based resynthesis: an efficient tool for optimization of digital circuits

Jitka Kocnova[1] · Zdenek Vasicek[1]

Received: 13 October 2019 / Revised: 1 January 2020 / Published online: 30 January 2020
© Springer Science+Business Media, LLC, part of Springer Nature 2020

Abstract

Since the early nineties the lack of scalability of fitness evaluation has been the main bottleneck preventing the adoption of evolutionary algorithms for logic circuits synthesis. 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 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
vasicek@fit.vutbr.cz

Extended author information available on the last page of the article

Author's personal copy

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. 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 an 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 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 [3, 4, 21]. 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 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

Author's personal copy

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 evolutionary optimization with the principle of so called Boolean network scoping. Boolean network scoping represents a common approach incorporated in 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 Sect. 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 rewriting, evolutionary resynthesis has the potential to reduce a substantially larger number of gates (e.g. low hundreds of gates).

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.

Author's personal copy

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 Sect. 5. Finally, Sect. 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-based network scoping* [18].

The windowing algorithm determines the working area denoted as *window* by computing so called 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 corresponds to a set of nodes on the fanin side that are distance-m or less from the pivot node. Similarly, the transitive fanout corresponds to a set of nodes on the fanout side that are distance-n or less from the pivot

Author's personal copy

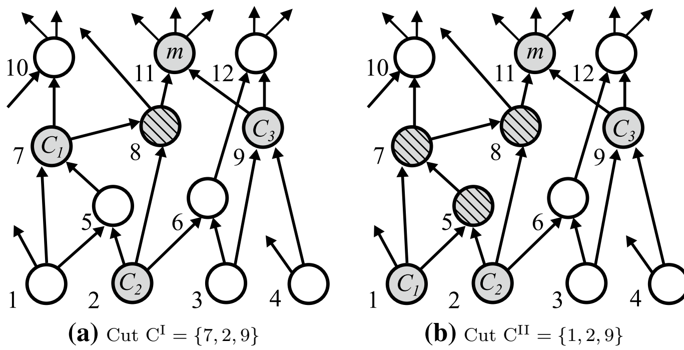

Fig. 1 Example of two possible 3-feasible cuts $\{C_1, C_2, C_3\}$ for root node m and given Boolean network consisting of 12 nodes. The nodes belonging to a particular cut are highlighted in grey color. The hatched nodes correspond to the contained nodes. The cut C^{II} is preferred as its volume is four (root node m and contained nodes 5, 7, and 8). The volume of the cut C^{I} is two because this cut contains only one contained node (node 8)

node. These two sets are then used to obtain so called leaf and root sets that uniquely determine the window[1]. The complete algorithm can be found in [18]. 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.

Apart from the windowing, many logic synthesis algorithms uses network scoping based on computing so called k-feasible cuts. For example, the rewriting is based on 4-feasible cuts [18]. The principle of this technique is to compute a cut which is subsequently expanded to a window. 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. An example of two different 3-feasible cuts is shown in Fig. 1. A reconvergence-driven heuristic is typically applied in practice to maximixe the cut volume, i.e. the total number of nodes encountered on all paths between the root node and the cut leaves. The problem is that the cut computed using a naïve 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 naïve approach and Fig. 1b showing the output of reconvergence-driven heuristic). A tree-like logic structure does not lead to any redundancy and attempting optimization using such cuts would be wasted time.

A simple and efficient cut computation algorithm producing a cut close to a required 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 uses the network scoping based on computation of the k-feasible

[1] The window of a Boolean network N is a connected subnetwork $N' \subseteq N$ 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.

Author's personal copy

reconvergence-driven cuts, we briefly introduce this algorithm. The algorithm starts with a set of leaves consisting of a single root node (i.e. a trivial cut). This set is incrementally expanded 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 sequence of four steps leading to the cut $C^{II} = \{1, 2, 9\}$ shown in Fig. 1b is as follows: $\{\underline{11}^{\langle 2\rangle}\} \rightarrow \{\underline{8}^{\langle 2\rangle}, 9^{\langle 2\rangle}\} \rightarrow \{\underline{7}^{\langle 2\rangle}, 2^{\langle\infty\rangle}, 9^{\langle 2\rangle}\} \rightarrow \{\underline{5}^{\langle 0\rangle}, 1^{\langle\infty\rangle}, 2^{\langle\infty\rangle}, 9^{\langle 2\rangle}\} \rightarrow \{1^{\langle\infty\rangle}, 2^{\langle\infty\rangle}, 9^{\langle 2\rangle}\}$. The node removed in each step is underlined. The cost of the nodes included in the cut is shown using a superscript. The infinity means that the node is a primary input which cannot be removed from the set. Note that the algorithm returns the set $\{1, 2, 9\}$ because removal of the node 9 would produce a 4-feasible cut (nodes 3 and 4 would be included instead of 9, both with infinite cost).

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-circuit. Otherwise it would be impossible to replace the whole subcircuit by a precomputed optimal implementation / a single LUT. Compared to the windowing, much smaller windows are typically produced. Typically, 4-feasible and 5-feasible cuts are used for rewriting-based logic synthesis [12, 18]. 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, a 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.

Gate-level evolution has rarely been addressed before the year 2000. The first results in the area of digital circuit synthesis were reported by Koza in 1992, who

Author's personal copy

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 [14, 17]. 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 [15, 20, 22, 23, 31]. 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 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

Author's personal copy

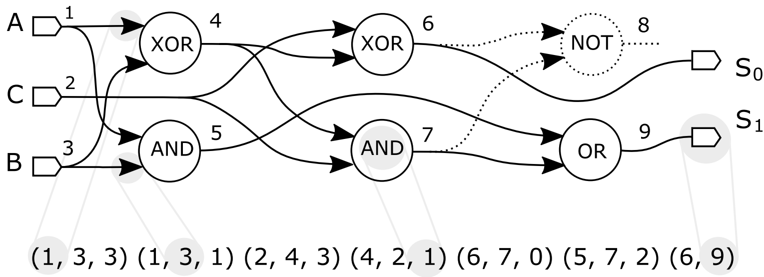

Fig. 2 Example of a CGP individual encoding a logic circuit (one-bit full adder) with $n_i = 3$ inputs and $n_o = 2$ outputs. The individual is encoded using an array of $n_n = 6$ two-input single-output nodes whose functions are chosen from a set of primitive functions $\Gamma = \{\mathrm{NOT}, \mathrm{AND},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

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 parameter L defines the level of connectivity and impacts the search space size. For example, if $L = 1$ only neighboring nodes may be connected; if $L = n_n$, full connectivity is enabled.

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. This means that the effective number of nodes is not fixed as many nodes may not be 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 example of the CGP encoding is shown in Fig. 2.

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

Author's personal copy

circuit optimization, the initial population is seeded by the original circuit. 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 [16].

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 [27, 28]. 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.

Author's personal copy

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

```
Input: A Boolean network N
Output: Optimized network N', cost(N') ≤ cost(N)
1 N' ← N
2 while terminated condition not satisfied do
3     W ← GetSubcircuit(N) ;
4     if W is a suitable candidate then
5         W' ← OptimizeNetworkUsingEA(W)
6         if cost((N' \ W) ∪ W') < cost(N') then
7             N' ← (N' \ W) ∪ W'
8 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. 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 proposed 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 Sect. 2.2.

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 ← identify the best candidate root node m ∈ N
2 C ← ReconvergenceDrivenCut(m, c_min, c_max, k_min, k_max)
3 W ← ExpandCutToWindow(m, C)
4 return W
```

Author's personal copy

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 Sect. 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 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 Fig. 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 Fig. 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, $w_{min} \leq |W| \leq w_{max}$

1 $m \leftarrow$ select a random node $m \in N$
2 init queue q with m
3 $W \leftarrow \emptyset$
4 **while** q *not empty* $\wedge$ $|W| < w_{max}$ **do**
5     $m \leftarrow$ pop a node from q
6     $W \leftarrow W \cup \{m\}$
7     $X \leftarrow fanin(m) \cup fanout(m)$
8   push all nodes from $X \setminus W$ that are not already in q into q
9 **if** $|W| < w_{min}$ **then**
10     $W \leftarrow \emptyset$
11 $W \leftarrow \bigcup_{m \in W} fanin(m)$
12 **return** W

Author's personal copy

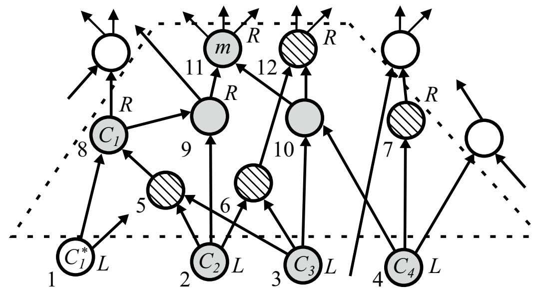

Fig. 3 Example of the window created using the cut-based algorithm GS1. The set of contained nodes of a 4-feasible cut C = $\{C_1, C_2, C_3, C_4\}$ rooted in node m is highlighted using the filled nodes. The hatched nodes are added to the window during the expansion of the cut. As a consequence of that, 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 $f_i \in$ {NOT,AND,OR,XOR, …} is the function of the node with index i

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 restrict 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 Fig. 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).

Author's personal copy

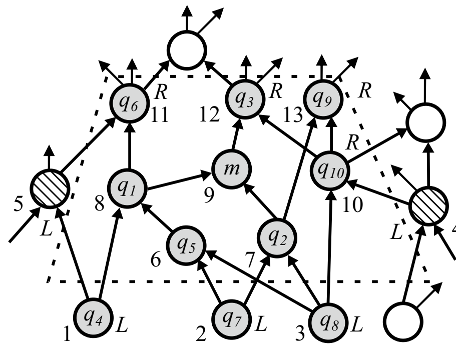

Fig. 4 Example of the window consisting of 10 nodes ($w_{max} = 10$) created using the proposed alternative windowing algorithm GS2. The neighboring nodes added into W are highlighted using the filled nodes. The hatched nodes are those added during the final step. The nodes at the bottom are primary 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 to 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 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

Author's personal copy

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 Fig. 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 (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.

The computation of the fitness score is implemented as suggested in [27]. The overall principle is illustrated in Fig. 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

Author's personal copy

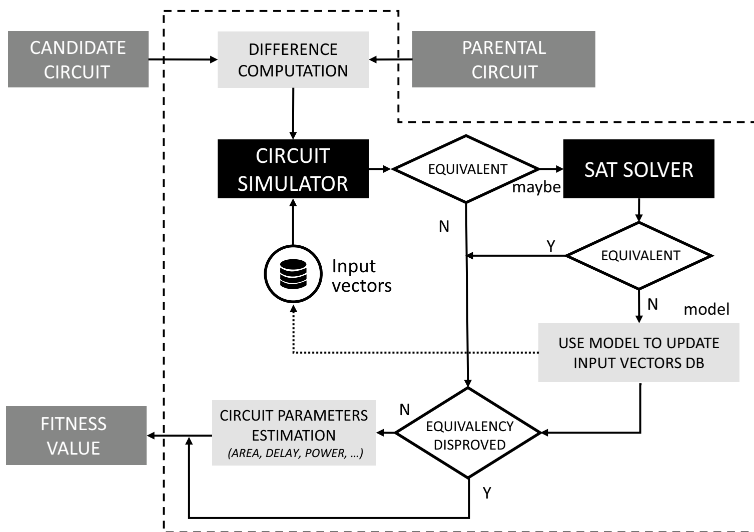

Fig. 5 Principle of the fitness score computation using the hybrid approach combining a circuit simulator with a SAT solver

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

Author's personal copy

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.[2] 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 Sects. 2.3 and 3.2. The CGP parameters were chosen in accordance with [27] and are summarized in Table 1. 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 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.

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

Author's personal copy

Table 1 The CGP parameters used in the experiments

	Parameter	Description
CGP encoding	n_n	A linear 1D array of $n_n = \|W\|$ CGP nodes is considered; n_n equals to the size of the optimized circuit
	n_a	each CGP node has $n_a = 2$ inputs and can implement one of eight predefined Boolean functions in $\Gamma = \{$BUF,
	Γ	NOT, AND, OR, XOR, NAND, NOR, XNOR$\}$
Mutation operator	h	Up to $h = 2$ active genes are modified
	L	Full connectivity is enabled, i.e. $L = n_n$
Search strategy	λ	$(1 + \lambda)$-ES is employed, where $\lambda = 1$

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[3].

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 10,000, 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 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 2. 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

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

Author's personal copy

Table 2 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

	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

The best results in each row are italics

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 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.

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 3. 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 3, the efficiency of

Author's personal copy

Table 3 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

	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

The best results in each row are italics

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 4. 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

Author's personal copy

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.

All the evolutionary approaches were able to further reduce the size of the benchmark circuits despite that the fact that they were highly optimized by the ABC synthesis tool. On 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 best results in each section of Table 4, 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 no 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 Fig. 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 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 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 5 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

Author's personal copy

Table 4 Comparison of the evolutionary methods (global and both proposed) against ABC

			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%*

Author's personal copy

Table 4 (continued)

			ABC		Average improvement			Best improvement		
Benchmark	PIs	POs	Gates	Delay	Global	GS1	GS2	Global	GS1	GS2
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%*

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‘. For each method, we report the average as well as best obtained improvement. The statistics is based on all five independent runs. The median is used to determine the average improvement

Author's personal copy

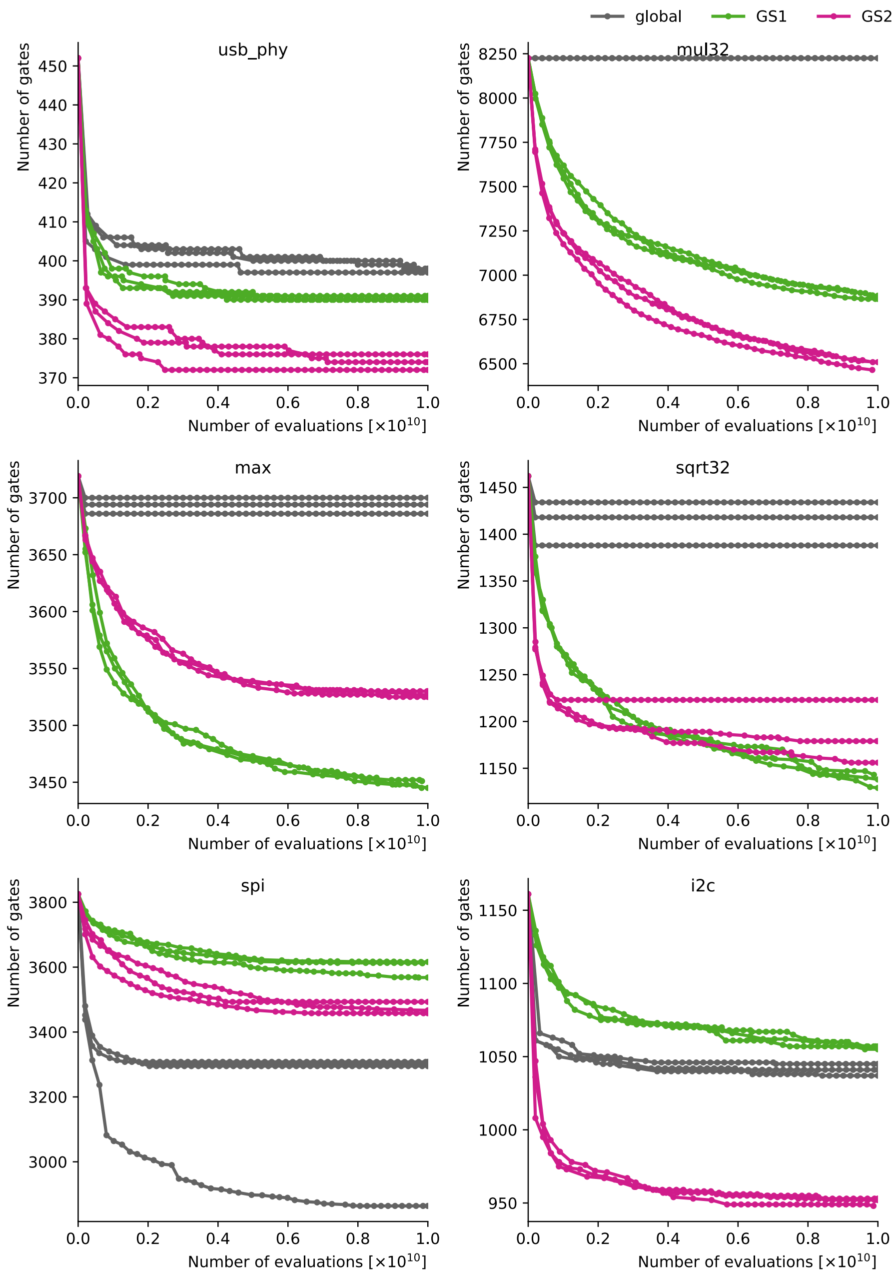

Fig. 6 The exemplary convergence curves representing the typical progress of the fitness score observed during the evolutionary optimization of digital circuits. Records from three independent evolutionary runs are shown in each figure. The lower number of gates, the better result. The data are downsampled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

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

Author's personal copy

Table 5 The average number of generated and evaluated candidate solutions needed to achieve 1%, 5%, and 10% 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}	$\mathbf{1.2 \times 10^{8}}$	–	–	–	–	–	–
aes_core	$\mathbf{2.1 \times 10^{8}}$	2.1×10^{9}	6.8×10^{8}	$\mathbf{1.6 \times 10^{9}}$	$> 10^{10}$	8.8×10^{9}	–	–	–
des_area	$\mathbf{5.3 \times 10^{7}}$	9.7×10^{8}	3.9×10^{8}	$> 10^{10}$	$\mathbf{7 \times 10^{9}}$	8.6×10^{9}	–	–	–
des_perf	$> 10^{10}$	3.4×10^{9}	$\mathbf{1 \times 10^{9}}$	$> 10^{10}$	$> 10^{10}$	$\mathbf{7.6 \times 10^{9}}$	–	–	–
dsp	$> 10^{10}$	8×10^{8}	$\mathbf{4.6 \times 10^{8}}$	–	–	–	–	–	–
ethernet	$> 10^{10}$	$> 10^{10}$	$\mathbf{2.4 \times 10^{9}}$	–	–	–	–	–	–
i2c	$\mathbf{3 \times 10^{5}}$	6.7×10^{7}	2.5×10^{6}	$\mathbf{8.6 \times 10^{6}}$	6.8×10^{8}	2.9×10^{7}	2.9×10^{9}	$> 10^{10}$	$\mathbf{1.6 \times 10^{8}}$
mem_ctrl	$\mathbf{1.4 \times 10^{4}}$	2.6×10^{8}	1.5×10^{8}	$\mathbf{8.1 \times 10^{4}}$	4.5×10^{9}	1.6×10^{9}	$\mathbf{2.4 \times 10^{5}}$	$> 10^{10}$	6.4×10^{9}
pci_bridge32	1.4×10^{9}	3.1×10^{8}	$\mathbf{2.3 \times 10^{8}}$	–	–	–	–	–	–
pci_spoci_ctrl	$\mathbf{1.2 \times 10^{4}}$	3×10^{7}	2.5×10^{6}	$\mathbf{1.7 \times 10^{5}}$	2.3×10^{8}	1.7×10^{7}	$\mathbf{7.3 \times 10^{5}}$	6.4×10^{8}	4.9×10^{7}
sasc	2.2×10^{7}	2.7×10^{7}	$\mathbf{1.8 \times 10^{7}}$	$> 10^{10}$	$\mathbf{8.6 \times 10^{8}}$	1.7×10^{9}	–	–	–
simple_spi	$\mathbf{6.6 \times 10^{6}}$	3.9×10^{7}	1.1×10^{7}	$> 10^{10}$	2.8×10^{9}	$\mathbf{7.6 \times 10^{8}}$	–	–	–
spi	$\mathbf{5.5 \times 10^{6}}$	1×10^{8}	5×10^{7}	$\mathbf{7.3 \times 10^{7}}$	3.9×10^{9}	9.4×10^{8}	$\mathbf{2.1 \times 10^{8}}$	$> 10^{10}$	$> 10^{10}$
ss_pcm	$\mathbf{9.3 \times 10^{6}}$	1.1×10^{8}	2.2×10^{7}	$> 10^{10}$	$\mathbf{2.1 \times 10^{9}}$	6.6×10^{9}	–	–	–
systemcaes	$> 10^{10}$	2.1×10^{8}	$\mathbf{1.1 \times 10^{8}}$	$> 10^{10}$	1.7×10^{9}	$\mathbf{1.1 \times 10^{9}}$	$> 10^{10}$	6.5×10^{9}	$\mathbf{3.5 \times 10^{9}}$
systemcdes	$\mathbf{5.8 \times 10^{6}}$	2.4×10^{8}	4.2×10^{7}	$\mathbf{5.9 \times 10^{7}}$	$> 10^{10}$	4.5×10^{8}	$\mathbf{1.7 \times 10^{9}}$	$> 10^{10}$	1.8×10^{9}
tv80	$\mathbf{4.2 \times 10^{4}}$	2.3×10^{8}	8.9×10^{7}	$\mathbf{1.9 \times 10^{7}}$	4.8×10^{9}	6.4×10^{8}	$\mathbf{2.2 \times 10^{8}}$	$> 10^{10}$	2.8×10^{9}
usb_funct	9.7×10^{7}	3×10^{8}	$\mathbf{7.5 \times 10^{7}}$	$> 10^{10}$	6.6×10^{9}	$\mathbf{8.4 \times 10^{8}}$	$> 10^{10}$	$> 10^{10}$	$\mathbf{5.1 \times 10^{9}}$
usb_phy	$\mathbf{6.2 \times 10^{4}}$	4.3×10^{6}	1.5×10^{6}	$\mathbf{2.2 \times 10^{6}}$	5.6×10^{7}	4×10^{6}	6.3×10^{8}	3.8×10^{8}	$\mathbf{3.8 \times 10^{7}}$
average	$\mathbf{5.4 \times 10^{6}}$	2.2×10^{8}	8.7×10^{7}	$\mathbf{7.5 \times 10^{6}}$	1.7×10^{9}	7.4×10^{8}	$\mathbf{2.1 \times 10^{8}}$	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}	$\mathbf{6.2 \times 10^{7}}$	$> 10^{10}$	1.7×10^{9}	$\mathbf{3.4 \times 10^{8}}$	$> 10^{10}$	6.8×10^{9}	$\mathbf{8.1 \times 10^{8}}$

Author's personal copy

Table 5 (continued)

Benchmark	1% improvement Global	GS1	GS2	5% improvement Global	GS1	GS2	10% improvement global	GS1	GS2
div16	$> 10^{10}$	8.5×10^{7}	$\mathbf{1.9 \times 10^{7}}$	$> 10^{10}$	6.3×10^{8}	$\mathbf{9.3 \times 10^{7}}$	$> 10^{10}$	2.6×10^{9}	$\mathbf{2.3 \times 10^{8}}$
hamming	$\mathbf{3.6 \times 10^{4}}$	1.9×10^{7}	5×10^{6}	$\mathbf{4 \times 10^{5}}$	1.8×10^{8}	2.5×10^{7}	$\mathbf{1.5 \times 10^{6}}$	6.1×10^{8}	4.8×10^{7}
mac32	$> 10^{10}$	$\mathbf{6.7 \times 10^{7}}$	7.2×10^{7}	$> 10^{10}$	$\mathbf{7 \times 10^{8}}$	8.5×10^{8}	$> 10^{10}$	$> 10^{10}$	$\mathbf{7.2 \times 10^{9}}$
max	$> 10^{10}$	1.1×10^{8}	$\mathbf{1 \times 10^{8}}$	$> 10^{10}$	$\mathbf{1.2 \times 10^{9}}$	6.2×10^{9}	–	–	–
mul32	$> 10^{10}$	7.7×10^{7}	$\mathbf{2.3 \times 10^{7}}$	$> 10^{10}$	4.9×10^{8}	$\mathbf{1.4 \times 10^{8}}$	$> 10^{10}$	1.6×10^{9}	$\mathbf{5.6 \times 10^{8}}$
mult64	$> 10^{10}$	3.7×10^{8}	$\mathbf{1.2 \times 10^{8}}$	$> 10^{10}$	6.8×10^{9}	$\mathbf{1.4 \times 10^{9}}$	–	–	–
revx	$> 10^{10}$	1×10^{8}	$\mathbf{2.3 \times 10^{7}}$	$> 10^{10}$	8.6×10^{8}	$\mathbf{1.4 \times 10^{8}}$	$> 10^{10}$	3.3×10^{9}	$\mathbf{3.8 \times 10^{8}}$
sqrt32	$\mathbf{3.4 \times 10^{5}}$	2.5×10^{7}	5×10^{6}	$\mathbf{3.8 \times 10^{6}}$	1.4×10^{8}	3×10^{7}	$> 10^{10}$	4.8×10^{8}	$\mathbf{9.1 \times 10^{7}}$
average	$\mathbf{4.9 \times 10^{4}}$	8.4×10^{7}	2.5×10^{7}	$\mathbf{4.3 \times 10^{5}}$	6.9×10^{8}	1.4×10^{8}	$\mathbf{1.5 \times 10^{6}}$	1.7×10^{9}	3.8×10^{8}
success rate	22%	*100%*	*100%*	22%	*100%*	*100%*	14%	85%	*100%*

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

Author's personal copy

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 5, we can easily identify that the global method converges faster compared to GS1 and GS2. On the other hand, the globally applied CGP has a tendency to get 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 no 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 5 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. The 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 sub-circuits. 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 '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 Fig. 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 'tv80'. The common feature is the steep convergence of the global method. The chosen 'spi' benchmark is however,

Author's personal copy

a bit exceptional, because we can observe how the global method can get stuck at a local optima. As evident also from Table 4, 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 got 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 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 6. 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 6, 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 occur 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 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.

Author's personal copy

Table 6 The average number of inputs, outputs and size of the sub-circuits produced by the implemented windowing algorithms

	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

Author's personal copy

Table 6 (continued)

Benchmark	Improved sub-circuits								Unchanged sub-circuits							
	avg PIs		avg POs		avg |W|		max |W|		avg PIs		avg POs		avg |W|		max |W|	
	GS1	GS2	GS1	GS2	GS1	GS2	GS1	GS2	GS1	GS2	GS1	GS2	GS1	GS2	GS1	GS2
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

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

Author's personal copy

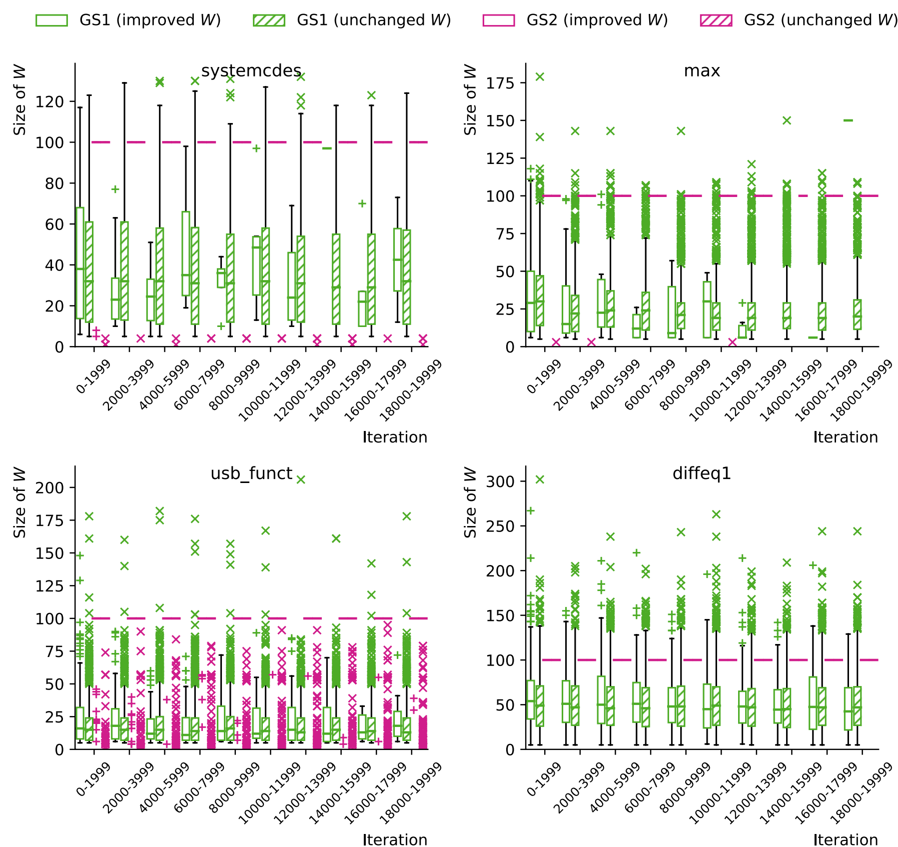

Fig. 7 Size of the sub-circuits extracted from the benchmark circuits in course of the optimization. Data from a single evolutionary run are plotted for each benchmark circuits. The boxes visualize distribution of $|W|$ for sub-circuits generated in 2000 consecutive iterations. Outliers are plotted as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

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 Sect. 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

Author's personal copy

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 this, 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 was significantly better than the cut-based alternative. On 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. R. Brayton, A. Mishchenko, ABC: An academic industrial-strength verification tool, in *Computer Aided Verification*, (Springer, Berlin, 2010), pp. 24–40
2. C.C.A. Coello, A.D. Christiansen, A.H. Aguirre, 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., 1997Springer, Vienna, 1998), pp. 333–336. https://doi.org/10.1007/978-3-7091-6492-1_73
3. P. Fiser, J. Schmidt, Small but nasty logic synthesis examples, in *Proceedings of 8th International Workshop on Boolean Problems*, pp. 183–190 (2008)
4. P. Fiser, J. Schmidt, Z. Vasicek, 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*, pp. 346–351 (2010)

Author's personal copy

5. B. Goldman, W. Punch, Reducing wasted evaluations in cartesian genetic programming. Lecture Notes in Computer Science **7831 LNCS**, 61–72 (2013). https://doi.org/10.1007/978-3-642-37207-0_6
6. B.W. Goldman, W.F. Punch, Analysis of cartesian genetic programmings evolutionary mechanisms. IEEE Trans. Evol. Comput. **19**(3), 359–373 (2015)
7. T.G.W. Gordon, P.J. Bentley, On evolvable hardware, in *Soft Computing in Industrial Electronics*, (Physica-Verlag, London, 2002), pp. 279–323.
8. P.C. Haddow, A. Tyrrell, Challenges of evolvable hardware: past, present and the path to a promising future. Genet. Program Evolvable Mach. **12**, 183–215 (2011)
9. T. Higuchi, T. Niwa, T. Tanaka, H. Iba, H. de Garis, T. Furuya, Evolving hardware with genetic learning: a first step towards building a darwin machine, in *Proceedings of the 2nd International Conference on Simulated Adaptive Behaviour*. (MIT Press, 1993), pp. 417–424
10. J. Kocnova, Z. Vasicek, Towards a scalable ea-based optimization of digital circuits, in *Genetic Programming 22nd European Conference, EuroGP 2019* (Springer International Publishing, 2019), pp. 81–97. https://doi.org/10.1007/978-3-030-16670-0_6
11. J.R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection* (MIT Press, Cambridge, MA, 1992)
12. N. Li, E. Dubrova, AIG rewriting using 5-input cuts, in *Proceedings of the 29th International Conference on Computer Design*, pp. 429–430. IEEE CS (2011)
13. J. Miller, P. Thomson, Cartesian genetic programming, in *Proceedings of the 3rd European Conference on Genetic Programming EuroGP2000, LNCS*, vol. 1802, pp. 121–132. Springer (2000)
14. J.F. Miller, Digital filter design at gate-level using evolutionary algorithms, in *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 1999* (Morgan Kaufmann, 1999), pp. 1127–1134
15. J.F. Miller, *Cartesian Genetic Programming* (Springer, 2011)
16. J.F. Miller, Cartesian genetic programming: its status and future. Genet. Program Evolvable Mach. (2019). https://doi.org/10.1007/s10710-019-09360-6
17. J.F. Miller, P. Thomson, T. Fogarty, Designing electronic circuits using evolutionary algorithms. arithmetic circuits: A case study, in *Genetic Algorithms and Evolution Strategies in Engineering and Computer Science*. (Wiley, 1997), pp. 105–131
18. A. Mishchenko, R. Brayton, Scalable logic synthesis using a simple circuit structure, in *International Workshop on Logic and Synthesis*, pp. 15–22 (2006)
19. A. Mishchenko, S. Chatterjee, R. Brayton, DAG-aware AIG rewriting: a fresh look at combinational logic synthesis, in *2006 43rd ACM/IEEE Design Automation Conference*, pp. 532–535 (2006). https://doi.org/10.1145/1146909.1147048
20. L. Sekanina, *Evolvable Components: From Theory to Hardware Implementations, Natural Computing Series* (Springer, New York, 2004)
21. L. Sekanina, O. Ptak, Z. Vasicek, Cartesian genetic programming as local optimizer of logic networks, in *2014 IEEE Congress on Evolutionary Computation*. (IEEE CIS, 2014), pp. 2901–2908
22. A.P. Shanthi, R. Parthasarathi, Practical and scalable evolution of digital circuits. Appl. Soft Comput. **9**(2), 618–624 (2009)
23. E. Stomeo, T. Kalganova, C. Lambert, Generalized disjunction decomposition for evolvable hardware. IEEE Trans. Syst. Man Cybern. B **36**(5), 1024–1043 (2006)
24. E. Stomeo, T. Kalganova, C. Lambert, 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. Y. Tao, L. Zhang, Y. Zhang, A projection-based decomposition for the scalability of evolvable hardware. Soft. Comput. **20**(6), 2205–2218 (2016). https://doi.org/10.1007/s00500-015-1636-2
26. A. Thompson, Silicon evolution, in *Proceedings of the First Annual Conference on Genetic Programming, GECCO '96* (MIT Press, Cambridge, 1996), pp. 444–452
27. 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, LCNS 9025* (Springer International Publishing, 2015), pp. 139–150
28. Z. Vasicek, L. Sekanina, Formal verification of candidate solutions for post-synthesis evolutionary optimization in evolvable hardware. Genet. Program Evolvable Mach. **12**(3), 305–327 (2011)
29. V. Vassilev, D. Job, J.F. Miller, 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)

Author's personal copy

30. C. Wolf, J. Glaser, J. Kepler, Yosys-a free verilog synthesis suite, in *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)* (2013)
31. S. Zhao, L. Jiao, Multi-objective evolutionary design and knowledge discovery of logic circuits based on an adaptive genetic algorithm. Genet. Program Evolvable Mach. **7**(3), 195–210 (2006)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Affiliations

Jitka Kocnova[1] · Zdenek Vasicek[1]

Jitka Kocnova
ikocnova@fit.vutbr.cz

[1] Faculty of Information Technology, Brno University of Technology, IT4Innovations Centre of Excellence, Brno, Czech Republic