Towards a Scalable EA-Based Optimization of Digital Circuits

Jitka Kocnova(✉) and Zdenek Vasicek

Faculty of Information Technology, IT4Innovations Centre of Excellence, Brno University of Technology, Brno, Czech Republic
{ikocnova,vasicek}@fit.vutbr.cz

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

Keywords: Cartesian Genetic Programming · Resynthesis · Logic optimization

1 Introduction

Logic synthesis, as understood by the hardware community, is a process that transforms a high-level description into a gate-level or transistor-level implementation. Due to the complexity of the problem, the synthesis process is typically broken into a sequence of steps. Among others, logic optimization represents an important part of the whole process. The goal of the logic optimization is to transform a suboptimal solution into an optimal gate-level implementation with respect to given synthesis goals. Due to the scalability issues, the problem is typically represented using a suitable internal representation. Current state-of-the-art logic synthesis tools, such as ABC, represent circuits using a directed acyclic graph composed of two-input AND nodes connected by direct or negated edges denoted as and-inverter graph (AIG). The optimization of AIGs is based

© Springer Nature Switzerland AG 2019
L. Sekanina et al. (Eds.): EuroGP 2019, LNCS 11451, pp. 81–97, 2019.
https://doi.org/10.1007/978-3-030-16670-0_6

on *rewriting*, a greedy algorithm which minimizes size of AIG by iteratively selecting subgraphs rooted at a node and replacing them with smaller precomputed subgraphs, while preserving the functionality of the root node [1]. AIG rewriting is local, however, the scope of changes becomes global by application of rewriting many times. In addition to that, *resubstitution* and *refactoring* can be employed. Resubstitution expresses the function of a node using other nodes present in the AIG [2]. Refactoring iteratively selects large cones of logic rooted at a node and tries to replace them with a more efficient implementation [1]. Refactoring can be seen as a variant of rewriting. The main difference is that rewriting selects subgraphs containing few leaves because the number of leaves determines the number of variables of a Boolean function whose optimal implementation is sought.

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

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

In order to improve the results of EA-based synthesis, we propose to combine the EA-based approach with refactoring while following the principle of local resynthesis applied in common logic synthesis tools. Firstly, a logic circuit is optimized by means of a common synthesis approach. Then, the optimized circuit is mapped to standard gates and optimized using the proposed method that extracts a relatively small sub-circuits that are subsequently optimized by Cartesian Genetic Programming (CGP). The original sub-circuit is then replaced by

its optimized variant provided that there is an improvement at the global level and the whole process is repeated. Our approach is based on iterative optimization of large portions of the original circuit. Compared to rewriting, we do not impose any limitation on the number of leaves because the larger subgraphs offer more opportunities for potential area improvement.

2 Background

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

2.1 Boolean Networks

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

2.2 Limiting the Scope of Boolean Networks

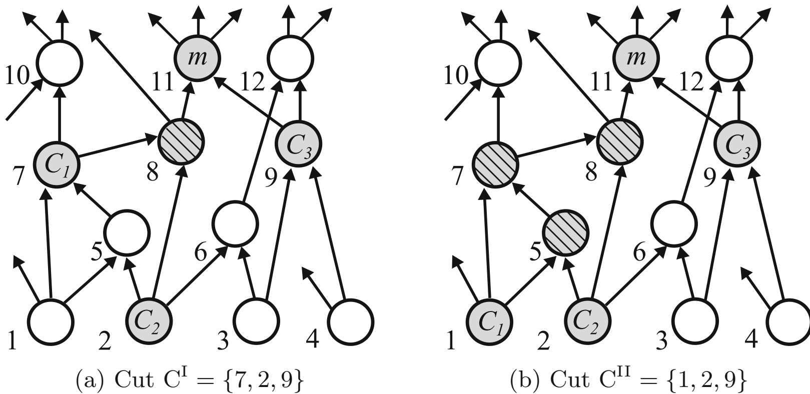
Network scoping represents a key operation to ensure a good scalability of synthesis tools when working with large Boolean networks. In addition, it forms an integral part of rewriting as well as refactoring. Two approaches have been proposed to limit the scope of logic synthesis to work only on a small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

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

A complementary approach based on computing so called k-feasible cuts is usually preferred to avoid determining the required number of logic levels. A cut

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

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



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

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

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

2.3 Evolutionary Synthesis of Logic Circuits

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

Since its introduction, CGP remains the most powerful evolutionary technique in the domain of logic synthesis and optimization [9]. In this area, a linear form of CGP is preferred today. CGP models a candidate circuit having n_i PIs and n_o POs as a linear 1D array of n_n configurable nodes. Each node has n_a inputs and corresponds with a single gate with up to n_a inputs. The inputs can be connected either to the output of a node placed in the previous L columns or directly to PIs. This avoids a feedback. The function of a node can be chosen

from a set of n_f functions. Depending on the function of a node, some of its inputs may become redundant. In addition to that, the fixed number of nodes n_n does not mean that all the nodes contribute to the POs. These key features allow redundancy and flexibility of CGP.

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

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

3 The Proposed Method

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

Firstly a node which may potentially lead to the best improvement of N is determined. Since the identification of this node itself is a nontrivial problem, some heuristic needs to be implemented. The size of transitive fan-in cone, level of the node or a more complex information can be used to determine the most suitable candidate. Then, a working area (window) is extracted from the Boolean network. This procedure starts with computation of the reconvergence-driven cut C as described in 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 ($c_{min} < c_{max}$), and k_{min} and k_{max} ($k_{min} \leq |C| \leq k_{max}$) limiting the size of cut (feasibility).

This step is followed by expansion of the cut C into a window W, i.e. expansion of the set of leaf nodes to a set of contained nodes. In addition to the nodes inside the cut, we should consider also all nodes that are not contained in the cut but have fanins inside the cut. Our expansion is similar to that employed in the resubstitution [2] where transitive fanout of C is considered, however, we do not impose any limit on the number of included nodes or their maximum level. The process of cut identification and the subsequent expansion is illustrated in Fig. 2.

Algorithm 1. 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      m ← identify the best candidate root node m ∈ N'
4      C ← ReconvergenceDrivenCut(m)
5      W ← ExpandCutToWindow(m, C)
6      if W is not a suitable candidate then
7          continue
8      W' ← OptimizeNetworkUsingEA(W)
9      if cost((N' \ W) ∪ W') < cost(N') then
10         N' ← (N' \ W) ∪ W'
11 return N'
```

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

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

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

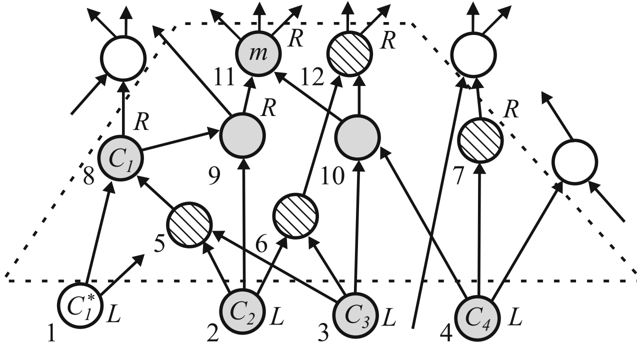


Fig. 2. Example of the window created using the proposed algorithm. The set of contained nodes of a 4-feasible cut C = $\{C_1, C_2, C_3, C_4\}$ rooted in node m is highlighted using the filled nodes. The hatched nodes are added to the window during the expansion of the cut. As a consequence of that,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

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

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

4 Experimental Evaluation

4.1 Experimental Setup

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

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

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

Approach	k_{min}	k_{max}	c_{min}	c_{max}	Scoping method
Rewriting [2,15]	–	4	–	–	Cut computation
Redundancy removal [2]	6	12	–	–	Windowing
Conventional resynthesis [16]	–	–	–	3360 (30%)	Windowing (various)
Evolutionary resynthesis [3]	1	10	8	50	Radius-based windowing
Proposed approach	–	–	10	10^4	Cut-based windowing

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

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

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

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

4.2 Experimental Results

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

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

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

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

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

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

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

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

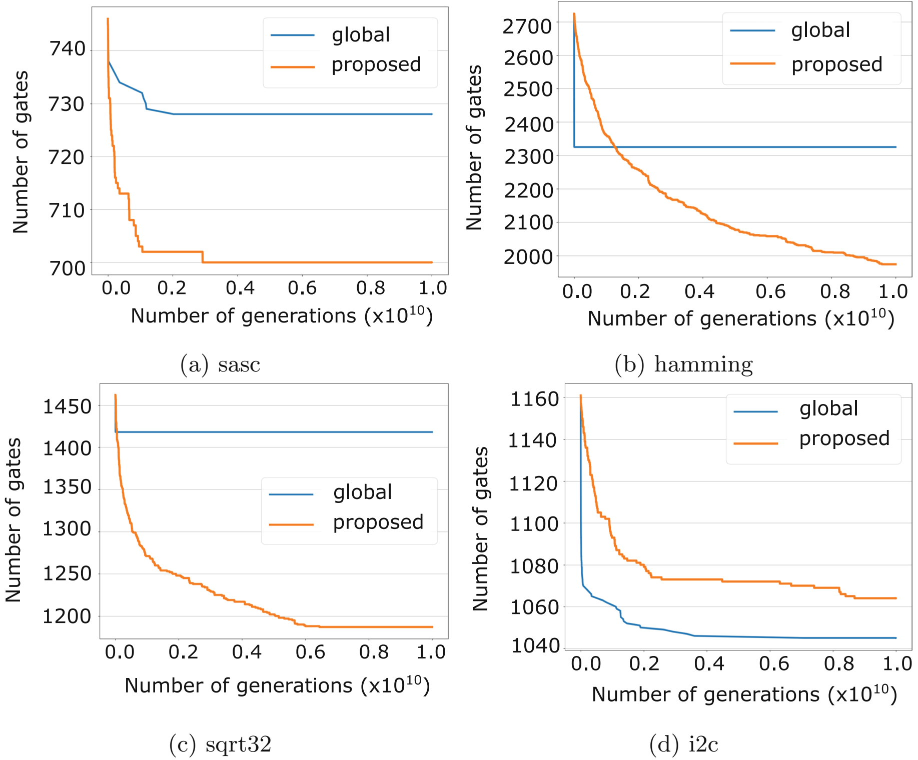


Fig. 3. Typical convergence curves for four chosen benchmark circuits. The lower value (number of gates) the better result.

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

We assume that the slow convergence is caused by the fact that each subcircuit produced by the proposed windowing algorithm is optimized for a fixed number of generations independently on its parameters (the number of gates,

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

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

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

We analyzed all the evolutionary runs across all benchmarks circuits and determined the maximum number of generations that caused a reduction of a sub-circuit. For each run of CGP we recorded the last generation that caused a change in the number of gates together with the volume of the optimized sub-circuit. The obtained numbers are plotted as a function of sub-circuit volume using a boxplot in Fig. 4. As expected, the more nodes are there in the sub-circuit the more CGP generations are typically used to optimize it. We can also see that the dependence between these two values is exponential – this is illustrated also by the blue curve representing polynomial interpolation of the median value. As the volume of the window increases, the number of occurrences

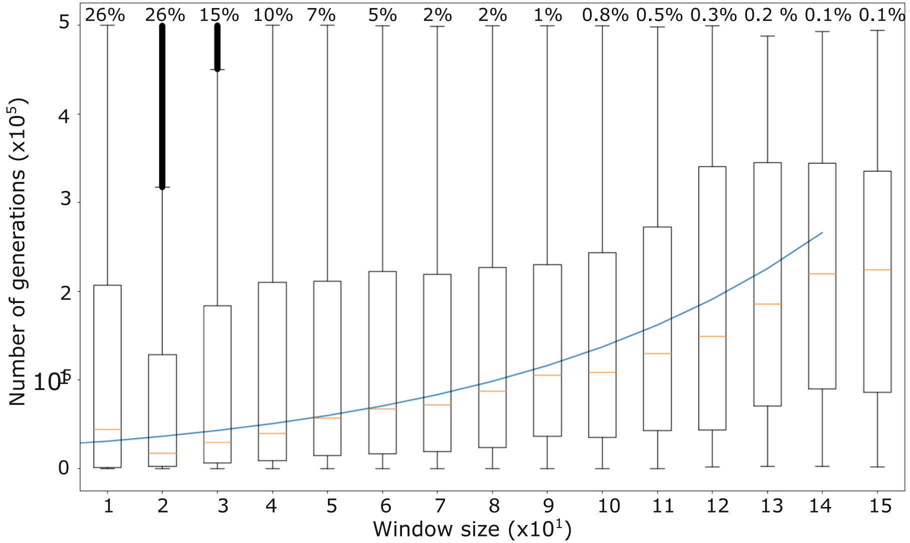


Fig. 4. Boxplots showing the number of generations that caused removal of a gate. The numbers above each boxplot show the number of occurrences of the window of a certain volume.

of such cases decreases (see the numbers above each boxplot showing how many times we seen a window having volume between X and X + 10). Usually, small windows are produced. Windows up to 45 nodes were produced in more than 77% cases. Due to this fact, the interpolation is limited to 150 nodes because there is insufficient number of results for the bigger windows.

5 Conclusion

Compared to the conventional logic synthesis, state-of-the-art EA-based optimization is able to produce substantially better results but at the cost of a higher run time. Unfortunately, the run time increases with the increasing complexity of the Boolean networks. This paper addresses this problem by combining the EA-based optimization with windowing that allows to work on a smaller portions of the original Boolean network. Even though we used a very simple strategy of root node selection which may degrade the capabilities of the resynthesis, the proposed method is able to outperform the original EA-based optimization applied to the whole Boolean networks. The number of nodes w.r.t the original method was improved by 9.2% on average. Even though only area was analyzed in this study, the depth of the optimized circuits is comparable with the original circuits.

In our future work, we would like to implement an adaptive strategy that modifies the maximum number of generations according to the size of the optimized logic circuit. In addition to that, we would like to focus on improvement

of root node selection strategy. The question here is whether the result would be better if the cut is built from a node near to the previously chosen one.

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

References

1. Mishchenko, A., Chatterjee, S., Brayton, R.: DAG-aware AIG rewriting: a fresh look at combinational logic synthesis. In: 2006 43rd ACM/IEEE Design Automation Conference, pp. 532–535, July 2006
2. Mishchenko, A., Brayton, R.: Scalable logic synthesis using a simple circuit structure. In: International Workshop on Logic and Synthesis, pp. 15–22 (2006)
3. Sekanina, L., Ptak, O., Vasicek, Z.: Cartesian genetic programming as local optimizer of logic networks. In: 2014 IEEE Congress on Evolutionary Computation, pp. 2901–2908. IEEE CIS (2014)
4. Fiser, P., Schmidt, J., Vasicek, Z., Sekanina, L.: On logic synthesis of conventionally hard to synthesize circuits using genetic programming. In: 13th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems, pp. 346–351 (2010)
5. Fiser, P., Schmidt, J.: Small but nasty logic synthesis examples. In: Proceedings of the 8th International Workshop on Boolean Problems, pp. 183–190 (2008)
6. Vasicek, Z.: Cartesian GP in optimization of combinational circuits with hundreds of inputs and thousands of gates. In: Machado, P., et al. (eds.) EuroGP 2015. LNCS, vol. 9025, pp. 139–150. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-16501-1_12
7. Li, N., Dubrova, E.: AIG rewriting using 5-input cuts. In: Proceedings of the 29th International Conference on Computer Design, pp. 429–430. IEEE CS (2011)
8. Lohn, J.D., Hornby, G.S.: Evolvable hardware: using evolutionary computation to design and optimize hardware systems. IEEE Comput. Intell. Mag. **1**(1), 19–27 (2006)
9. Miller, J.F., Thomson, P.: Cartesian genetic programming. In: Poli, R., Banzhaf, W., Langdon, W.B., Miller, J., Nordin, P., Fogarty, T.C. (eds.) EuroGP 2000. LNCS, vol. 1802, pp. 121–132. Springer, Heidelberg (2000). https://doi.org/10.1007/978-3-540-46239-2_9
10. Miller, J.F.: Cartesian Genetic Programming. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-17310-3
11. Vassilev, V., Job, D., Miller, J.F.: Towards the automatic design of more efficient digital circuits. In: Lohn, J., Stoica, A., Keymeulen, D., Colombano, S. (eds.) Proceedings of the 2nd NASA/DoD Workshop on Evolvable Hardware, pp. 151–160. IEEE Computer Society, Los Alamitos (2000)
12. Tao, Y., Zhang, L., Zhang, Y.: A projection-based decomposition for the scalability of evolvable hardware. Soft Comput. **20**(6), 2205–2218 (2016)
13. Stomeo, E., Kalganova, T., Lambert, C.: Generalized disjunction decomposition for the evolution of programmable logic array structures. In: First NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2006), pp. 179–185 (2006)
14. Vasicek, Z., Sekanina, L.: Formal verification of candidate solutions for post-synthesis evolutionary optimization in evolvable hardware. Genet. Program. Evolvable Mach. **12**(3), 305–327 (2011)

15. Fiser, P., Halecek, I., Schmidt, J.: Are XORs in logic synthesis really necessary? In: IEEE 20th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS), pp. 138–143 (2017)
16. Fiser, P., Schmidt, J.: It is better to run iterative resynthesis on parts of the circuit. In: Proceedings of the 19th International Workshop on Logic and Synthesis, pp. 17–24. University of California Irvine (2010)
17. Wolf, C., Glaser, J., Kepler, J.: Yosys-a free Verilog synthesis suite. In: Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip) (2013)
18. Brayton, R., Mishchenko, A.: ABC: an academic industrial-strength verification tool. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 24–40. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14295-6_5