An Enhanced Resubstitution Algorithm for Area-Oriented Logic Optimization

Andrea Costamagna
EPFL
Lausanne, Switzerland

Alan Mishchenko
U.C.Berkley
Berkley, California

Satrajit Chatterjee
Kepler AI
Palo Alto, California

Giovanni De Micheli
EPFL
Lausanne, Switzerland

***Abstract*—*Logic synthesis* is an ensemble of algorithms that optimizes digital circuit representations and maps them to a chosen technology. Minimizing the number of gates is essential to reduce area occupation and power consumption. As the problem is intractable, heuristic logic transformations are used. In particular, *resubstitution* attempts to express the function of a node using other nodes already present in the network. State-of-the-art *resubstitution* engines can only identify new implementations with support of up to three inputs or being simply decomposable. This work aims at extending *resubstitution* to non-decomposable functions with more than three inputs and it outperforms previous methods. We apply our method on highly optimized designs from the ISCAS and EPFL benchmarks, achieving additional average improvements of 18.50% and 8.36%.**

I. Introduction

THE operation of digital chips relies on switching millions of gates within the combinational circuits they contain. Each of these circuits is one of many possible implementations of complex Boolean functions. Finding implementations minimizing the gate count is an effective approach to obtain smaller chip sizes, fewer wires, and reduced power consumption.

Logic synthesis is an ensemble of algorithms that optimizes network representations before mapping to a specific technology library [1]–[3]. Reducing the number of nodes in the network representation is crucial to reduce the number of gates after mapping. As the problem is intractable [4], various logic heuristic transformations are used.

Resubstitution is a very effective transformation that tries to express (*resynthesize*) the function of a node using a set of candidate nodes already present in the network [1]. The transformation is accepted if the new implementation reduces the node count. The exploited nodes are named *divisors* because resynthesis generally occurs through a decomposition presenting the mathematical structure of a *division* [5]–[7].

There are two key sub-problems in *resubstitution*:

1) Select a support from a set of candidate divisors.
2) Resynthesize the function using this support.

State-of-the-art resubstitution engines adress these problems by combining functional simulation, decomposition and SAT-solving [5], [8]–[10]. These engines are highly effective when:

1) The number of support divisors is 3 or less.
2) The functionality of the node is simply decomposable.

This work extends *resubstitution* to functions with more than three inputs and that are not simply decomposable. Our method relies on *set of pairs of functions to be distinguished* (SPFD), a powerful functional representation for investigating the dependency of different nodes [8], [11]–[13]. In this work:

1) We define an SPFD-based cost-function $\mathcal{H}(x_i)$ for evaluating whether a divisor x_i should be selected.
2) We devise a support selection algorithm based on $\mathcal{H}(\cdot)$.
3) We devise a resynthesis algorithm based on $\mathcal{H}(\cdot)$.
4) We use these algorithms in a *resubstitution* engine.

The experimental results show that our approach enables *resubstitution* to identify optimization opportunities that are impossible to find with state-of-the-art engines. We apply our method on designs for which state-of-the-art *resubstitution* [10] could not find any further optimization. Our heuristic unlocks additional average improvements of $\mathbf{18.50\%}$ and $\mathbf{8.36\%}$ on the ISCAS and EPFL benchmarks.

II. Background

A. Boolean Network Representations

*Boolean network*s are directed acyclic graphs in which nodes represent logic gates, and edges represent wires. *And-Inverter Graphs* (AIGs) are Boolean networks in which nodes are two-input ANDs, and negated edges represent inverters. XAIGs extend AIGs with two-input XORs, enabling design-dependent optimizations not achievable with AIGs [14]. As state-of-the-art resubstitution heuristics rely on unateness information, they struggle with XOR functionalities [5]. Hence, investigating XAIGs is useful to test the effectiveness of a novel resubstitution algorithm to identify XOR-dependent optimizations. Due to space limitations, we focus on XAIGs, but our method generalizes to other representations.

If there is a path from a node x_i to a node x, x_i is in the *transitive fanin* (TFI) of x. The primary inputs (PIs) are nodes without fanins in the network. The *maximum fanout free cone* (MFFC) of a node x is the set of nodes that are x, or a node in the TFI of x whose fanout is uniquely in the MFFC.

A *cut* of a node x, is a set of nodes, named *leaves* such that any path from a PI to x passes through one leaf. The *local function* of node x with respect to a cut is the Boolean function of the subcircuit identified by the cut. The *global function* of node x is the Boolean function of the sub-circuit defined from node x to the PIs. With an abuse of notation we refer to the *global function* of x as $x : \mathbb{B}^n \mapsto \mathbb{B}$.

B. Sets of Pairs of Functions to be Distinguished

Let $x : \mathbb{B}^n \mapsto \mathbb{B}$ be the *global function* of a node. A *minterm* $M \in \mathbb{B}^n$ is a possible inputs assignment. Then, x induces a

This research was supported in part by Synopsys inc., in part by SRC Contract 3173.001 "Standardizing Boolean transforms to improve quality and runtime of CAD tools".

partition of $\mathbb{B}^n$ in two subsets: the *onset* and the *offset*, such that $x(M)=1$ and $x(M)=0$, respectively.

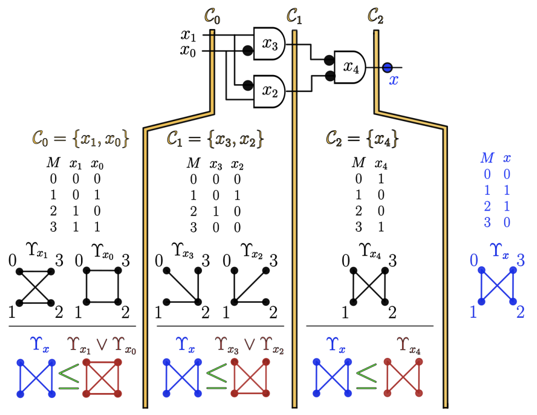
The *global sets of pairs of functions to be distinguished* (gSPFD) at node x is a function $\Upsilon_x : \mathbb{B}^n \times \mathbb{B}^n \to \mathbb{B}$:

$$\Upsilon_x(M_i, M_j) \doteq x(M_i) \oplus x(M_j) \qquad \forall M_i, M_j \in \mathbb{B}^n \qquad (1)$$

This function verifies if x *distinguishes* M_i and M_j because one is in the *onset* and the other is in the *offset* [8], [11], [12].

When exhaustively simulating a network is intractable, it is customary to select a subset of the input minterms $\mathcal{M} \subseteq \mathbb{B}^n$, perform the functional simulation of the network, and treat the resulting partial truth tables $\tilde{x}$, named *simulation signatures*, as an approximation of the *global functions*. The *approximate SPFD* (aSPFD) $\Upsilon_{\tilde{x}}$ is obtained by evaluating Eq. 1 on $\tilde{x}$ [15].

An *Information graph* (IG) is the graph representation of an SPFD where each vertex is a minterm, and there is an edge $\{M_i, M_j\}$ if $x(M_i) \neq x(M_j)$, i.e., if Eq. 1 evaluates to 1 [16]. Fig. 1 shows the IGs of the nodes in a simple network.



M	x_1	x_0
0	0	0
1	0	1
2	1	0
3	1	1

M	x_3	x_2
0	0	0
1	0	1
2	1	0
3	0	0

M	x_4
0	1
1	0
2	0
3	1

M	x
0	0
1	1
2	1
3	0

Fig. 1: From top to bottom: global functions at the cut nodes, their IGs, and the covering check of the target IG at each cut.

C. Dependency and Graph Covering

A *dependency function* for node x and set $\mathcal{C} = \{x_{c(l)}\}_{l=1}^{k}$, is a function $g : \mathbb{B}^k \to \mathbb{B}$ s.t. $x = g(\mathcal{C})$ [17]. Given a node x and a set of variables $\mathcal{C}$, a necessary and sufficient condition for the existence of a *dependency function* is that, $\forall M_i, M_j \in \mathbb{B}^n$,

$$x(M_i) \neq x(M_j) \Rightarrow \exists x_l \in \mathcal{C} \text{ s.t. } x_l(M_i) \neq x_l(M_j). \qquad (2)$$

When $\mathcal{C}$ is a *cut* of a node x, the *local function* constitutes a *dependency function*, and Eq. 2 is necessarily satisfied.

Using the definition of SPFDs (Eq. 1), we rewrite Eq. 2 as

$$\Upsilon_x(M_i, M_j) \leq \bigvee_{l=1}^{k} \Upsilon_{x_{c(l)}}(M_i, M_j) \qquad (3)$$

Where $a \leq b$ (or $a \Rightarrow b$ [4]) is 0 only if $a=1$ and $b=0$. Eq. 3 identifies a covering condition on the IGs: the IG of node x is covered by the union of the IGs of the nodes in $\mathcal{C}$ for which there is a *dependency function*. Fig. 1 illustrates the satisfaction of Eq. 3 at the cuts $\{x_0, x_1\}$, $\{x_2, x_3\}$, and $\{x_4\}$: the IG of x is covered by the union of the IGs at each cut.

D. Resubstitution and Related Works Based on SPFDs

Boolean resubstitution (RS) is a logic minimization heuristic that can optimize Boolean networks in which:

1) There are nodes with constant global function.
2) There are nodes that can be resynthesized with a new set of variables reducing the node count of the network.

An important sub-problem to address in RS is the *support selection* problem, i.e., finding a new set of nodes to resynthesize a target node. Since a valid support must satisfy Eq. 3, several RS methods can be understood using this equation.

Originally, gSPFDs were computed exhaustively [18]. More recently, Zhang et al. [8] combined partial simulation and SAT-solving to reconstruct gSPFDs and identified valid supports from a set of candidates. Reconstructing the gSPFD with SAT-solving makes the method computationally intensive [15].

In logic rectification, Yang et al. [15] addressed this problem using aSPFDs. Nevertheless, their method relies on expensive heuristics that iterate through all the edges of the aSPFDs.

Goldberg et al. [19] proposed an RS-like algorithm for subcircuit replacement. However, they only use the initial subcircuit's inputs, restricting the method to local optimizations.

E. The Simulation-Guided Paradigm For Resubstitution

Algorithm 1 outlines the principle of operation of modern resubstitution [5], [8]–[10]. First, the network is simulated with $|\mathcal{M}|$ simulation patterns. Next, for each node x, a structural exploration of the network identifies a list of divisors $\mathcal{D}$. If node x can be resynthesized with some divisors from $\mathcal{D}$, and the transformation reduces the node count, a SAT-solver checks if the transformation preserves functional equivalence. This paper focuses on the research problem of resynthesizing a node given a set of divisors, i.e., the core of Algorithm 1.

State-of-the-art resynthesis engines for XAIGs enumerate optimum XAIGs with up to 3 inputs, and verify if evaluating any of them at a subset of the divisors yields the simulation pattern of node x [10]. Unateness-based heuristics improve runtime and quality by reducing the number of cases considered, and identifying if a function is *simply decomposable*, i.e., $f(\mathcal{S}) = x_i \odot f(\mathcal{S} \backslash x_i)$ for some Boolean function $\odot$ [20]. However, these heuristics can result in missing optimization opportunities, especially when non-unate functions like XORs are involved. We name Algorithm 1 with this engine `urs`.

In this paper, we propose a resynthesis algorithm capable of handling arbitrary functions with more than 3 inputs. We address the problem in two steps: *support selection* and *resynthesis*. We name Algorithm 1 with our method `irs`.

III. SPFD-Based Resubstitution

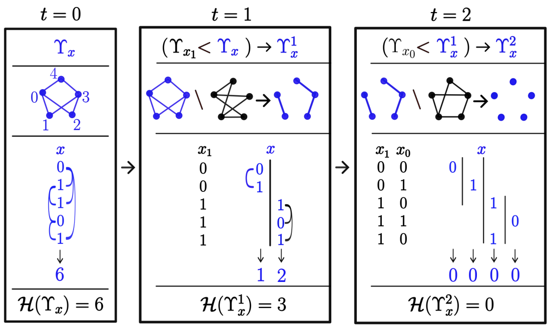
A. Covering Processes and SPFD Representations

Given a node x and an ordered set of nodes $\mathcal{C} = (x_{c(t)})_{t=1}^{T}$, a *covering process* is the sequence $\Upsilon_x^0 \to \Upsilon_x^1 \ldots \Upsilon_x^t \ldots \to \Upsilon_x^T$, where $\Upsilon_x^0 = \Upsilon_x$, and at each step t we remove the edges covered by $\Upsilon_{x_{c(t)}}$. Fig 2 shows an example where we use simulation signatures of length $|\mathcal{M}| = 5$. We name $\mathcal{H}(\Upsilon_x^t)$ the number of edges of Υ_x^t. To implement this process, we need an efficient representation for SPFDs.

Using the truth table of the function in Eq. 1 allows for a straightforward implementation, because covering is a bitwise

Algorithm 1 $\langle\texttt{METHOD}\rangle\texttt{rs}(\mathcal{N}; |\mathcal{M}|, N_{max})$

```
1: random_simulation($\mathcal{N}, |\mathcal{M}|$)
2: for all $x \in \mathcal{N}$ do
3:   while iteration $< N_{max}$ do
4:     $\mathcal{D} \leftarrow$ collect_divisors()
5:     $x_{new} \leftarrow$ resynthesize$\langle$METHOD$\rangle(x, \mathcal{D})$
6:     if #(removed nodes) > #(new nodes) then
7:       if SAT-CEC($x_{new}, x$) == true then
8:         return the subcircuit to perform optimization
9:       else
10:        update_simulation ($\mathcal{N}$)
```



t = 0
t = 1
t = 2
Υ_x
$(\Upsilon_{x_1} < \Upsilon_x) \to \Upsilon_x^1$
$(\Upsilon_{x_0} < \Upsilon_x^1) \to \Upsilon_x^2$
x
x_1
x_1 x_0
$\mathcal{H}(\Upsilon_x) = 6$
$\mathcal{H}(\Upsilon_x^1) = 3$
$\mathcal{H}(\Upsilon_x^2) = 0$

Fig. 2: From top to bottom: covering using IGs and simulation signatures, and number of edges $\mathcal{H}(\Upsilon_x^t)$ after covering.

operation $\Upsilon_x^t = \Upsilon_{x_{c(t)}} < \Upsilon_x^{t-1}$, and counting the edges is a popcount operation $\mathcal{H}(\Upsilon_x^t) = \frac{\texttt{popcount}(\Upsilon_x^t)}{2}$. However, the memory cost of this representation is quadratic in the number of minterms $\mathcal{O}(|\mathcal{M}|^2)$, limiting the size of the simulation signatures and, consequently, the expressiveness of aSPFDs.

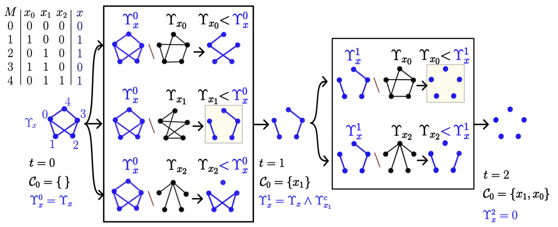
We propose an alternative covering approach using the $|\mathcal{M}|$-dimensional simulation signatures. Covering the graph with $\Upsilon_{x_{c(t)}}$ corresponds to exploiting the information of $x_{c(t)}$, i.e., to separate the graph in two disconnected parts identified by $x_{c(t)}$. This amounts to partitioning the signature of x using the *onset* and *offset* of $x_{c(t)}$. The bottom part of Fig. 2 shows the partitioning of the signature. At each step, $\mathcal{H}(\Upsilon_x^t)$ is the sum of the products of the number of 0s and 1s in each part. This representation requires a $\mathcal{O}(2^T|\mathcal{M}|)$-memory occupation.

The choice of a representation depends on T and $|\mathcal{M}|$.

B. The Support Selection Algorithm

Let x be a target node and $\mathcal{D}=\{x_{d(l)}\}_{l=1}^{D}$ a set of candidate divisors. The *support selection problem* consists of finding a subset $\mathcal{C}\subset\mathcal{D}$ satisfying Eq. 3. We define two algorithms, both relying on the definition of a *covering process*: at each step, we choose a divisor, we update the support, and we cover Υ_x^t. We aim for sets $\mathcal{C}$ satisfying Eq. 3 and having small sizes: small supports contain more informative variables, which are more likely to result in compact resynthesis sub-circuits.

The first approach we consider is a greedy strategy [21]. Let Υ_x^t be the partially covered IG at iteration t, and $\mathcal{H}(\Upsilon_{x_i} < \Upsilon_x^t)$ the number of remaining edges after covering Υ_x^t with Υ_{x_i}. In *greedy support selection* we choose the divisor x_i minimizing $\mathcal{H}(\Upsilon_{x_i} < \Upsilon_x^t)$, and we break ties at random.



M x_0 x_1 x_2 x
Υ_x
t = 0
$\mathcal{C}_0 = \{\}$
$\Upsilon_x^0 = \Upsilon_x$
Υ_x^0
Υ_{x_0}
$\Upsilon_{x_0} < \Upsilon_x^0$
Υ_{x_1}
$\Upsilon_{x_1} < \Upsilon_x^0$
Υ_{x_2}
$\Upsilon_{x_2} < \Upsilon_x^0$
t = 1
$\mathcal{C}_0 = \{x_1\}$
$\Upsilon_x^1 = \Upsilon_x \wedge \Upsilon_{x_1}^c$
Υ_x^1
$\Upsilon_{x_0} < \Upsilon_x^1$
$\Upsilon_{x_2} < \Upsilon_x^1$
t = 2
$\mathcal{C}_0 = \{x_1, x_0\}$
$\Upsilon_x^2 = 0$

Fig. 3: Greedy support selection: At $t = 1$ two x_0 and x_1 have the same cost $\mathcal{H}(x_i)=3$. The tie is broken at random and x_1 is selected. At $t = 2$ the divisor x_0 completes the covering.

Fig. 3 illustrates the greedy support selection using the 5-dimensional *simulation signatures* of three divisors: at each step t, the divisors are tested based on the IG covering they induce, until obtaining the support $\mathcal{C}_0 = \{x_1, x_0\}$.

The greedy heuristic does not guarantee finding the smallest size support or the best support for the resynthesis problem. To increase the capability of the algorithm to explore small size solutions, we define a statistical version of the algorithm. At each step t of the *covering process*, we define a probability distribution over the candidate divisors, and we sample a divisor from it. The probability distribution reads

$$p(x_i; \beta, t) \propto e^{-\beta\mathcal{H}(\Upsilon_{x_i} < \Upsilon_x^{t-1})} \qquad \forall x_i \in \mathcal{D} \tag{4}$$

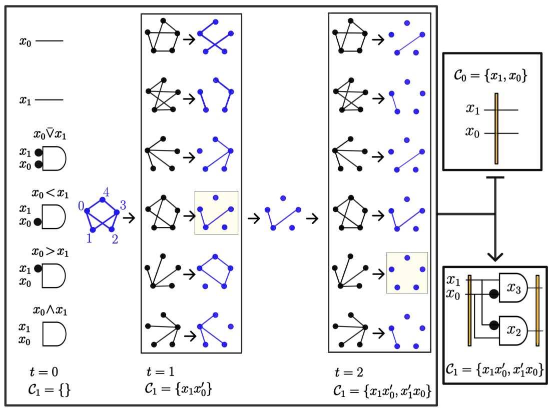
The divisors covering the largest number of edges have the highest probabilities. In the limit $\beta \to \infty$, the algorithm degenerates to the greedy approach: the only divisors with non zero probability are the ones with the lowest $\mathcal{H}(\Upsilon_{x_i} < \Upsilon_x^{t-1})$. As β decreases, the likelihood of selecting locally sub-optimal (but potentially globally optimal) divisors increases, until all divisors are equally likely in the limit $\beta \to 0$.

The flexibility of the statistical approach comes at the cost of an extra $\mathcal{O}(D)$ runtime at each iteration for sampling.

C. SPFD-Based Cut-by-Cut Synthesis

Let x be a target node, and $\mathcal{C}_0$ a set of divisors satisfying Eq. 3, i.e., a support. The *resynthesis problem* consists of finding a sub-circuit taking the nodes in $\mathcal{C}_0$ as the input, and having the output node with the same *global function* as x. We *synthesize* circuits one cut at the time. We aim for networks with a close-to-minimum number of nodes to maximize the optimization potential of resynthesis. The set $\mathcal{C}_0$ contains the inputs of a small network, or a set of divisors in a bigger circuit. At each level l, we add a new cut to the netlist, i.e., a set $\mathcal{C}_l = \{x_{c_l(i)}\}_{i=1}^{k}$. Following the discussion in Sec. II-B, each cut $\mathcal{C}_l$ must satisfy Eq. 3, i.e., $\mathcal{C}_l$ must have enough information to resynthesize x.

Synthesizing a circuit one cut at the time has the advantage that, at each level l, there is a small number of candidate new nodes to add: each element of $\mathcal{C}_l$ is either a wire pushing a variable in $\mathcal{C}_{l-1}$ forward, or a binary Boolean operation from the set $\{\bar{\vee}, <, >, \wedge, \oplus\}$ between two variables in $\mathcal{C}_{l-1}$. The enumeration of these candidates identifies a set $\mathcal{D}_l$, from which we can find a cut $\mathcal{C}_l$ by solving the *support selection* problem. We iterate the procedure until there is a cut of size 1. Fig. 4 shows the first step of the procedure for $\mathcal{C}_0 = \{x_0, x_1\}$. We



x_0
x_1
$x_0 \bar{\vee} x_1$
$x_0 < x_1$
$x_0 > x_1$
$x_0 \wedge x_1$
$\mathcal{C}_0 = \{x_1, x_0\}$
$\mathcal{C}_1 = \{x_1 x_0', x_1' x_0\}$
$t = 0$
$\mathcal{C}_1 = \{\}$
$t = 1$
$\mathcal{C}_1 = \{x_1 x_0'\}$
$t = 2$
$\mathcal{C}_1 = \{x_1 x_0', x_1' x_0\}$

Fig. 4: Adding the cut $\mathcal{C}_1$ to the circuit. From left to right: enumeration of the candidate nodes and selection of the cut.

remove the XOR to make the example non-trivial. Then, $\mathcal{D}_1 = \{x_0, x_1, x_0 \bar{\vee} x_1, x_0 < x_1, x_0 > x_1, x_0 \wedge x_1\}$, from which we obtain the cut $\mathcal{C}_1 = \{x_1 < x_0, x_1 > x_0\}$.

Compared to previously proposed cut-by-cut synthesis methods [19], our approach is simpler, and its design space exploration capabilities are noteworthy in their own right.

IV. Experiments

A. SPFD-Based Synthesis

The goal of this experiment is to show that, if provided with the support divisors, our resynthesis method can identify high-quality solutions in the design space. We target a Boolean function for which no state-of-the-art decomposition algorithm finds the optimum XAIG (12 nodes) [9]. The hexadecimal representation of the truth table of this function is 0x43B86C25. We used our engine to synthesize 10 circuits at 150 different calls (10 designs for each statistically selected first cut). We repeated the experiment varying β. The small size of the problem allows us to use the gSPFD instead of the aSPFD. Fig. 5 shows that we can find an optimum XAIG. The convergence time to the optimum is influenced by β. We validated the effectiveness of this synthesis method with $\beta = 100$ via experiments on other 5-input functions with known optimum [4], which we do not report for space limitation reasons.

B. SPFD-Based Resubstitution

We show now the effectiveness of our technique by applying it to a set of standard benchmark circuits. In this experiment we show that `irs` can find optimization transformations not accessible to the state-of-the-art resubstitution algorithm `urs` [10]. First, we iteratively optimize industrial designs with `urs` until no further improvement can be found ($\texttt{urs}^{\infty}$). Next, we apply one round of our strategy. We use the notation $\texttt{irs}_{K,S,I}$: K is the maximum allowed support size, S is the maximum number of supports we sample from the candidate divisors for optimizing a node, and I is the number of calls to the statistical resynthesis engine for a given support. Increasing these parameters enables higher effort optimization.

We used $N_{max} = 100$ for both `urs` and `irs`. Despite the high effort optimization of the state-of-the-art engine, `irs` unlocks additional improvements in most designs.

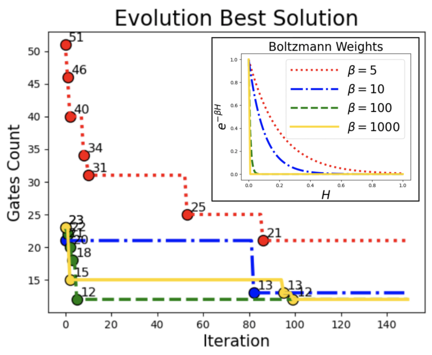


Fig. 5: Function 0x43B86C25: size of the best XAIG found during the exploration of the design space. Experiment used to set $\beta = 100$ based on the convergence time to the optimum.

TABLE I: `irs` on the ISCAS and EPFL benchmarks, pre-optimized with the `urs` algorithm [10] until convergence.

Design	$\texttt{urs}^{\infty}$	$\rightarrow \texttt{irs}_{4,1,1}$		$\rightarrow \texttt{irs}_{7,10,10}$	
ISCAS	|XAIG|	|XAIG|	time[s]	|XAIG|	time[s]
c17	6	6	0.00	6	0.00
c432	166	166	0.01	166	0.15
c499	388	**246**	0.03	**214**	0.15
c880	296	**269**	0.02	**261**	0.27
c1355	420	**263**	0.03	**202**	0.14
c1908	280	**181**	0.02	**165**	0.15
c2670	532	**484**	0.04	**455**	0.65
c3540	787	**750**	0.08	**730**	1.40
c5315	1277	**1211**	0.10	**1168**	1.27
c6288	1480	**1426**	0.07	**1420**	0.48
c7552	1291	**1146**	0.11	**1039**	1.41
		−13.84%		**−18.50%**	
EPFL	|XAIG|	|XAIG|	time[s]	|XAIG|	time[s]
adder	892	**653**	0.07	**643**	0.22
bar	2968	**2904**	0.35	**2840**	6.43
div	38942	**33304**	11.09	**32876**	20.81
hyp	205329	**171170**	114.97	**169450**	178.07
log2	29390	**26316**	33.75	**24858**	78.61
max	2862	2862	0.24	2862	3.64
multiplier	25403	**20341**	5.94	**19103**	27.40
sin	4929	**4578**	1.63	**4444**	7.82
sqrt	18450	**18015**	3.92	**16559**	19.58
square	16199	**14261**	1.79	**13460**	7.47
arbiter	11839	11839	0.90	11839	16.37
cavlc	591	591	0.17	**588**	2.78
ctrl	85	85	0.01	85	0.10
dec	304	304	0.01	304	0.01
i2c	1151	1145	0.10	**1145**	1.52
int2float	204	204	0.03	**202**	0.50
mem_ctrl	33579	**33303**	3.73	**32745**	48.34
priority	484	484	0.04	484	0.48
router	205	**181**	0.03	**177**	0.26
voter	7325	**6922**	0.72	**6416**	5.68
		−6.52%		**−8.65%**	

V. Conclusions

This paper proposes a new synthesis heuristic capable of achieving optimum or close-to-optimum XAIGs for *non-simply decomposable* functions with more than 3 inputs. By using this algorithm in a resubstitution engine, we show that it pushes the optimization capabilities of *resubstitution* beyond the limitations of state-of-the-art engines.

References

[1] G. D. Micheli, *Synthesis and optimization of digital circuits*. McGraw-Hill Higher Education, 1994.

[2] S. Chatterjee, *On algorithms for technology mapping*. University of California, Berkeley, 2007.

[3] A. T. Calvino, H. Riener, S. Rai, A. Kumar, and G. De Micheli, "A versatile mapping approach for technology mapping and graph optimization," in *2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2022, pp. 410–416.

[4] D. E. Knuth, *The art of computer programming, volume 4A: combinatorial algorithms, part 1*. Pearson Education India, 2011.

[5] S.-Y. Lee and G. De Micheli, "Heuristic logic resynthesis algorithms at the core of peephole optimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2023.

[6] A. Mishchenko and R. Brayton, "Scalable logic synthesis using a simple circuit structure," in *Proc. IWLS*, vol. 6, 2006, pp. 15–22.

[7] R. K. Brayton, "The decomposition and factorization of boolean expressions," *ISCAS-82*, pp. 49–54, 1982.

[8] J. S. Zhang, S. Sinha, A. Mishchenko, R. K. Brayton, and M. Chrzanowska-Jeske, "Simulation and satisfiability in logic synthesis," *computing*, vol. 7, p. 14, 2005.

[9] R. Brayton and A. Mishchenko, "Abc: An academic industrial-strength verification tool," in *Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings 22*. Springer, 2010, pp. 24–40.

[10] S.-Y. Lee, H. Riener, A. Mishchenko, R. K. Brayton, and G. De Micheli, "A simulation-guided paradigm for logic synthesis and verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 8, pp. 2573–2586, 2021.

[11] S. Sinha, *SPFDs: A new approach to flexibility in logic synthesis*. University of California, Berkeley, 2002.

[12] R. K. Brayton, "Understanding spfds: A new method for specifying flexibility," in *Notes of International Workshop on Logic Synthesis (IWLS'97), May*, 1997.

[13] S. Muroga, Y. Kambayashi, H. C. Lai, and J. N. Culliney, "The transduction method-design of logic networks based on permissible functions," *IEEE Transactions on Computers*, vol. 38, no. 10, pp. 1404–1424, 1989.

[14] I. Háleček, P. Fišer, and J. Schmidt, "Are xors in logic synthesis really necessary?" in *2017 IEEE 20th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*. IEEE, 2017, pp. 134–139.

[15] Y.-S. Yang, S. Sinha, A. Veneris, and R. K. Brayton, "Automating logic rectification by approximate spfds," in *2007 Asia and South Pacific Design Automation Conference*. IEEE, 2007, pp. 402–407.

[16] L. Józwiak, "Information relationships and measures: an analysis apparatus for efficient information system synthesis," in *EUROMICRO 97. Proceedings of the 23rd EUROMICRO Conference: New Frontiers of Information Technology (Cat. No. 97TB100167)*. IEEE, 1997, pp. 13–23.

[17] J.-H. R. Jiang and R. K. Brayton, "Functional dependency for verification reduction," in *Computer Aided Verification: 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004. Proceedings 16*. Springer, 2004, pp. 268–280.

[18] H. Sato, Y. Yasue, Y. Matsunaga, and M. Fujita, "Boolean resubstitution with permissible functions and binary decision diagrams," in *Proceedings of the 27th ACM/IEEE Design Automation Conference*, 1991, pp. 284–289.

[19] E. Goldberg, K. Gulati, and S. Khatri, "Toggle equivalence preserving (tep) logic optimization," in *10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007)*. IEEE, 2007, pp. 271–279.

[20] Bertacco and Damiani, "The disjunctive decomposition of logic functions," in *1997 Proceedings of IEEE International Conference on Computer Aided Design (ICCAD)*. IEEE, 1997, pp. 78–82.

[21] V. V. Vazirani, *Approximation algorithms*. Springer, 2001, vol. 1.