Scalable Generic Logic Synthesis: One Approach to Rule Them All

Heinz Riener
EPFL, Lausanne, CH

Eleonora Testa
EPFL, Lausanne, CH

Winston Haaswijk
EPFL, Lausanne, CH

Alan Mishchenko
University of California, Berkeley, CA, USA

Luca Amarù
Synopsys Inc., Sunnyvale, CA, USA

Giovanni De Micheli
EPFL, Lausanne, CH

Mathias Soeken
EPFL, Lausanne, CH

ABSTRACT

This paper proposes a novel methodology for multi-level logic synthesis that is independent from a specific graph data-structure, but formulates synthesis procedures using an abstract concept definition of a logic representation. The idea is to capture the essence of optimisations in a general manner and tailor only small performance-critical sections to the underlying logic representation. This generic, yet scalable approach, saves many man-months of development time and enables logic synthesis and technology-mapping procedures parameterised in a logic representation. We present the generic design methodology and demonstrate its practicality by providing a complete state-of-the-art logic synthesis flow.

ACM Reference Format:
Heinz Riener, Eleonora Testa, Winston Haaswijk, Alan Mishchenko, Luca Amarù, Giovanni De Micheli, and Mathias Soeken. 2019. Scalable Generic Logic Synthesis: One Approach to Rule Them All. In *The 56th Annual Design Automation Conference 2019 (DAC '19), June 2–6, 2019, Las Vegas, NV, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3316781.3317905

1 INTRODUCTION

Logic synthesis is a core engine in all *Electronic Design Automation* (EDA) tools. Its purpose is to express a functional design specification in terms of logic gates, while optimizing area, delay, and/or power—with the major challenge to algorithmically scale-up to today's design sizes. State-of-the-art logic synthesis algorithms [1, 2] achieve this goal by operating in three steps: (1) The design specification is expressed as a simple, technology-independent logic representation. (2) Fast peephole optimization techniques are repeatedly applied to optimize the logic representation. (3) The optimized logic is mapped to a technology-specific representation.

The most widely used technology-independent logic representation are *And-inverter graphs* (AIGs) [3], a homogeneous graph data-structure consisting of two-input AND gates and inverters. In recent years, several drop-in replacements for AIGs have been proposed: *Majority-inverter graphs* (MIG) [4, 5] consist of three-input majority gates and inverters. Their usage is motivated by the fact that many nano-emerging technologies, e.g., spin-wave devices or quantum-dot cellular automata, can be realized in terms of majority voters, such that technology-mapping for these technologies is significantly simplified. Moreover, the majority operation enables several novel optimizing transformations on the intermediate logic, which led in previous work to impressive delay reductions for arithmetic-intensive benchmark circuits. Alternatively, XOR-enhanced logic representations, such as *Xor-And graphs* (XAG) [6] or *Xor-majority graphs* (XMG) [7], extend AIGs and MIGs with a two-input and three-input XOR gate, respectively. They offer an improved compactness from which especially rewriting techniques beneft that repeatedly match small sub-networks and replace them with their size-optimal representations.

As of today, no logic synthesis tool is entirely based on MIGs, XAGs, or XMGs. In contrast, the advantages of these individual graph data-structures have been evaluated for specific optimizing procedures, such that the overall optimization potential of the representations remain unknown. The development of a sophisticated and complete state-of-the-art logic synthesis flow, however, can take several man-months for each of them.

In this paper, we propose a novel methodology for logic synthesis that is independent from a specific graph data-structure. Instead, our methodology formulates optimization procedures using an abstract concept definition of a logic representation. The idea is to capture the common essence of logic synthesis and technology-mapping procedures in a generic way and to tailor only small performance-critical sections to specific graph data-structures. We propose the concept definitions and present the four most common optimizations (rewriting, resubstitution, refactoring, and balancing) as well as k-LUT mapping generically applicable to any graph-based multi-level logic representation. We have implemented the generalized approach in C++ using template meta-programming, which allows us to make a fair comparison of the advantages of the different logic representations. We propose a generic resynthesis flow for area optimization similar to the state-of-the-art optimization script, `compress2rs`, in the logic synthesis package ABC [2]: (1) in a comparison with ABC, we show that the generic resynthesis flow using AIGs as logic representation is competitive with state-of-the-art logic synthesis algorithms, that are specifically designed for the optimization of AIGs; (2) using this generic approach, we propose, for the first time, a complete resynthesis flow for MIGs and XAGs; (3) we further present a fair comparison of AIGs, MIGs, and XAGs in a complete resynthesis flow and show that the individual logic representations are capable of achieving similar improvements (29.53%, 27.01%, and 29.82% for AIGs, MIGs, and XAGs, respectively) when compared after mapping into 6-input look-up tables (LUTs).

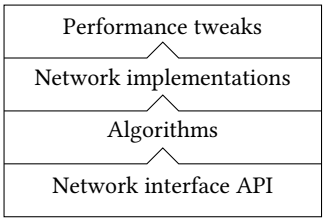


Figure 1: Stacked 4-layered architecture for supporting logic synthesis independent from a specific graph data-structure.

2 SCALABLE GENERIC LOGIC SYNTHESIS

Modern academic multi-level logic synthesis tools typically rely on a single logic representation, such as AIGs, as the basis for most optimization efforts. Using this representation, they are able to implement fundamental logic synthesis operations in an efficient and scalable way. The scalability of this approach comes from the compactness of the underlying logic representation, allowing such tools to represent large logic networks. Furthermore, the representation is often chosen so as to allow for the efficient implementation of fundamental techniques such as cut enumeration, Boolean reasoning, and DAG-awareness. In this section, we propose a *generic* methodology for logic synthesis, which generalizes the conventional approach described above. Our methodology is independent from any particular logic representation while maintaining efficiency and scalability. It allows users to tweak performance and to implement representation-specific code when necessary.

The central idea behind our method is to describe logic synthesis techniques generically, in terms of an abstract concept definition of logic networks. This abstract definition is applicable to any graph-based multi-level logic representation. Besides being generic, our approach is *scalable*, because it relies on the same efficient techniques for cut enumeration, fast truth table computation, and sophisticated exact synthesis engines as the conventional single-representation approach. In the remainder of this section we describe our generic methodology in more detail.

2.1 Genericity

The backbone of the proposed methodology is a stacked 4-layered architecture, depicted in Fig. 1. The base layer shown on the bottom provides an abstract concept definition of a network API, called *Network Interface API*, which defines a logic representation in terms of primary inputs, primary outputs, and logic gates. Naming conventions for types and methods ensure that all network interfaces can be used in the same way. Some of these conventions are mandatory, while others are optional. Mandatory interfaces are, for instance, methods to iterate over input, output or gates, which have to be provided for a logic representation; an example of an optional interface is a method that allows its user to query the level of a node in the logic representation, which may or may not be provided by a logic representation. The network interface API does not provide an implementation of a logic network.

Algorithms, the second layer of the methodology, are generically formulated on top of the network interface API. An algorithm takes as input an instance of a network type, that is required to implement all mandatory and some optional interface as defined by the algorithm. The algorithms make use of the network interface API,

```
Data: Logic network N
Result: depth d of the logic network
foreachInput n in N do
    set ℓ(n) ← 0;
foreachGate n in N do
    set ℓ(n) ← 0;
    foreachFanin n′ of n do
        set ℓ(n) ← max(ℓ(n), ℓ(n′));
    set ℓ(n) ← ℓ(n) + 1;
set d ← 0;
foreachOutput n in N do
    set d ← max(d, ℓ(n));
return d
```

Algorithm 1: Generic algorithm to compute logic depth

but make no assumptions on the internal implementations of the input networks. For instance, no algorithm depends on how gates of the network are internally represented. Rather, gates are accessed through the network API. Many algorithms in logic synthesis, such as algorithms for computing cuts or maximum fanout-free cones, can be formulated easily using graph-based analysis procedures without requiring knowledge of the internal logic network. Algorithm 1 shows an example algorithm to compute the depth of a logic network. For this task only four methods of the network API are required: *foreachInput* to iterate over all primary inputs, *foreachGate* to iterate over all gates, *foreachFanin* to iterate over all fanins of a gate, and *foreachOutput* to iterate over all outputs of a network.

The third layer, *network implementations*, consists of actual implementations of logic networks that provide concrete definitions of the network interface API, e.g., And-inverter graphs, Majority-inverter graphs, or k-LUT networks. In particular, the network implementations define the node type used to represent logic and a storage which contains the generated nodes. Technical details such as structural hashing are implemented on this layer. Algorithms from the second layer can be called on instances of the network types, if they implement the required interfaces. Static compile-time assertions ensure that compilation only succeeds for those network implementations that do provide all required types and methods. This further avoids dynamic polymorphism, which adds unnecessary overhead to the runtime.

Finally, on the last layer, *performance tweaking*, offers the possibility to improve performance by specializing some algorithmic details for specific network types based on their internal implementation. This is done for each network individually and without affecting the generic implementation nor the implementation of other network types.

This methodology provides a generic approach to logic synthesis, while offering its user's the possibility to tweak algorithmic performance if indispensable.

2.2 Scalability

Using the stacked 4-layer architecture described in Section 2.1, we implement efficient and scalable optimization methods in a way that is representation-agnostic. In this section, we describe fundamental principles and algorithms that we have implemented in this generically scalable way. They form the basis for the complete synthesis flow developed in Section 3.

2.2.1 Cut Enumeration. A common operation in logic optimization algorithms is the partitioning of the global subject graph into smaller subnetworks. The smaller size of these subnetworks allows us to perform peephole optimizations, in which we can exert control by, for example, restricting the number of inputs to subnetworks. This makes them more amenable to various optimizations methods, such as balancing, resubstitution, or exact synthesis. Cut enumeration is a common method to create such partitions.

Our methodology supports two distinct types of cut enumeration: (i) bottom-up enumeration through the Cartesian product method [8], and (ii) top-down cut enumeration based on reconvergence-driven cuts [1]. Both of these methods are suitable for different optimization algorithms. For instance, rewriting methods commonly use bottom-up cut enumeration, whereas top-down cut enumeration is used by resubstitution algorithms. Moreover, both of these methods are suitable for generic implementation, as they rely essentially only on notions of fanin connectivity. Hence, our generic DAG-based approach supports this in a straightforward way by defining the appropriate fanin interface functions.

2.2.2 Boolean Reasoning. Peephole optimization methods focus on repeatedly optimizing small sections of a logic network, e.g., a cut, with a restricted number of inputs. If the number of inputs does not exceed $16 - 20$, explicit, exhaustive simulation techniques based on the usage of truth tables outperform heavy-weight Boolean reasoning techniques. As a consequence, truth tables can be used as the basis for various optimization methods.

In particular, they allow for the efficient implementation of another important Boolean reasoning method: SAT-based exact synthesis [9], a powerful optimization technique that computes optimum representations of Boolean functions. As this technique finds exact optima using a SAT solver, it does not scale to large functions. However, it is suitable for use as a peephole optimization, e.g. for subnetworks with up to 8 inputs. Our generic implementation supports state-of-the-art SAT-based exact synthesis, including those based on families of DAG topologies such as fences [10]. Moreover, through its specialization mechanism, our method also supports exact synthesis encodings that are tuned to specific logic representations like AIGs or XAGs.

2.2.3 DAG-Awareness. Many logic optimization algorithms, such as balancing or rewriting, can be viewed as a *restructuring* of the DAG representation of the logic network, where local subnetworks are replaced by new structures. We generally only want to replace a subnetwork by a new structure if the replacement leads to some positive gain.[1] In theory, this gain may be expressed by an arbitrary cost function. In practice, we are interested in objectives such as size optimization, so we focus on reducing the overall number of nodes in the subject graph. DAG-awareness refers to the notion that the gain of a replacement structure takes into account both the existing graph structure as well as the replacement structure. In doing so, it can find opportunities for logic sharing, thus enabling more efficient replacements.

To compute replacement gains, we make use of reference counting and assign a value to each node in the network. These values are initialized with the nodes' fanout sizes. New nodes that are added to the network for a possible replacement will be assigned a reference count of 0. The reference count of a node indicates how many other nodes require this node in the network. In particular, a reference count of 0 means that the node is not required in the network. Thus, reference counting can be used to measure how many nodes a given replacement removes from the network. Our generic implementation considers structural hashing (for those network types that support it). In other words, nodes from a replacement candidate that are already in the network will not be added another time. Moreover, their reference counters will not be changed. To simulate the removal of a node n from a network, we recursively decrement all predecessors in the transitive fanin of the node and continue as long as the reference counters of a child become 0 or a leaf node is reached. Full details for our DAG-aware replacement algorithm are outside the scope of this paper. We refer the interested reader to [11] and [12].

[1] Under some circumstances it is advantageous to also allow zero-cost replacements.

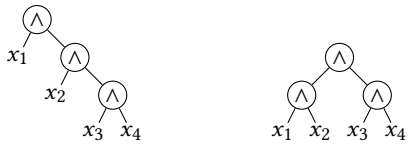
2.3 Generic Logic Optimization Algorithms

In the following, we present the four most common optimizations (rewriting, refactoring, resubstitution, and balancing) generically applicable to any graph-based multi-level logic representation.

2.3.1 Balancing. Balancing is a technique to reduce the number of logic levels in a network [13]. Depending on the application, balancing may or may not permit to increase the network size. In this paper, we consider balancing methods that do not increase the network size. A generic balancing method is *tree-balancing*. So far, tree-balancing has been described and implemented in terms of AIGs [13], exploiting the associativity of the AND operation, $x_1 \wedge (x_2 \wedge x_3) = (x_1 \wedge x_2) \wedge x_3$. However, sufficient requirements for tree-balancing are commutativity and associativity of the gate function. This observation allows us to implement a generic balancing function. For example, if $f(x_1, f(x_2, x_3)) = f(f(x_1, x_2), x_3)$, then we can rewrite the expression

$$f(x_1, f(x_2, f(x_3, x_4))) \text{ into } f(f(x_1, x_2), f(x_3, x_4)), \tag{1}$$

which requires one fewer logic level while using the same number of operations. This property holds for AND gates, XOR gates, and for Majority gates, if all operations share a common input value, i.e., $\langle x_1 u \langle x_2 u x_3 \rangle \rangle = \langle \langle x_1 u x_2 \rangle u x_3 \rangle$. The following illustration shows balancing applied to three AND gates.



Algorithm 2 illustrates a generic tree-balancing algorithm. It performs two steps. In the first step, it groups together gates of the same type if they permit commutativity and associativity, and if the group does not contain complemented edges or external fanout, except for the root node. In the second step, the gates and the group are rearranged by approaching a balanced tree by applying the principle in (1), by additionally taking into account the arrival times of the inputs. While the first step is unique, different tree decompositions are possible in the second steps. However, none of the possible decompositions leads to an increase in gate count. On the contrary, a tree decomposition can lead to gates that are already in the network, and logic sharing will decrease the overall gate count. Besides methods to iterate through the gates of a network, the balancing algorithm requires that the methods *gateFunction* and *substituteNode* are implemented for the logic network.

```
Data: Logic network N
foreachGate n in N do
    set f ← N.gateFunction(n);
    if isAssociative(f) then
        set G ← growGroup(n, f);
        set n′ ← treeBalance(G);
        N.substituteNode(n, n′);
```

Algorithm 2: Generic balancing algorithm

2.3.2 Rewriting. The so-called *DAG-aware* logic rewriting algorithm [11] is an efficient method of replacing parts of a logic network by different, but equivalent, logic structures. Based on cut enumeration, the algorithm partitions the subject graph into small subnetworks, typically ranging from 4 to 8 inputs. For each subnetwork, the rewriting procedure attempts to find a better representation, thus performing a local optimization. This optimization is commonly implemented in one of two ways: (i) optimization networks are drawn from a precomputed database of optimal structures, or (ii) replacement structures are computed at runtime using an *exact synthesis* [12]. Our generic methodology naturally supports both approaches. As illustrated in Algorithm 3, for each subnetwork, the algorithm computes the gain of replacing the existing structure with potential replacement structures. This is done in a DAG-aware way, as described in Section 2.2.3, where reference counters are recursively incremented and decremented using the functions *ref* and *deref*, respectively. Thus, the algorithm obtains global optimizations by locally *rewriting* the network.

```
Data: Logic network N
enumerateCuts(N);
foreachGate n in N do
    foreach cut C of n do
        set v ← deref(n);
        set f ← computeTruthTable(C);
        set n′ ← synthesize(f);
        set v′ ← ref(n′);
        if v′ < v′ then
            N.substituteNode(n, n′);
        else
            deref(n′);
            ref(n);
```

Algorithm 3: Generic rewriting algorithm

There is not one CNF encoding that works best for all combinations of logic representations and SAT solvers. Hence, it is sometimes necessary to construct specialized encodings for certain representations. Fortunately, as described above, our method lends itself to such specializations. Indeed, we use different encodings to rewrite AIG and XAG networks, but this is completely transparent to users of the rewriting flow. Using the same optimization script, our method automatically generates the appropriate CNF, depending on what logic representation is used.

2.3.3 Refactoring. Refactoring is a technique which resynthesizes large parts of the network without aiming to reuse existing logic in the network. Since it collapses a rather large part of the network into a Boolean function and then resynthesizes it from scratch, it is a powerful technique to overcome structural bias.

```
Data: Logic network N, fanin limit l, cost function cost
foreachGate n in N do
    set M ← computeMFFC(n, l);
    set f ← computeTruthTable(M);
    set M′, n′ ← synthesize(f);
    if cost(M′) < cost(M) then
        N.substituteNode(n, n′);
```

Algorithm 4: Generic refactoring algorithm

Algorithm 4 illustrates the generic procedure. For each node in the network, refactoring computes the node's fanout-free cone with a restricted fanin such that its truth table or a two-level representation, e.g., sum-of-products, can be efficiently computed. Based on this function, a new network structure is computed that uses the gates from the targeted logic network data structure. Various synthesis methods can be used for this purpose (see, e.g., [14, 15]). The new network structure is compared to the fanout-free cone with respect to a given cost function, e.g., size. If the new network structure is less expensive with respect to the cost function, the fanout-free cone can be removed and replaced by the new data structure. Note that fanout-free cone has by construction no external fanout, except for the root node. Moreover, some of the required network API is hidden behind functions such as *simulate*.

Refactoring is suitable for generic logic synthesis. Only two aspects require knowledge about the supported gates types: 1) simulation of the fanout-free cone; 2) resynthesis of the collapsed function into a new data structure. For the latter, containment relations among the logic network types can be exploited. For example, a synthesis algorithm that generates an AIG for a truth table, can also be used inside refactoring for MIGs or XAGs.

2.3.4 Resubsitution. Boolean resubstitution is an optimizing transformation that re-expresses the logic function of a node in a logic network using nodes already existing in the network. Resubstitution techniques are distinguished by the maximal number k of operators added to a logic network to re-express a logic function. A substitution of a candidate node is beneficial, if the number of nodes freed after substitution is greater than k, such that particularly roots of maximal fanout-free cones qualify as candidates for Boolean resubstitution.

Algorithm 5 presents a generic resubstitution procedure. The logic function of each node n in the logic network is computed locally in a reconvergence-driven cut C with restricted fanin size using exhaustive truth table simulation. Each node $d \neq n$ in the reconvergence-driven cut, which is not part of the maximum fanout-free cone, is a divisor that potentially be used for resynthesis. The core of k-resubstitution is a computational kernel, tryResubstitute that selects up to k divisors $D = \{d_1, \ldots, d_k\}$ and tests if n can be resynthesized using them. On success, the new nodes required for resynthesis (at most k) are generated, added to the logic network, and the gain λ of resubstituting n is computed using DAG-aware rewriting techniques. If $\lambda > 0$, n is substituted with the root n' of the new nodes.

Only the computational kernel of the Boolean resubstitution technique depends on the underlying logic representation. For each

```
Data: Logic network N, maximum number d of inserted nodes
foreachGate n in N do
    set C ← findCut(N, n);
    set M ← computeMFFC(N, n);
    if |M| > 0 then
        set D ← collectDivisors(N, n);
        computeTruthTables(C);
        for k = 0 to d do
            if n' ← tryResubstitute(N, n, k, D) then
                set λ ← computeGain(N, n, n');
                if λ > 0 then
                    N.substituteNode(n, n');
                    break;
```

Algorithm 5: Generic resubstitution algorithm

supported logic representation, the computational kernel has to specialize three aspects that depend on the supported gate types:

(1) *Divisor selection*: The gate types (and their fanin sizes) supported by the logic representation define the selection of divisors. For instance, for 1-resubstitution in MIGs, consisting of majority-of-three gates, three divisors have to be selected, whereas in AIGs, consisting of AND nodes with fanin size 2, only two divisors have to be selected.
(2) *Resubstitution rules*: For each topology composed of k gates, a computational kernel has to be defined that compares the logic function of the node to be substituted with the logic function of the topology with the selected divisors and, on success, triggers the substitution of the node.
(3) *Filtering rules*: A Boolean filter is a sufficient condition used to rule out divisors that cannot be resubstituted. Several Boolean filters have been discovered for different gate types [16, 17]. They are reported to significantly improve the performance and scalability of Boolean resubstitution techniques.

3 EXPERIMENTAL RESULTS

In this section, we present a generic resynthesis flow implemented using the EPFL logic synthesis libraries[2] [18]. We evaluate the flow in area optimization for LUT-mapping showing that different graph representations (AIGs, MIGs, and XAGs) can be used to represent logic networks. As benchmarks, we use the EPFL combinational benchmarks suite[3], which are initially provided as AIGs.

3.1 Generic Resynthesis Flow

We create a generic logic resynthesis flow, based on the standard non-depth-preserving area optimization flow `compress2rs` in the logic synthesis package ABC [2]. The flow consists of the following sequence of optimizing transformations:

```
bz; rs -c 6; rw; rs -c 6 -d 2; rf; rs -c 8; bz;
rs -c 8 -d 2; rw; rs -c 10; rwz; rs -c 10 -d 2;
bz; rs -c 12; rfz; rs -c 12 -d 2; rwz; bz;
```

where `b`, `rs`, `rw`, and `rf` refer to balancing, resubstitution, rewriting, and refactoring, respectively. The additional suffix `z` is appended to denote a zero-gain transformations, which not necessarily reduces the size of a logic network but uses resynthesis to restructure the benchmark and thus enable other optimization capabilities for following transformations. The parameters `-c` and `-d` for `rs` denote the maximum size of a cut and the maximum number of gates inserted by resubstitution.

[2]EPFL logic synthesis libraries, https://github.com/lsils/lstools-showcase
[3]EPFL combinational benchmark suite, https://github.com/lsils/benchmarks

Table 1: Apple-to-apple comparison with ABC.

Flows	Nd	Lvl	LUTs
Baseline (ABC)	1	1	1
Generic flow using AIGs	+1.14%	+3.02%	+0.65%

Comparing with ABC. We first compare in an apple-to-apple experiment the generic resynthesis flow using AIGs as logic network representation with the state-of-the-art logic synthesis package ABC to analyze the overhead of the generic flow to an approach specifically designed for AIGs. The accumulated results for all EPFL benchmarks are listed in Table 1. Our implementation of the generic resynthesis flow achieves similar results when compared to ABC, but requires in total 1.14% more nodes and 3.02% more levels. Note that the area optimization flow does not focus on preserving the depth of a logic representation during optimization. After LUT-mapping, the generic flow requires less than 1% more 6-LUTs. Overall, we conclude that the implementation of the generic approach is competitive to ABC.

Comparing different logic representations. In a second experiment, we compare the number of 6-LUTs after area optimization and LUT-mapping for FPGAs using the generic resynthesis flow with three different logic representations (AIGs, MIGs, and XAGs). As baseline, we use the EPFL benchmark suite in their AIG representation. Table 2 lists the number of nodes (Nd), number of levels (Lvl), the number of 6-LUTs (LUTs), as well as the required time (Time) for the baseline and the optimizations using AIGs, MIGs, and XAGs as logic representations, respectively.

In total, we conclude that our flow is capable of optimizing all three logic representations achieving comparable good results. Logic resynthesis using AIGs achieves in 8 cases the best results when compared to the other logic representation which leads to a total improvement of 30.04% in LUTs. MIGs are particularly good for the representation of arithmetic-intensive circuits and outperform the other data structures for the benchmarks `multiplier` and `sqrt`. In total, MIG optimizations achieve an improvement of 27.78%. Representing the logic networks as XAGs leads in 10 cases to the best results and in total to an improvement of 29.82%. In general, we advocate a portfolio approach which achieves a total improvement of 31.39%.

4 CONCLUSION

In this paper, we propose a generic representation-independent resynthesis methodology for multi-level logic synthesis, using an abstract concept definition of a logic network. Using this method we show, for the first time, a complete design flow for AIGs, MIGs, and XAGs that is competitive with the state of the art. Moreover, it allows us to compare resynthesis techniques using different logic representations, and to show that various logic synthesis algorithms work across representations. With the introduction of new representation forms, our method allows users to quickly develop completely novel design flows, by implementing a lightweight interface. Finally, our method enables users of various technology

Table 2: Optimization results for the EPFL benchmark suits using different logic representations.

Benchmark	I/O	Baseline			AIGs				MIGs				XAGs			
		Nd	Lvl	LUTs	Nd	Lvl	LUTs	Time	Nd	Lvl	LUTs	Time	Nd	Lvl	LUTs	Time
adder	256 / 129	1020	255	192	894	255	192	0.56	511	130	192	0.51	766	255	192	0.51
arbiter	256 / 129	11839	87	2595	11839	88	2594	28.21	6719	47	3112	9.73	11839	88	2594	17.78
bar	135 / 128	3336	12	512	2952	13	512	10.84	2939	14	512	4.62	2952	13	512	2.51
cavlc	10 / 11	693	16	119	662	18	122	14.56	558	14	165	12.24	668	18	124	0.78
ctrl	7 / 26	174	10	28	94	9	29	0.67	83	11	29	3.57	118	8	28	0.17
dec	8 / 256	304	3	273	304	3	273	0.35	304	3	273	0.55	304	3	273	0.32
div	128 / 128	57247	4372	23955	40782	4500	8182	437.19	33006	4303	8289	81.89	32696	4363	8212	124.94
i2c	147 / 142	1342	20	341	1167	17	319	3.55	1053	14	350	2.40	1197	16	321	0.98
int2float	11 / 7	260	16	47	212	17	48	1.98	191	11	58	1.06	209	18	48	0.14
log2	32 / 32	32060	444	8124	29509	398	8008	708.23	27300	372	8179	228.61	24305	332	7828	210.71
max	512 / 130	2865	287	744	2780	355	703	4.90	2212	133	734	2.51	2787	369	699	2.57
mem_ctrl	1204 / 1231	46836	114	11754	44846	126	11458	270.09	39338	118	13050	84.16	45480	134	11673	147.56
multiplier	128 / 128	27062	274	6578	24497	271	6342	546.01	22847	260	5579	57.58	18764	288	5723	119.38
priority	128 / 8	978	250	266	720	240	245	2.56	482	124	257	0.77	815	239	250	0.69
router	60 / 30	257	54	54	244	40	68	0.64	208	28	61	0.41	205	37	60	0.30
sin	24 / 25	5416	225	1584	5089	196	1593	36.22	4649	179	1624	33.70	4307	171	1515	9.62
sqrt	128 / 64	24618	5058	8204	18450	6100	4121	373.42	16039	7617	4030	22.55	14422	5996	4039	57.38
square	64 / 128	18484	250	4130	16566	249	4053	63.66	14799	137	4183	24.70	14061	279	3937	33.24
voter	1001 / 1	13758	70	2979	8586	70	1841	24.69	5082	68	1664	9.73	6814	51	1701	12.62
Total								2528.33				581.29				742.2
Improvement				0.00%			+30.04%				+27.78%				+31.39%	

types to choose an optimization flow based on the logic representation most suitable to them. For instance, users that work in the domain of nano-emerging technologies can use our tool to develop a complete majority-logic design flow.

The generic method presented in this paper opens the door to some interesting directions for future work. Currently, we use one logic representation across the entire design flow. However, it may be advantageous to dynamically switch between representations during the flow, as some representations may lend themselves more naturally to certain optimization steps. Another interesting path is the development of a portfolio logic synthesis method. Often, we do not know a priori which representation is best in a given domain. Our method allows one to run the same design flow with all representations and pick the best result. Finally, at the moment we use the same generic design flow with all representations. However, it may be that a certain combination of flow × representation unlocks very good optimizations in some specialized domain. Our method enables users to prototype and experiment with generic scripts which can be rewritten into more specialized design flows.

ACKNOWLEDGMENTS

This research was supported by the Swiss National Science Foundation (200021-169084 MAJesty), by the European Research Council in the project H2020-ERC-2014-ADG669354 CyberCare, by the EPFL Open Science Fund, and in part by SRC Contracts 2710.001 and 2867.001.

REFERENCES

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

[2] R. K. 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*, 2010, pp. 24–40. [Online]. Available: https://doi.org/10.1007/978-3-642-14295-6_5

[3] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, "Robust Boolean reasoning for equivalence checking and functional property verification," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 21, no. 12, pp. 1377–1394, 2002. [Online]. Available: https://doi.org/10.1109/TCAD.2002.804386

[4] L. G. Amarù, P. Gaillardon, and G. D. Micheli, "Majority-inverter graph: A novel data-structure and algorithms for efficient logic optimization," in *The 51st Annual Design Automation Conference 2014, DAC '14, San Francisco, CA, USA, June 1-5, 2014*, 2014, pp. 194:1–194:6. [Online]. Available: https://doi.org/10.1145/2593069.2593158

[5] ——, "Majority-inverter graph: A new paradigm for logic optimization," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 35, no. 5, pp. 806–819, 2016. [Online]. Available: https://doi.org/10.1109/TCAD.2015.2488484

[6] Ç. Çalik, M. S. Turan, and R. Peralta, "The multiplicative complexity of 6-variable Boolean functions," *IACR Cryptology ePrint Archive*, vol. 2018, p. 2, 2018. [Online]. Available: http://eprint.iacr.org/2018/002

[7] W. Haaswijk, M. Soeken, L. G. Amarù, P. Gaillardon, and G. D. Micheli, "A novel basis for logic rewriting," in *22nd Asia and South Pacific Design Automation Conference, ASP-DAC 2017, Chiba, Japan, January 16-19, 2017*, 2017, pp. 151–156. [Online]. Available: https://doi.org/10.1109/ASPDAC.2017.7858312

[8] P. Pan and C.-C. Lin, "A New Retiming-based Technology Mapping for LUT-based FPGAs Algorithm," in *FPGA 98*, 1998, pp. 35–42.

[9] M. Soeken, W. Haaswijk, E. Testa, A. Mishchenko, L. G. Amarù, R. K. Brayton, and G. D. Micheli, "Practical exact synthesis," in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2018, pp. 309–314.

[10] W. Haaswijk, A. Mishchenko, M. Soeken, and G. De Micheli, "SAT based exact synthesis using DAG topology families," in *Proceedings of the 55th Annual Design Automation Conference*, ser. DAC '18. New York, NY, USA: ACM, 2018, pp. 53:1–53:6. [Online]. Available: http://doi.acm.org/10.1145/3195970.3196111

[11] A. Mishchenko, S. Chatterjee, and R. K. Brayton, "DAG-aware AIG rewriting: a fresh look at combinational logic synthesis," 2006, pp. 532–535.

[12] H. Riener, W. Haaswijk, A. Mishchenko, G. D. Micheli, and M. Soeken, "On-the-fly and DAG-aware: Rewriting Boolean networks with exact synthesis," in *2019 Design, Automation & Test in Europe, DATE 2019, Florence, Italy, March 25-29, 2019*, 2019.

[13] A. Mishchenko, R. K. Brayton, S. Jang, and V. N. Kravets, "Delay optimization using SOP balancing," in *2011 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2011, San Jose, California, USA, November 7-10, 2011*, 2011, pp. 375–382. [Online]. Available: https://doi.org/10.1109/ICCAD.2011.6105357

[14] J. Rajski and J. Vasudevamurthy, "The testability-preserving concurrent decomposition and factorization of boolean expressions," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 11, no. 6, pp. 778–793, 1992. [Online]. Available: https://doi.org/10.1109/43.137523

[15] K. Chang, V. Bertacco, I. L. Markov, and A. Mishchenko, "Logic synthesis and circuit customization using extensive external don't-cares," *ACM Trans. Design Autom. Electr. Syst.*, vol. 15, no. 3, pp. 26:1–26:24, 2010. [Online]. Available: https://doi.org/10.1145/1754405.1754411

[16] H. Riener, E. Testa, L. Amaru, and G. Mathias Soeken De Micheli, "Size optimization of MIGs with an application to QCA and STMG technologies," in *IEEE/ACM International Symposium on Nanoscale Architectures*, 2018.

[17] L. G. Amarù, M. Soeken, P. Vuillod, J. Luo, A. Mishchenko, J. Olson, R. K. Brayton, and G. De Micheli, "Improvements to Boolean resynthesis," in *2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018*, 2018, pp. 755–760. [Online]. Available: https://doi.org/10.23919/DATE.2018.8342108

[18] M. Soeken, H. Riener, W. Haaswijk, and G. D. Micheli, "The EPFL logic synthesis libraries," *CoRR*, vol. abs/1805.05121, 2018. [Online]. Available: http://arxiv.org/abs/1805.05121