Simulation-Guided Boolean Resubstitution

Siang-Yun Lee
EPF Lausanne
Switzerland

Heinz Riener
EPF Lausanne
Switzerland

Alan Mishchenko
University of California, Berkeley
United States

Robert K. Brayton
University of California, Berkeley
United States

Giovanni De Micheli
EPF Lausanne
Switzerland

Abstract

This paper proposes a new logic optimization paradigm based on circuit simulation, which reduces the need for Boolean computations such as SAT-solving or constructing BDDs. The paper develops a Boolean resubstitution framework to demonstrate the effectiveness of the proposed approach. Methods to generate highly expressive simulation patterns are developed, and the generated patterns are used in resubstitution for efficient filtering of potential resubstitution candidates to reduce the need for SAT validation. Experimental results show that improvements in circuit size reduction were achieved by up to 74%, compared to a state-of-the-art resubstitution algorithm.

1 Introduction

Logic optimization and circuit minimization [3, 9] play an important role in *electronic design automation* (EDA). As the size and complexity of digital circuits grow, the demand for scalable optimizations of multi-level logic networks grows. Boolean methods, such as Boolean decomposition and resubstitution [12, 16], often improve the quality of logic synthesis but at the cost of more runtime taken by having to solve Boolean problems using a *satisfiability problem* (SAT) solver or a *binary decision diagram* (BDD) package. Algebraic and other local-search methods, on the other hand, are much faster because they are based on structural analysis, circuit simulation, *sum-of-product* (SOP) factorization, or other time-efficient computations. However, the reductions in area and delay cannot compete with those achieved by Boolean methods.

In this paper, we introduce a promising new paradigm, *simulation-guided logic synthesis*, that leverages simulation to minimize the number of expensive NP-oracle calls to equivalence checkers or SAT solvers during synthesis. The runtime gained can be used to further improve the synthesis quality, leading to faster place-and-route, as happened in one study [15]. The underlying hypothesis about using simulation is that *expressive* simulation patterns can be amassed for a logic network and used later as an efficient filter to avoid unnecessary formal checks. The proposed paradigm is useful in algorithms dominated by expensive Boolean computation and is particularly suitable for techniques such as (1) computation of structural choices [4] to improve the quality of mapping, (2) scalable combinational equivalence checking for large designs [14], and (3) gate matching between several versions of the same network. The resulting patterns can also be used in *automatic test pattern generation* (ATPG) [18] and in circuit reliability analysis [5].

We demonstrate the advantages of simulation guidance by presenting a fast and efficient Boolean resubstitution framework, which iterates over the nodes and attempts to re-express their functions using other nodes in the network. If updating a node's function makes other nodes in its fan-in cone dangling (i.e., having no fan-out), they can be deleted, resulting in the reduction of the network's size. For the special case of replacing a node directly with an existing node, it is equivalent to *functional reduction* (FRAIG) [10].

The resubstitution engine can quickly identify and rule out most illegal resubstitution candidates by simply comparing simulation signatures. SAT is used in two tasks: generating simulation patterns before synthesis, and validating resubstitutions found using simulation during synthesis. The runtime of the former task is not critical because the resulting patterns are reusable by different engines working on the same design, or by the same tool working on similar versions of the design. On the other hand, the runtime of SAT validation is reduced substantially, provided that the simulation patterns are expressive enough. To this end, we study what makes simulation patterns *expressive* and profile different pattern generation strategies, including random simulation, stuck-at-value testing, observability checking, and combinations of these. In the process of resubstitution, pre-computed simulation patterns can be refined further with the counter-examples generated by SAT.

The contributions of the paper are: (1) a simulation-based logic optimization framework, which separates the computation of expressive simulation patterns and their use to validate Boolean optimization choices; (2) methods to increase expressive power of simulation patterns, resulting in reduced runtime due to fewer SAT calls; (3) improvements to the computation of resubstitution functions, resulting in better quality of logic optimization. The experimental results show that our new engine allows implicit consideration of global satisfiability don't-cares, which achieves 41% improvement in circuit size reduction compared to a state-of-the-art windowing-based resubstitution algorithm [12]. The

proposed framework also allows low-cost extension of the window sizes used, resulting in a total of 74% improvement.

The rest of this paper is organized as follows: After preliminaries are given in Section 2 and related works introduced in Section 3, we first describe the general algorithmic framework in Section 4. Then, pattern generation methods are explained in Section 5, and simulation-guided resubstitution techniques are proposed in Section 6. Finally, experimental results are given in Section 7, and conclusions in Section 8.

2 Preliminaries

2.1 Logic Network

Logic networks are *directed acyclic graphs* (DAGs) composed of *logic gates* and realizing *Boolean functions*, which are functions defined over the Boolean space $\mathbb{B} = \{0, 1\}$, taking *primary inputs* as inputs and presenting the function outputs at the *primary outputs*. In this paper, we work with *and-inverter graphs* (AIGs), but this framework can also be applied to other types of logic networks.

A *gate* in a logic network usually computes a simple function of its *fan-ins*, and passes the resulting value to all of its *fan-outs*. In the case of AIGs, a gate is always an AND gate, and the inverters are represented by complemented wires with no cost (that is, they do not add to the circuit size). We also refer to a gate as a *node*. The *transitive fan-in* (TFI) or the *transitive fan-out* (TFO) of a node n is a set of nodes such that there is a path between n and these nodes in the direction of fan-in or fan-out, respectively.

2.2 Don't-Cares

The functionality of a logic circuit can be specified using an input-output relation. Input assignments when the output value does not matter, are called *external don't-cares* (EXDCs). These assignments can be utilized to optimize the circuit.

For a set of internal nodes of a circuit, there might be some value combinations that never appear at these nodes. For example, an AND gate g_1 and an OR gate g_2 sharing the same fan-ins can never have $g_1 = 1$ and $g_2 = 0$ at the same time. This combination is a *satisfiability don't-care* (SDC) of a common TFO node of g_1 and g_2.

An input assignment $\vec{x}$ is said to be *un-observable* for node n if the circuit outputs do not change when n is replaced by its negation $\overline{n}$. Or conversely, n is *un-testable* under $\vec{x}$. Because the function of n under $\vec{x}$ does not matter, the unobservable patterns, called *observability don't-cares* (ODCs), can be used to optimize node n.

2.3 Simulation

A *simulation pattern* is a set of values assigned to the primary inputs. Circuit simulation is done by visiting nodes in a topological order. The *simulation signature* of a node is an ordered set of values produced at the node under each simulation pattern. When the number of patterns is more than one, bit-parallel operations can be used to speed up simulation substantially.

2.4 Resubstitution

For each *root* node in the circuit that we want to replace, we first estimate its *gain*, or the number of nodes that can be deleted after a successful resubstitution, using the size of the node's *maximum fan-out free cone* (MFFC). A node n is said to be in the MFFC of the root node r if n is in the TFI of r and all path from n to the primary outputs pass through r. Then, a set of *divisors* is collected. A divisor is a node that can be used to express the function of the root node. It should not be in the TFO cone of the root, otherwise the resulting circuit would be cyclic. It should also not be in the MFFC because nodes in the MFFC may be removed after resubstitution. Nodes depending on primary inputs that are not in the TFI of the root node can also be filtered out from the set of potential divisors.

A *resubstitution candidate* (also abbreviated as a *candidate*) is either a divisor itself or a simple function, named the *dependency function*, built with several divisors. In the latter case, the candidate is represented by the top-most node of the implementation, named the *dependency circuit*, of the function. A *resubstitution*, or simply *substitution*, is a pair composed of a root node and a resubstitution candidate, and it is said to be *legal* if replacing the root node with the candidate does not change the global input-output relation of the logic network. Otherwise, the resubstitution is said to be *illegal*.

3 Related Work

Research in Boolean resubstitution techniques dates back to the 1990s [7, 19]. In the 2000s, efforts were made to improve the scalability of BDD-based computations [8] and to move away from BDDs to simulation and Boolean satisfiability (SAT) [11]. The structural analysis (windowing) was introduced to speed up the algorithm further [16]. In [11], the dependency function is computed by enumerating its onset and offset cubes using SAT. Random simulation is used for initial filtering of resubstitution candidates. In [16], simulation is also used to find potential candidates, which are then checked by SAT solving. The dependency function is computed using interpolation [6]. Windowing is used to limit the search space and the SAT instance size, with the inner window as a working space, and the outer window as the scope for computing don't-cares.

A state-of-the-art Boolean resubstitution algorithm for AIGs using windowing was presented in [12]. It relies entirely on truth table computation, without any use of BDDs or SAT. The search for divisors is limited to a window near the root node. The window inputs are computed as a size-limited reconvergence-driven cut. The node functions in the

window are expressed in terms of the cut variables. The dependency function is not computed as a separate step after minimizing its support, as in [16]. Instead, simple functions up to three AND gates are tried for resubstitution using several heuristic filters. The windowing-based resubstitution framework has been generalized to many different gates types including majority gates [17] and complex gates [1].

Random simulation is a core tool in logic synthesis and has been used successfully to reduce the runtime of various computations. Functional reduction [10], for example, uses random and guided simulation to identify equivalent nodes and merge them. Effective combinational equivalence checking [14] also is used to find cut-points between two networks that serve as stepping stones for the final proof of equivalence at the primary outputs. Motivated by the efficacy of these techniques, the simulation-guided paradigm in this paper focuses on identifying a set of expressive simulation patterns. Once identified, these can be reused multiple times to speed up logic synthesis for the same network in various applications.

4 Framework

We introduce a promising new paradigm for logic synthesis that exploits fast bit-parallel simulation to reduce the number of expensive NP-hard checks, such as those based on SAT. The rationale behind the idea is to pre-compute a set of "expressive" simulation patterns for a given logic network, which can rule out illegal transformations by comparing simulation signatures.

Definition 1. *We call a non-exhaustive set of simulation patterns* expressive *for a logic network if the set can be used to pair-wise distinguish functionally non-equivalent nodes that either already exist in the logic network or can be derived from the existing nodes.*

Obviously, the set of *all* simulation patterns of primary inputs satisfies this definition, but this is typically too large for logic networks with 16 or more primary inputs. In practice, only expressive simulation patterns that can be efficiently stored and simulated using less than, say, a few hundred or thousand bits are of interest.

Assumption 1. *We assume that, for a logic network with N nodes, a set S of expressive simulation patterns with size $|S| \leq C \cdot N$ exists, where C is some constant parameter determined by the structure of the logic network.*

This means that a set of expressive simulation patterns can be pre-computed, stored, and re-used by different logic synthesis engines when applied to the same design, or by the same engine when invoked multiple times. In the rest of this paper, this paradigm is demonstrated using Boolean resubstitution.

The resubstitution framework performs these steps:

1. Generation of a set of expressive simulation patterns. In general, we can start with a set of random patterns, and refine or expand it with the techniques proposed in Section 5.
2. Simulation of the network with these patterns to obtain simulation signatures for each node.
3. Choosing a root node to be substituted. Estimating the gain by computing its MFFC and collecting the divisors. Skipping this node if the gain is too small or if there are no divisors.
4. Searching for resubstitution candidates and the dependency function using simulation signatures. Details of this step are described in Section 6.
5. Validating the resubstitution with SAT solving by assuming non-equivalence. An UNSAT result validates the resubstitution, while a SAT result provides an input assignment under which the substituted network is non-equivalent to the original network. In the latter case, the counter-example is added to the set of simulation patterns.
6. Iterating starting from Step 3, until all nodes in the circuit have been processed.

5 Simulation Pattern Generation

It was observed that expressive simulation patterns cannot be derived directly from the input-output function of the logic network, but must account for some structural information. An intuitive explanation of this may be that an input-output function can be implemented by a large number of structurally different logic networks. This agrees with the idea of re-using simulation patterns in multiple optimization passes because the initial structure of the network often is determined by high-level synthesis and later carefully fine-tuned by logic optimization. Consequently, only a small fraction of closely-related structures is encountered during logic optimization of the network.

Motivated by Assumption 1, we suggest two simple orthogonal strategies for pre-computing simulation patterns:

1. Random patterns: this generates random values for the primary inputs with equal probability of 0 or 1 for each bit.
2. Stuck-at patterns: this iteratively selects nodes and computes patterns that distinguish each from constant functions 0 and 1.

The implementation of the first strategy is straightforward. We describe the second one in the following section. Then, in Section 5.2, we propose an observability-based method to strengthen the computed stuck-at patterns.

5.1 Stuck-at Values

Some nodes in the circuit may rarely produce a value (0 or 1) during random simulation. For example, the output of an AND gate with many fan-ins may be 0 most of the time,

hence 1 is rather rare and may be critical. Thus we refine the set of simulation patterns by checking that every node has both values appearing in its simulation signature. If only one value occurs, a new simulation pattern is created by solving a SAT problem, which forces the node to have the other value.

The algorithm is illustrated in Figure 1. In line `01`, we can either start with an empty set or a random set of simulation patterns. Then, in line `04`, for each node in the circuit, if 0 or 1 does not appear, we try to generate a pattern to express the missing value (lines `05-08`). In an un-optimized circuit, there may be nodes which never take one of the values, so these are replaced by a constant node in line `10`.

We can strengthen the pattern set further by assuring both values appear multiple times (for example, at least 10 times) in the signature of every node. This can be done by running the SAT solver multiple times while making sure it takes different computation paths.

StuckAtCheck
input: a circuit C
output: a set of expressive simulation patterns S
01 S := a small set of random patterns; C.*simulate*(S)
02 **initialize** *Solver*; *Solver.generate_CNF*(C)
03 **foreach** node n in C **do**
04 **if** n.signature = $\vec{0}$ **or** n.signature = $\vec{1}$ **do**
05 **if** n.signature = $\vec{0}$ **do** *Solver.add_assumption*(n)
06 **else do** *Solver.add_assumption*($\neg n$)
07 **if** *Solver.solve*() = SAT **do**
08 $S := S \cup$ {*Solver.pi_values*}
09 **else do**
10 Replace n with constant node.
11 **return** S

Figure 1. Generation of expressive simulation pattern by asserting stuck-at values.

5.2 Observability

As described in Section 2.2, there may be some simulation patterns that are not observable with respect to an internal node; these patterns are deemed less expressive. Here, two cases are identified where a (re-)generation of an observable pattern may be done:

- Case 1: In *StuckAtCheck* when a node is stuck at a value, and a new pattern is generated to express the other value, this pattern is not observable.
- Case 2: A node assumes both values, but for all the patterns under which the node assumes one of the values, it is not observable.

To *resolve* un-observable patterns, a procedure *ObservablePatternGeneration* is devised, which generates an observable simulation pattern $\vec{x}$ with respect to a given node n and makes sure that n expresses a specified value v under $\vec{x}$. This procedure builds a CNF instance, as shown in Figure 2, and solves it using the SAT solver. If the instance is satisfiable, an observable pattern is generated (Claim 1), and we say that the originally un-observable pattern is *resolved*. Else if the solver returns UNSAT, we conclude that value v at node n is not observable. Hence, it can be replaced by the constant node in the respective polarity (Claim 2).

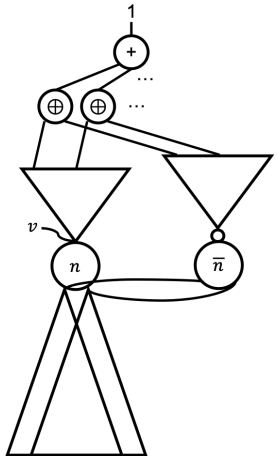

Figure 2. Circuit of the CNF instance built in procedure *ObservablePatternGeneration*. It is constructed by duplicating the TFO cone of n and connecting it to $\overline{n}$. Primary outputs of the two TFO cones are matched and connected to XOR gates ($\oplus$), and the XOR gates are fed to an OR gate (+), whose output is asserted to be 1, forming a miter sub-circuit.

Claim 1. *A satisfying input assignment $\vec{x}$ in the circuit of Figure 2 is an observable pattern with respect to node n.*

Proof. By the definition in Section 2.2, $\vec{x}$ is observable with respect to n if the value of at least one of the primary outputs of the circuit under $\vec{x}$ is different when n is replaced by $\overline{n}$. This condition is ensured by the miter of the TFO cones of n and $\overline{n}$ in Figure 2. ■

Claim 2. *If a node n is never observable with value v ($v \in \{0, 1\}$), then it can be replaced by constant $\neg v$ ($\neg 0 = 1, \neg 1 = 0$) without changing the circuit function(s). That is, there does not exist a primary input assignment $\vec{x}$, such that one of the primary outputs has different values in the original circuit and in the substituted circuit.*

Proof. Assume the opposite: there exists a primary input assignment $\vec{x}$, such that at least one of the primary outputs has a different value after substituting n with $\neg v$. If the value of n is $\neg v$ under $\vec{x}$, all node values in the circuit, including primary outputs, remain unchanged if n is replaced by $\neg v$. If the value of n is v under $\vec{x}$, because n is not observable with v, all primary outputs remain at the same value when the node value of n changes to $\overline{n} = \neg v$, which contradicts the assumption. ■

In order to limit the computation in large circuits, the TFO in Figure 2 can be restricted to nodes within a certain distance from n, called the *depth* of the TFO cone, instead of extending all the way to primary outputs. In this case, all the leaves of the cone should be XOR-ed with their counterparts to build the miter. Using depth of 5 is empirically a good tradeoff between quality and runtime.

After an observable pattern $\vec{x}$ is generated, in Case 1, we can replace the pattern generated by *StuckAtCheck* with $\vec{x}$. In Case 2, we simply add $\vec{x}$ to the set of patterns.

6 Simulation-Guided Resubstitution

When the set of expressive simulation patterns is available, we exploit the simulation signatures of the nodes to implement Boolean resubstitution. The main difference of our algorithm, compared to a state-of-the-art resubstitution algorithm [13], is in the representation of divisors. Instead of using the complete truth table of the *local* function of the node, we use the simulation signature approximating the *global* function of the node.

The resubstitution algorithm is shown in Figure 3. Bit operators ($\sim$, | and &) are implemented using bit-wise operations on simulation signatures. Symbols $\neg$, $\wedge$ and $\vee$ indicate the creation of a complemented wire, an AND gate, and an OR gate (AND gate with the complemented inputs/output), respectively.

In line 03, procedure *collect_divisors* collects potential divisors in the TFI cone of n, excluding nodes in its MFFC or nodes depending entirely on other divisors. This computation can be extended to the whole circuit, excluding only nodes in the TFO of n or in its MFFC. In practice to keep the runtime reasonable, the number of collected divisors is limited.

First, resubstitution is tried without new nodes, as shown in lines 05–09. If n can be expressed directly using a divisor or its negation, the network size is reduced by removing n and its MFFC. Procedure *verify* uses the SAT solver to try to find a pattern, under which nodes n_1 and n_2 have different values. The resubstitution is validated if the solver returns UNSAT (lines 25–26); otherwise, a counter-example is added to the set of simulation patterns (lines 27–29).

If the MFFC is not empty, substituting n with simple functions of two divisors can be tried, which will lead to creating one new node. In the case of an AIG, the divisors are partitioned into positive (P), negative (N) unate divisors and other, by checking the implication relation between their signatures (lines 11–16). Then, in lines 17–22, the signatures are compared to find potential resubstitutions using OR and AND functions. Resubstitution candidates of this type also need to be formally verified.

SimResub
input: a root node n in a simulated circuit C
output: a legal (verified) candidate to substitute n
01 **initialize** *Solver*; *Solver.generate_CNF*(C)
02 *MFFC_size* := |*compute_MFFC*(n)|
03 D := *collect_divisors*(n)
04 **if** $D = \emptyset$ **do return NULL**
05 **foreach** divisor d in D **do** `/* resub-0 */`
06 **if** n.signature = d.signature **do**
07 **if** *Solver.verify*(n, d) **do return** d
08 **if** n.signature = $\sim d$.signature **do**
09 **if** *Solver.verify*(n, $\neg d$) **do return** $\neg d$
10 **if** *MFFC_size* = 0 **do return NULL**
11 $P := \emptyset$; $N := \emptyset$
12 **foreach** divisor d in D **do**
13 **if** d.signature $\rightarrow$ n.signature **do** $P := P \cup \{d\}$
14 **else if** $\sim d$.signature $\rightarrow$ n.signature **do** $P := P \cup \{\neg d\}$
15 **else if** n.signature $\rightarrow$ d.signature **do** $N := N \cup \{d\}$
16 **else if** n.signature $\rightarrow$ $\sim d$.signature **do** $N := N \cup \{\neg d\}$
17 **foreach** pair of divisors d_1, d_2 in P **do** `/* resub-1 */`
18 **if** n.signature = d_1.signature | d_2.signature **do**
19 **if** *Solver.verify*(n, $d_1 \vee d_2$) **do return** $d_1 \vee d_2$
20 **foreach** pair of divisors d_1, d_2 in N **do** `/* resub-1 */`
21 **if** n.signature = d_1.signature & d_2.signature **do**
22 **if** *Solver.verify*(n, $d_1 \wedge d_2$) **do return** $d_1 \wedge d_2$
23 **return NULL**

Solver.Verify
input: two nodes, n_1 and n_2, in a simulated circuit C
output: whether it is legal to substitute n_1 with n_2
24 *Solver.add_assumption*(*literal*(n_1) $\oplus$ *literal*(n_2))
25 **if** *Solver.solve*() = UNSAT **do**
26 **return** TRUE
27 **else**
28 C.*add_pattern*(*Solver.pi_values*)
29 **return** FALSE

Figure 3. Simulation-guided resubstitution.

7 Experimental Results

The pattern generation and the simulation-guided resubstitution framework are implemented in C++-17 as part of the EPFL logic synthesis library *mockturtle*[1]. The experiments are performed on a Linux machine with Xeon 2.5 GHz CPU and 256 GB RAM. The OpenCore designs from IWLS'05 benchmark[2] are used for testing.

In this section, we investigate the expressiveness of simulation patterns generated using different methods and compare their impact on resubstitution. The advantages of the framework are measured in terms of the circuit size reduction and compared against state-of-the-art. Also, quality/speed trade-offs are explored.

[1] github.com/lsils/mockturtle
[2] iwls.org/iwls2005/benchmarks.html

7.1 Size of Simulation Pattern Set

Intuitively, the more simulation patterns used, the higher is the chance that the framework saves time by not attempting to validate illegal resubstitutions, i.e. a larger set of simulation patterns is expected to be more expressive. Following Definition 1 in Section 4, we measure the *expressive power* of a pattern set using the percentage decrease in the number of counter-examples encountered in the resubstitution flow, compared to the baseline set calculated separately for each benchmark. Different from a typical resubstitution flow, the counter-examples are not added to the simulation set, to isolate the impact of the provided patterns.

We start by investigating the expressive power of random patterns based on their count. In Figure 4, each bar represents how expressive is a pattern set of the respective size, compared to the baseline of using only four simulation patterns. The smaller sets are subsets of the larger sets to avoid the biasing effect of randomness. As the size grows by the factor of four (leading to 4, 16, 64, etc patterns), the expressive power increases very fast at first, as expected, but saturates at a few hundreds to a few thousands of patterns. Fortunately, a thousand patterns is still a practical size, for which bit-parallel simulation runs fast.

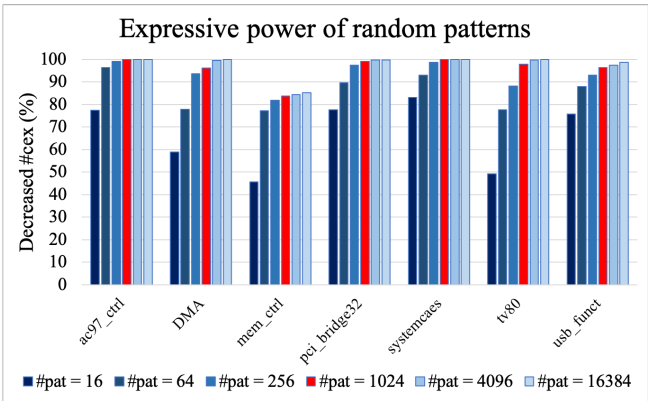

Figure 4. Decreased percentages of counter-examples when provided with different number (#*pat*) of random simulation patterns comparing to #*pat* = 4.

A similar phenomenon is observed when patterns are generated by *StuckAtCheck*. As discussed in Section 5.1, additional patterns can be used to ensure that every node has at least b bits of 0 and b bits of 1 in its signature. In the following experiments, stuck-at patterns are abbreviated as "`s-a`", with a prefix "bx" listing parameter b. Since the stuck-at pattern counts are different for each testcase, the pattern set size is normalized to the circuit size and plotted in the logarithmic scale in Figure 5 and the following figures.

In Figure 5, it is observed that larger sets of patterns are usually more expressive. Note that randomness plays a role in this case, since the default variable polarities, which determine initial variable values in the SAT solver, are randomly reset before each run.

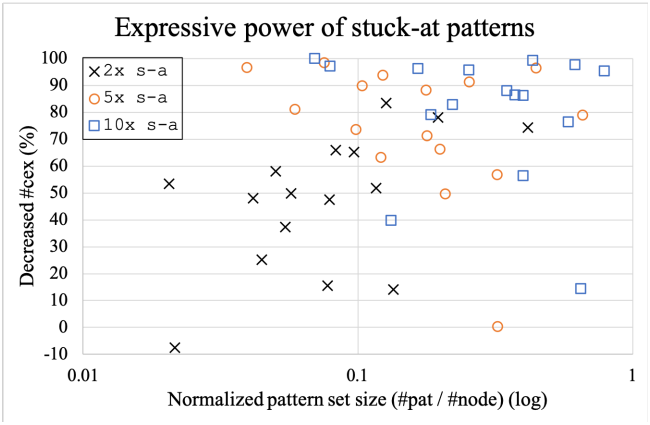

Figure 5. Decreased percentages of counter-examples when using different sets of stuck-at simulation patterns, compared to using the "`1x s-a`" set.

7.2 Pattern Generation Strategies

In this section, the expressive power of simulation patterns generated by *StuckAtCheck* is compared with the case when observability is used (suffix "`-obs`") and/or when an initial random pattern set of size 256 is used (prefix "`rand 256`").

The observability check and observable pattern generation are done with a fan-out depth of 5 levels. A set of 256 random patterns is used as the baseline in Figure 6. Note that there are a few benchmarks, for which the random pattern sets are more expressive than "`1x s-a`" and/or "`1x s-a-obs`". These are not shown in the figure. The geometric means of the sizes of the pattern sets are 143 for "`1x s-a`", 244 for "`1x s-a-obs`", 354 for "`rand 256 + 1x s-a`" and 462 for "`rand 256 + 1x s-a-obs`". On the other hand, the geometric means of the decreased percentages of the counter-examples are 91.3%, 96.5%, 97.1% and 99.5%, respectively.

It is observed that patterns generated by *StuckAtCheck* are usually more expressive than random patterns, except for a few, typically small, benchmarks. Also, using observability increases the expressive power of the generated patterns. Finally, seeding the pattern generation engine with an initial set of random patterns not only speeds up the generation process, but also makes the resulting patterns more expressive.

7.3 Effect of Expressive Patterns in Resubstitution

As stated in the introduction, an expressive set of simulation patterns is used to shift the computation effort from the optimization algorithms to pattern pre-computation. Table 1 shows how the quality of the patterns affects the runtime of pattern generation (*patgen*) and resubstitution (*resub*). A

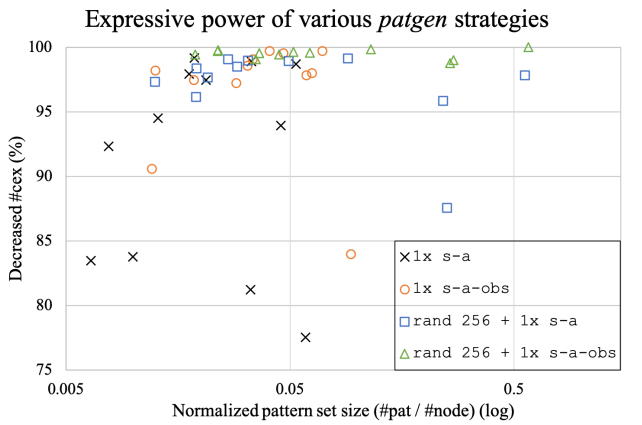

Figure 6. The decrease of percentage of counter-examples when different patterns are used, relative to "`rand 256`". Four benchmarks are excluded because their baseline is more expressive than "`1x s-a`" and/or "`1x s-a-obs`".

better set of patterns (Table 1, "`5x s-a`") efficiently filters out many illegal resubstitutions without calling the SAT solver, resulting in the reduced counter-example counts (#cex) and faster runtimes.

Furthermore, in practice, when the same design is repeatedly synthesized during development or when simulation patterns are reused by different optimization engines, counter-examples from the previous runs can be saved for later use. In this case, additional counter-example count, generated during later runs, can go down to nearly zero, and the runtime is only spent on logic synthesis or verification tasks, such as proving equivalences among the nodes or computing dependency functions and validating them. The latter scheme will be used in the next section.

Table 1. Resubstitution runtime as a function of the number of counter-examples produced.

	`rand 256`			`5x s-a`		
benchmark	#cex	runtime (s)		#cex	runtime (s)	
		patgen	*resub*		*patgen*	*resub*
ac97_ctrl	72	0.01	0.53	47	3.16	0.43
aes_core	50	0.02	2.22	9	3.91	1.95
DMA	1366	0.02	15.30	69	19.43	1.30
mem_ctrl	2737	0.01	16.23	211	5.30	0.81
pci_bridge32	1223	0.02	12.94	174	14.73	2.58
systemcaes	130	0.01	0.69	64	1.93	0.43
usb_funct	1189	0.01	7.84	195	5.49	1.70
tv80	748	0.01	2.75	192	3.43	1.04

7.4 Quality of Simulation-Guided Resubstitution

This section shows the improvements in terms of resubstitution quality. Table 2 compare the proposed framework with command `resub` [12] in ABC [2], which performs truth-table-based resubstitution. Because computing simulation patterns in our framework results in detecting combinational equivalences [10], for a fair comparison, the benchmarks are pre-processed by repeating the command `ifraig` in ABC until no more size reduction is observed. The quality of results is measured using the reduction in circuit size after optimization, presented in the *gain* columns. Simulation patterns used in our framework are initially generated with "`rand 256 + 1x s-a-obs`" and then supplemented with the counter-examples generated from the previous runs of the same experiment.

The maximum cut size K used to collect divisors in the TFI of the root node can be set in both flows. Since [12] relies on computing truth tables in the window, $K \leq 10$ is typically used as a reasonable trade-off between efficiency and quality. In contrast, windowing in our framework is applied only to avoid potential runtime blow-up for large benchmarks, and K can be set to arbitrarily large values when longer runtime is acceptable.

Table 2 shows that our framework achieves a 40.96% improvement on average using the same, small window size, and that we are able to extend the window size and achieve up to 73.82% improvement without runtime overhead.

8 Conclusions and Future Works

The paper presented (1) a simulation-based logic optimization framework, which separates the computation of expressive simulation patterns and their use to validate Boolean optimization choices; (2) methods to increase expressiveness of simulation patterns, resulting in reduced runtime due to fewer SAT calls; (3) improvements to the flexibility of resubstitution candidates, resulting in better optimization quality.

This work has been partially motivated by the success of approximate logic synthesis, when substantial reductions in the circuit size are achieved without expensive Boolean computations, at the cost of introducing some errors into logic functions. We hope that future work in the area of simulation-based methods will help improve speed and quality of both exact and approximate logic synthesis.

In particular, future work may include developing strategies to refine and enhance the generated simulation patterns further, metrics to evaluate and sort the patterns, and methods to compress the pattern set while preserving expressiveness. While resubstitution guided by simulation signatures automatically accounts for satisfiability don't-cares, observability don't-cares can also be considered in the validation of resubstitution, resulting in better quality.

The search space of resubstitution candidates can be extended to include complex dependency functions requiring more than one gate. As shown in Section 7.3, using expressive patterns reduces the chance of encountering counter-examples, making it possible to further reduce the use of SAT solving by validating several resubstitutions at the same time, if almost all of them are legal. To deal with benchmarks containing millions of nodes, incremental CNF construction

Table 2. Quality comparison against ABC with different window sizes.

abc> ifraig until sat.			abc> resub -K 10		Ours, $K = 10$			Ours, $K = 100$		
benchmark	size	#PIs	gain	time (s)	gain	time (s)	improv. (%)	gain	time (s)	improv. (%)
ac97_ctrl	14199	4482	177	0.13	178	0.25	0.56	181	0.26	2.26
aes_core	21441	1319	322	0.42	343	1.18	6.52	496	3.44	54.04
des_area	4827	496	88	0.07	105	0.11	19.32	104	0.55	18.18
DMA	21992	5070	195	0.24	229	0.87	17.44	286	2.05	46.67
i2c	1120	275	48	0.01	57	0.02	18.75	86	0.03	79.17
mem_ctrl	8822	2281	218	0.09	1219	0.27	459.17	1350	1.42	519.27
pci_bridge32	22521	6880	176	0.43	194	0.80	10.23	267	1.42	51.70
sasc	770	250	5	0.01	5	0.01	0.00	5	0.01	0.00
simple_spi	1034	280	18	0.01	17	0.01	-5.56	23	0.01	27.78
spi	3762	505	81	0.06	84	0.08	3.70	89	0.44	9.88
ss_pcm	405	193	1	0.00	1	0.00	0.00	1	0.00	0.00
systemcaes	12108	1600	36	0.11	48	0.19	33.33	55	0.66	52.78
systemcdes	2857	512	138	0.04	155	0.10	12.32	161	0.28	16.67
tv80	9093	732	221	0.14	260	0.34	17.65	451	3.05	104.07
usb_funct	15278	3620	452	0.15	581	1.03	28.54	1199	1.49	165.27
usb_phy	440	211	12	0.00	16	0.01	33.33	16	0.01	33.33
average				0.12		0.33	40.96		0.95	73.82

can be used to limit the size of the SAT instance because too many unrelated clauses slow down SAT solving. Finally, the framework can be extended to optimize mapped networks and to perform other types of Boolean optimization.

Acknowledgments

This work was supported in part by the EPFL Open Science Fund, and by SRC Contract 2867.001, "Deep integration of computation engines for scalability in synthesis and verification".

References

[1] Amaru, L.G., Soeken, M., Vuillod, P., Luo, J., Mishchenko, A., Janet, O., Brayton, R. K., & De Micheli, G. (2018). Improvements to Boolean resynthesis. In *Proceedings of DATE*, (pp. 755–760).

[2] Berkeley Logic Synthesis and Verification Group. ABC: A System for Sequential Synthesis and Verification. http://www.eecs.berkeley.edu/~alanmi/abc/

[3] Brayton, R. K., Hachtel, G. D., & Sangiovanni-Vincentelli, A. L. (1990). Multilevel logic synthesis. In *Proceedings of the IEEE, 78*(2), (pp. 264–300).

[4] Chatterjee, S., Mishchenko, A., Brayton, R., Wang, X., & Kam, T. (2006). Reducing structural bias in technology mapping. *IEEE Transactions on CAD of Integrated Circuits and Systems, 25*(12), (pp. 2894–2903).

[5] Cong, J., & Minkovich, K. (2010). LUT-based FPGA technology mapping for reliability. In *Proceedings of DAC*, (pp. 517–522).

[6] Craig, W. (1957). Linear reasoning: A new form of the Herbrand-Gentzen theorem. *J. Symbolic Logic*, *22*(3), (pp. 250–268).

[7] Kravets, V. N., & Sakallah, K. A. (1998). M32: A constructive multilevel logic synthesis system. In *Proceedings of DAC*, (pp. 336–341).

[8] Kravets, V. N., & Kudva, P. (2004). Implicit enumeration of structural changes in circuit optimization. In *Proceedings of DAC*, (pp. 438–441).

[9] De Micheli, G. (1994). *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education.

[10] Mishchenko, A., Jiang, R., Chatterjee, S., & Brayton, R. (2005). FRAIGs: A unifying representation for logic synthesis and verification. *ERL Technical Report*, EECS Dept., UC Berkeley.

[11] Mishchenko, A., & Brayton, R. (2005). SAT-based complete don't-care computation for network optimization. In *Proceedings of DATE*, (pp. 418–423).

[12] Mishchenko, A., & Brayton, R. (2006). Scalable logic synthesis using a simple circuit structure. In *Proceedings of IWLS*, (pp. 15–22).

[13] Mishchenko, A., Zhang, J. S., Sinha, S., Burch, J. R., Brayton, R., & Chrzanowska-Jeske, M. (2006). Using simulation and satisfiability to compute flexibilities in Boolean networks. *IEEE Transactions on CAD of Integrated Circuits and Systems, 25*(5), (pp. 743–755).

[14] Mishchenko, A., Chatterjee S., Brayton, R., & Een, N. (2006). Improvements to combinational equivalence checking. In *Proceedings of ICCAD*, (pp. 836–843).

[15] Mishchenko, A., Een, E., Brayton, R. K., Jang, S., Ciesielski, M., & Daniel, T. (2010). Magic: An industrial-strength logic optimization, technology mapping, and formal verification tool. In *Proceedings of IWLS*, (pp. 124–127).

[16] Mishchenko, A., Brayton, R., Jiang, J. H. R., & Jang, S. (2011). Scalable don't-care-based logic optimization and resynthesis. *ACM Transactions on Reconfigurable Technology and Systems (TRETS), 4*(4), (pp. 1–23).

[17] Riener, H., Testa, E., Amaru, L. G., Soeken, M., & De Micheli, G. (2018). Size optimization of MIGs with an application to QCA and STMG technologies. In *Proceedings of International Symposium on Nanoscale Architectures*, (pp. 157–162).

[18] Roth, J. P. (1966). Diagnosis of automata failures: A calculus and a method. *IBM journal of Research and Development, 10*(4), (pp. 278–291).

[19] Sato, H., Yasue, Y., Matsunaga, Y., & Fujita, M. (1991). Boolean resubstitution with permissible functions and binary decision diagrams. In *Proceedings of DAC*, (pp. 284–289).

[20] Soeken, M., Riener, H., Haaswijk, W., Testa, E., Schmitt, B., Meuli, G., Mozafari, F., & De Micheli, G. (2019). The EPFL logic synthesis libraries. *arXiv preprint arXiv:1805.05121v2*.