PROMISE: Property Mining for Sequential Synthesis

Jiahui Xu*, Jordi Cortadella†, and Lana Josipović*
*ETH Zurich, Department of Information Technology and Electrical Engineering, Zurich, Switzerland
†UPC Barcelona, Department of Computer Science, Barcelona, Spain

***Abstract*—Modularity—composing a large system using individually designed units—is an essential practice in hardware design. Yet, modularity might compromise quality: when individually designed units are put together, some of their states may become unreachable and, consequently, the logic that implements them is redundant. Sequential synthesis aims to remove redundant circuit logic by leveraging state unreachability. It critically depends on invariants—relations between signals and registers that hold in all reachable states—to prove the validity of redundancies. Yet, existing invariant generation techniques are mostly problem-specific (for a particular circuit or a property) or reliant on localized reasoning. We propose PROMISE, a fast circuit redundancy removal strategy. PROMISE exploits the rich information from simulation traces and uses efficient polynomial-time algorithms to infer global circuit invariants, optimizing the circuit and aiding other sequential synthesis procedures. Experiments show that PROMISE effectively optimizes circuits produced by high-level synthesis tools.**

I. INTRODUCTION

FPGAs offer flexibility, energy efficiency, and high performance; our goal is to make their programming as smooth as traditional software development. The key to devising large designs is modularity: a design is composed of small, manageable pieces that can be easily integrated. Yet, composability comes with a cost. It incurs substantial overhead [1] due to the interface logic required to assemble the modules correctly: when individually designed units are put together, some of their states may become unreachable and, consequently, the logic that implements them is redundant.

Sequential synthesis is a family of logic synthesis techniques that remove redundant circuit logic leveraging state unreachability. Its success critically depends on the available invariants—relations between *flip-flops* (FFs) that hold in all reachable states—to prove the validity of redundancies. However, useful invariants are hard to find: the space of possible properties is enormous, so strategies often rely on localized reasoning (e.g., a subgraph of adjacent circuit gates/FFs) or produce property- or circuit-specific invariants.

We present PROMISE, an invariant generation framework for sequential synthesis. PROMISE leverages the information available in circuit simulation traces and uses efficient polynomial-time algorithms to generate a system of invariants commonly found in circuits produced by *high-level synthesis* (HLS). The invariants characterize the unreachable circuit states, which we use to optimize the circuit's encoding and enhance the effectiveness of existing sequential synthesis approaches. Our result shows that PROMISE-generated invariants bring tangible and justifiable improvements to the circuit quality.

II. BACKGROUND

This section reviews the foundations in formal verification and logic synthesis techniques that PROMISE relies on to optimize redundant circuit logic.

A. Model Checking

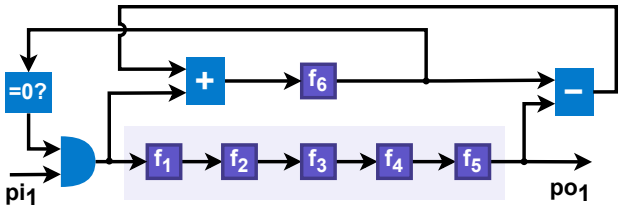
In a *finite-state machine* (FSM), such as a sequential circuit, an *invariant* is a property that holds in every reachable state [2]. Model checking [3]–[5] is a formal verification technique that formally proves whether a certain property holds for an FSM. If the property fails, it provides a counterexample. *k-induction* [6], [7] is an important model checking algorithm for invariant properties. k-induction verifies if the following two conditions hold: (1) the property holds in any k steps starting from the initial state; (2) for any k consecutive states where the property holds, the property holds after any transition. In practice, a very large bound k is needed for concluding non-trivial properties: when k is not big enough, the induction engine will return a counter-example, in which none of the states is reachable. An invariant can act as a constraint during model checking to rule out certain unreachable states (i.e., the model checker ignores the states that violate the invariants), thus it can speed up the verification of the k-induction proof of another safety property [8]–[10].

Model checking algorithms like k-induction not only apply to verifying the circuit's correctness against a certain specification, as we will see in the next section, but they also have important applications in circuit optimization.

B. Sequential Synthesis

Sequential synthesis refers to circuit transformations that change the circuit's combinational function but do not alter the value of the *primary outputs* (POs) in all reachable states [11], [12]. It leverages state unreachability (i.e., some values never appear in the input of the combinational circuit) to uncover optimizations that are unattainable in combinational synthesis [7], [13]: state reachability is formally verified before being applied to reduce the circuit area. This principle can be generalized into a *suggest-guarantee-optimize* procedure independent of the circuit transformation and the property being verified. This procedure is divided into these steps:

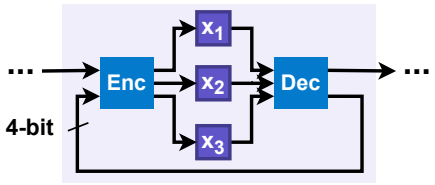
- *Suggest*: Candidate invariants—which indicate state unreachability—are identified using a custom heuristic.
- *Guarantee*: The suggested invariants must be guaranteed by formal verification. The surviving invariants are passed to the *optimize* phase.
- *Optimize*: The circuit is optimized using the unreachable state space generated by the proven invariants.



(a) The unoptimized circuit has *redundant encoding* (the 5 FFs with shaded background is reduced to 3 as in Figure 1c).

Cycle	f_1	f_2	f_3	f_4	f_5	f_6
C_0	0	0	0	0	0	0
C_1	1	0	0	0	0	1
C_2	0	1	0	0	0	1
C_3	0	0	1	0	0	1
C_4	0	0	0	1	0	1
C_5	0	0	0	0	1	1
C_6	0	0	0	0	0	0

(b) A simulation trace of Figure 1a, which is also the same as the entire reachable set of states.



(c) A circuit diagram (some details omitted) after encoding optimization. Section VII-A discusses how the optimization is carried out.

Fig. 1: Circuit with a redundancy encoding. We identify optimization opportunities and invariants from simulation traces.

However, current invariant generation methods (the "suggest" phase) do not synergize well with the "optimize" phase: they overlook encoding optimization opportunities and are unable to find useful invariants for effective sequential synthesis, as we will demonstrate next.

III. Are We Getting the Most Out of Our Invariants?

This section motivates new approaches for circuit encoding optimizations and invariant generation.

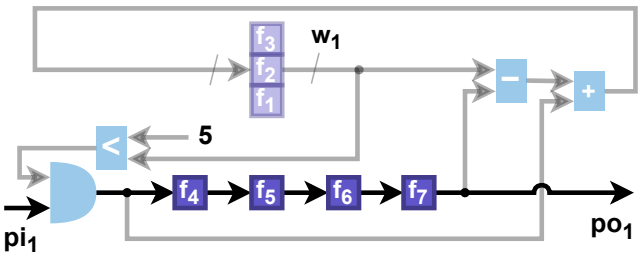
A. Missed Encoding Optimization Opportunities

Figure 1a depicts a sequential circuit with six FFs (f_1–f_6); all initialized to 0. The input to the start of the FF chain (f_1–f_5) can be 1 only if the value of f_6 is 0 in the current state; f_6 becomes one after the start of the chain receives an 1, and becomes 0 after the start of the chain outputs an 1. Figure 1b describes a simulation trace of 7 clock cycles (i.e., 7 circuit states) with the values of the FFs. The circuit has simple control logic, and this trace contains all of its reachable states. Notice that, in any state, at most one FF in f_1–f_5 has an output value equal to 1. In theory, it is possible to replace the chain of 5 FFs with only 3 FFs as in Figure 1c.

This redundancy appears in many circuits and is often *unintentional*. However, state-of-the-art sequential synthesis approaches [7], [13], [14] typically would not optimize away this redundancy, since existing state-encoding approaches require complete reachability information (e.g., state-transition graph, or a BDD of the set of reachable states) that is very expensive to obtain. PROMISE detects these redundancies in the form of a linear inequality[1]:

$$f_1 + f_2 + f_3 + f_4 + f_5 \leq 1. \quad (1)$$

When this relation is true for all reachable states, we know that there are only 6 reachable combinations of these 5 FFs, and PROMISE can reencode the 5 FFs into 3 FFs.



(a) The shaded parts can be removed without compromising the circuit's functionality.

Cycle	pi_1	f_1	f_2	f_3	f_4	f_5	f_6	f_7
C_0	1	0	0	0	0	0	0	0
C_1	1	1	0	0	1	0	0	0
C_2	1	0	1	0	1	1	0	0
C_3	1	1	1	0	1	1	1	0
C_4	0	0	0	1	1	1	1	1
C_5	0	1	1	0	0	1	1	1
C_6	0	0	1	0	0	0	1	1

(b) Simulation trace with 7 states and 7 state variables.

Fig. 2: The invariant extracted from a simulation trace can improve the effectiveness of sequential synthesis (see Section III-B).

B. Localized and Inexpressive Invariants

Consider the circuit in Figure 2a. All FFs are initialized to 0. A chain of FFs f_4–f_7 receives a 1 only when the 3-bit word w_1 (consisting of f_3,f_2, and f_1; f_3 is the MSB) has a value less than 5. Counter w_1 counts up when a 1 is loaded into the FF chain f_4–f_7, and counts down as f_7 outputs a 1. Since the counter's value directly corresponds to the number of 1's in f_4–f_7, the "<" gate always evaluates to 1. Theoretically, we can remove or simplify all the shaded wires without altering the functionality. Without the support of invariants, sequential synthesis approaches (e.g., `scorr` in ABC) require a large induction depth to simplify the circuit (as large as the maximum value of the counter). However, existing invariant generation techniques are unable generate useful invariants for optimizing this circuit, as they are limited to Boolean clauses over a subgraph of adjacent nodes (e.g., a

[1]We use "&", "|", and "∼" to denote logical *and*, *or*, and *not*. We use "+" for arithmetic sum

clause $f_1|f_2|f_3$) [15], simple implication or equality between signals [16], [17], are specialized to particular circuits [8], [9], or specific to the invariants relavant for proving one particular property [18].

On the other hand, PROMISE can detect and prove the following invariant, which cannot be obtained using the aforementioned circuit invariant generation techniques:

$$\underbrace{f_1 + 2^1 \cdot f_2 + 2^2 \cdot f_3}_{=w_1} = f_4 + f_5 + f_6 + f_7, \quad (2)$$

which specifies the relation between the 3-bit word w_1 and the FF chain f_4–f_7. Once assisted with this invariant, sequential synthesis can easily remove all the shaded logic.

The examples above show that current sequential synthesis techniques cannot *suggest* expressive invariants. Without invariants like Equation 1, encoding optimization like the one we describe in Section III-A cannot be carried out.

IV. PROMISE: Generating Circuit Invariants from Simulation

We propose PROMISE, an invariant generation framework to enable more effective sequential synthesis. PROMISE leverages polynomial-time algorithms to efficiently *suggest* candidate linear invariants from the circuit's simulation traces. PROMISE *guarantees* the validity of the invariants using a model checker, and uses the invariants to *optimize* the circuit's encoding or assist other sequential synthesis approaches [7], [19].

Suppose $c_1, \ldots, c_{N+1}$ are coefficients and $f_1, \ldots, f_N$ are the circuit's FFs; PROMISE generates:

- *Inequalities* with {0, -1, +1} coefficient for $c_1, \ldots, c_N$ and arbitrary integer value for c_{N+1}:

$$c_1 \cdot f_1 + c_2 \cdot f_2 + \cdots + c_N \cdot f_N + c_{N+1} \leq 0; \quad (3)$$

- *Equalities* with arbitrary integer coefficients.

$$c_1 \cdot f_1 + c_2 \cdot f_2 + \cdots + c_N \cdot f_N + c_{N+1} = 0. \quad (4)$$

Such linear relations frequently appear in the control logic of HLS-produced circuits [9], since the control status of a circuit is often realized using linearly evolving elements like counters. PROMISE makes better *suggestions* and therefore, enables better *optimizations*.

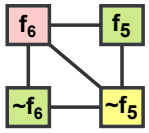
The rest of the paper is organized as follows: Section V describes PROMISE's *suggest* phase: two mathematical methods for inferring the candidate invariants. Section VI describes the *guarantee* phase; a standard model checking algorithm to verify the invariants. Section VII describes the *optimize* phase: how to exploit the properties to optimize the circuit. Section VIII evaluates the effectiveness of PROMISE.

V. The Suggest Phase: Mathematical Methods for Property Mining

This section describes the mathematical methods for identifying properties in the form of linear equalities and inequalities that we classified above.

Cycle	f_5	f_6	$\sim f_5$	$\sim f_6$
C_0	0	0	1	1
$C_{1\ldots5}$	0	1	1	0
C_6	1	1	0	0

(a) f_5, f_6, and their complemented values in Figure 1b.



(b) Conflict graph for Figure 3a colored using 3 colors.

Fig. 3: Coloring a conflict graph using the algorithm in Section V-A.

A. Extracting Mutually Exclusive Sets of Signals by Coloring a Conflict Graph

This subsection devises a systematic strategy for identifying mutually exclusive signals from simulation traces. This enables optimizations such as the one in Figure 1.

A set of FFs $f_1, \ldots, f_N$ are mutually exclusive if they are never simultaneously 1 [20], i.e., $f_1 + \cdots + f_N \leq 1$. Mutual exclusiveness translates to these relations: (1) *One-hot*: mutually exclusive FFs are one-hot. (2) *Implication*: if f_1 and f_2 are two mutually exclusive, they must have $\sim(f_1 \& f_2) = 1$. Now, suppose that f_1 and $\sim f_2$ are mutually exclusive, then $\sim(f_1 \& (\sim f_2)) = (f_1 \rightarrow f_2) = 1$.

PROMISE infers mutually exclusive sets of FFs by *coloring* a *conflict graph* [12], [21]. PROMISE constructs the conflict graph from simulation data as follows:

- For each FF output f_i and its complement $\sim f_i$, add a node in the graph.
- For each simulation cycle, if the corresponding signals of any two nodes are both 1, add an edge between those nodes.
- For each FF f_i, add an edge between f_i and its complement $\sim f_i$ (to avoid a trivial relation like: $\sim f_i + f_i \leq 1$).

Graph coloring assigns different colors to nodes connected by an edge. For scalability, we apply a greedy coloring which runs in linear time [21]; this heuristic leads to excellent results, as we will see in Section VIII. After coloring, each color denotes a set of mutually exclusive FFs. For each color $C := \{f_1, \cdots, f_N\}$, we devise the following invariant:

$$\sum_{f_i \in C} f_i \leq 1, \quad (5)$$

which is used by PROMISE to assist other sequential synthesis approaches and to optimize the encoding—Section VII-A describes how PROMISE carries out the optimization.

For example, Figure 3 describes a conflict graph built from f_5, f_6, and their complemented values. After coloring, we get 3 sets: $\{f_5, \sim f_6\}$, $\{f_6\}$, and $\{\sim f_5\}$. The first set corresponds to a non-trivial relation: $f_5 + \sim f_6 \leq 1$.

B. Extracting Equalities Using Gaussian Elimination

This subsection presents a systematic strategy for deriving a system of linear equalities of the signals in the circuit from simulation traces. This improves the effectiveness of sequential synthesis of the circuit, such as the one in Figure 2.

PROMISE infers linear equalities like Equation 2 from simulation traces. Since any valid invariant holds in all reachable

states, it must be at least valid for the observed simulation data. This requirement is equivalent to a system of linear constraints for possible values of coefficients $c_i \in C$ ($f_1^{s_1}$ denotes the value of f_1 in state s_1):

$$\underbrace{\begin{bmatrix} f_1^{s_1} & f_2^{s_1} & \cdots & f_N^{s_1} & 1 \\ f_1^{s_2} & f_2^{s_2} & \cdots & f_N^{s_2} & 1 \\ \vdots & & & & \\ f_1^{s_M} & f_2^{s_M} & \cdots & f_N^{s_M} & 1 \end{bmatrix}}_{\mathbf{A}:M\times(N+1)} \cdot \underbrace{\begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_N \\ c_{N+1} \end{bmatrix}}_{\mathbf{c}:(N+1)\times 1} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}, \quad (6)$$

which is equivalent to the equations like Equation 4. A solution to the system—a vector $\mathbf{c} = \begin{bmatrix} c_1 & \cdots & c_{N+1} \end{bmatrix}^T$—determines the coefficients of an equality invariant in Equation 4. This system potentially has infinite solutions, that is, infinite equations to be added to the set of invariants. Yet, many equations are redundant (e.g., $2 \cdot f_1 + 2 \cdot f_2 = 2 \cdot f_3$ is just a scaled version of $f_1 + f_2 = f_3$). PROMISE adopts a standard approach [22]–[24] based on Gaussian elimination (has $O(n^3)$ complexity) [25] to determine a minimal set of vectors that each cannot be represented using a linear combination of the others. In this way, PROMISE efficiently infers equations like Equation 2 and uses these equations to aid the sequential synthesis of circuits in Figure 2.

Consider the linear system constructed according to the simulation cycles 0…5 (no cycle 6) in Figure 2b:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_7 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}. \quad (7)$$

One possible set of solutions to this system is:

$$\begin{bmatrix} 1 & 0 & 2 & -1 & 1 & -1 & 1 \end{bmatrix}^T, \quad (8)$$

$$\begin{bmatrix} 0 & 1 & 1 & 0 & -1 & 0 & 0 \end{bmatrix}^T. \quad (9)$$

We plug them separately in Equation 4 (replacing the values of $c_1 \ldots c_7$) to obtain two relations:

$$f_1 + 2 \cdot f_3 + f_5 + f_7 = f_4 + f_6, \quad (10)$$

$$f_2 + f_3 = f_5. \quad (11)$$

These relations hold in cycles 0…5, but the second relation failed in cycle 6. In general, the inferred relation might be spurious since a simulation trace is not guaranteed to cover all states. Therefore, the validity of the invariants must be guaranteed by formal verification, as described next.

VI. The Guarantee Phase: Proving the Suggested Invariants

The result of the previous section is a set of *candidate* invariants inferred from simulation. The simulation traces only contain a subset of the reachable states. PROMISE attempts to verify that the conjunction of the invariants is valid in all reachable states using a model checker. If the proof failed, PROMISE adds the states in the counterexample trace to the set of simulation states to correct the set of invariants. If the invariant holds, we use it to optimize the circuit. For example, the model checker will return a counterexample trace that contains cycle 6 (in Figure 2b) to disprove Equation 11. Having the new state, PROMISE can reexecute the procedure in Section V-B to correctly infer Equation 2.

The verified invariants enable PROMISE to apply the circuit optimization techniques described in the next section.

VII. The Optimize Phase: Capitalizing on the Invariants

This section describes how PROMISE uses the invariants from Section V to optimize the circuits.

A. Applying Encoding Optimizations to the Circuit

This subsection describes how PROMISE performs encoding optimization using inequalities like Equation 3.

In general, for a set of signals $F := \{s_1, \cdots, s_N\}$, a system of inequalities like Equation 3 (that we aim to prove in Section V-A) describes that the sum of the signals is within a set of values $K := \{k_1, \ldots, k_M\}$, that is:

$$(s_1 + \cdots + s_N) \in \{k_1, \ldots, k_M\}. \quad (12)$$

For example, Figure 4a describes a subcircuit with 3 FFs, f_1, f_2, and f_3. Assume that the inequality

$$0 \leq f_1 + f_2 + f_3 \leq 1 \quad (13)$$

holds in all reachable states. Here, $F := \{f_1, f_2, f_3\}$ and $K := \{0, 1\}$. Since there are only 4 possible combinations of the values of f_1, f_2, and f_3, we can substitute the 3 FFs with another subcircuit with 2 FFs as in Figure 4c. The following describes how we construct the substituted circuit.

The substitution must preserve the circuit's functionality. Following a standard pattern as a previous work on encoding optimization [26], PROMISE uses an *encoding circuit* $Enc(\cdot)$ that takes the inputs to the original FFs (e.g., i_1, i_2, i_3 in Figure 4c) and send the encoded inputs (e_1, e_2) to the encoded FFs (x_1, x_2). We use a *decoding circuit* $Dec(\cdot)$ to convert the encoded FF's outputs back to the original outputs (o_1, o_2, o_3). For any reachable FF value assignment to a set of FFs $f_1, \ldots, f_N$, the encoding and decoding function cancel each other's effect, that is, $f_1, \ldots, f_N = Dec(Enc(f_1, \ldots, f_N))$, and the decoding circuit preserves the initial state.

PROMISE uses a *state mapping table* to decide on the encoding scheme and the encoding and decoding circuits. It maps a set of reachable FF values—all combinations that satisfy Equation 12—in the unoptimized circuit into the corresponding FF values in the optimized circuit. Figure 4b describes a state mapping table that maps a state in Figure 4a to a state in Figure 4c. For instance, according to the last entry in Figure 4b, when both the original and optimized circuits start from the initial state, if the original circuit (Figure 4a) reached a state $(f_1, f_2, f_3) = (1, 0, 0)$ after applying certain input sequence, the optimized circuit (Figure 4c) must have

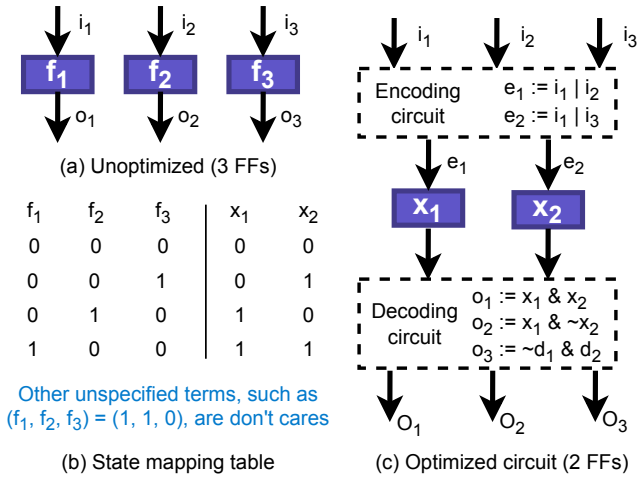


Fig. 4: Circuit substitution: 3 FFs to 2 FFs.

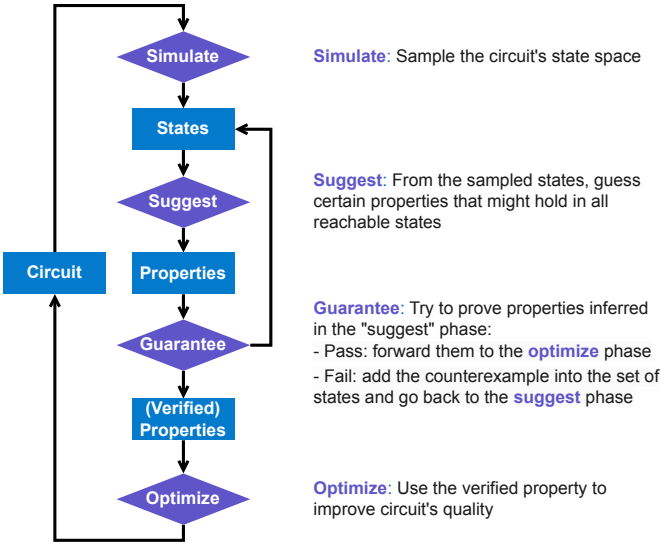


Fig. 5: PROMISE's *suggest-guarantee-optimize* circuit optimization procedure.

$(x_1, x_2) = (1, 1)$. Different mappings potentially have different effects on the cost of the encoding circuit. Although exploring different mappings is beyond the scope of this paper, we will see in Section VIII that the mappings used by PROMISE successfully simplify the circuit.

The encoding and decoding circuits can be built from the state mapping table using a standard logic synthesis technique (e.g., a Karnaugh map). We apply a pre-processing step for FFs with complemented outputs ($\sim f_i$), since they might not exist originally: we insert a pair of inverters at the input and output of each complemented FF ($\sim f_i$), and complement the initial value of that FF [7].

PROMISE performs encoding optimization only for F and K that reduce the number of FFs. The number of FFs in the *substituted subcircuit* $|R_F|$ after applying the encoding optimization is given by

$$|R_F| := \lceil \log_2 (\sum_{k \in K} \binom{|F|}{k}) \rceil, \tag{14}$$

where $\binom{a}{b}$ denotes the number of b-combinations of a elements. The expression in $\log_2(\cdot)$ describes the number of combinations of FF values that sum up to each value in the set $K \in \{k_1, \ldots, k_M\}$. Each combination corresponds to a row in the state mapping table; therefore, the number of FFs needed to represent these states is the $\log_2(\cdot)$ of the number of entries. Encoding optimization is profitable if $|R_F|$ is less than the number of FFs in the original subcircuit $|F|$. In Equation 1, $|F| = 5$ and $K \in \{0, 1\}$, therefore, $|R_F| = \lceil \log_2 (\binom{5}{0} + \binom{5}{1}) \rceil = 3$. Since $|R_F| < |F|$, applying the transformation reduces the number of FFs. Consider another case when $K = \{0, 1, 2, 3\}$. Here, $|R_F| = \lceil \log_2 (1 + 5 + 10 + 10) \rceil = 5$, and therefore the transformation is not desirable.

B. Enhancing Sequential Synthesis Using Invariants

In addition to the encoding optimization above, PROMISE embeds other sequential synthesis procedures in its *optimize* phase and assists them using the invariants in Section V. Without loss of generality, here, we discuss the synergy between PROMISE and one particular sequential synthesis flow [7]; we believe that this also applies to other sequential synthesis approaches (e.g., Marakkalage et al. [14]).

Signal correspondance [7] is a sequential synthesis technique that detects, proves, and merges sequentially equivalent nodes and FFs. This approach relies on k-induction internally to carry out the proof. Without a suitable invariant, the depth k required to prove the property might be prohibitively large. However, in the presence of invariants provided by PROMISE, they can efficiently prove the signal equivalence and perform more effective optimizations.

This concludes the optimization methods used by PROMISE. We now evaluate their effectiveness.

VIII. EVALUATION

This section evaluates the effectiveness of PROMISE.

A. Methodology

We implemented PROMISE—a *suggest-guarantee-optimize* procedure that uses the techniques we have seen so far. Figure 5 describes our invariant generation and circuit optimization flow. Unless stated otherwise, the parameters discussed apply to all experiments.

Benchmarks. We have a set of benchmarks generated using different circuit compilation tools: *Dynamatic* [27] (an MLIR-based HLS tool that converts an input C code to a dynamically scheduled dataflow circuit) and *XLS* [28] (converts a design specified in its input language into a statically scheduled circuit). They are decent targets for evaluating PROMISE: Dynamatic generates circuits by connecting individually designed dataflow units to ensure performance and flexibility,

TABLE I: Effectiveness of **Suggest** and **Optimize**: The features of PROMISE (EN and IN) consistently improve the cost. Compared with SC only, SC + EN + IN achieved average reductions of 31% 6-LUT and 20% FF.

Benchmark	Enabled features			AIG results			6-LUT results		
				FF	AIG	Depth	FF	6-LUT	Depth
factorial (xls)				**37**	974	54	**37**	219	**7**
	SC			**37**	1155	42	**37**	**161**	**7**
	SC	EN		38	1148	**41**	38	164	8
	SC		IN	**37**	**736**	50	**37**	165	**7**
	SC	EN	IN	38	744	50	38	167	**7**
iterative division (xls)				54	528	19	54	132	5
	SC			**53**	515	17	**53**	**116**	**4**
	SC	EN		54	532	17	54	118	**4**
	SC		IN	54	**427**	**15**	54	121	**4**
	SC	EN	IN	55	430	**15**	55	123	**4**
iterative sqrt (xls)				**43**	514	48	**43**	129	**8**
	SC			**43**	600	**35**	**43**	**122**	**8**
	SC	EN		**43**	595	38	**43**	**122**	**8**
	SC		IN	**43**	**450**	46	**43**	125	**8**
	SC	EN	IN	45	455	46	45	128	**8**
simple loop (xls)				34	243	**13**	34	74	4
	SC			**32**	195	**13**	**32**	**57**	**3**
	SC	EN		**32**	209	**13**	**32**	58	**3**
	SC		IN	**32**	**173**	**13**	**32**	58	**3**
	SC	EN	IN	34	177	**13**	34	61	**3**
factorial (Dynamatic)				497	3908	34	497	982	7
	SC			482	4495	33	482	917	7
	SC	EN		443	4596	38	443	955	10
	SC		IN	367	**1603**	**25**	367	**525**	**5**
	SC	EN	IN	**360**	1606	**25**	**360**	528	**5**
iterative division (Dynamatic)				392	2809	27	392	752	7
	SC			311	2405	26	311	544	6
	SC	EN		**134**	876	17	**134**	**197**	**4**
	SC		IN	235	**832**	**15**	235	297	**4**
	SC	EN	IN	204	865	**15**	204	269	**4**
iterative sqrt (Dynamatic)				605	4732	46	605	1225	11
	SC			581	4944	42	581	1087	10
	SC	EN		**191**	1092	**22**	**191**	**269**	5
	SC		IN	288	**829**	23	288	334	**4**
	SC	EN	IN	254	840	23	254	302	**4**
simple loop (Dynamatic)				57	385	18	57	98	4
	SC			45	359	17	45	87	4
	SC	EN		**23**	181	**12**	**23**	**42**	**3**
	SC		IN	28	**178**	15	28	47	**3**
	SC	EN	IN	29	**178**	15	29	48	**3**
matvec (Dynamatic)				262	1997	34	262	587	8
	SC			240	2128	29	240	543	7
	SC	EN		**185**	1750	30	**185**	439	7
	SC		IN	194	**927**	**25**	194	354	**5**
	SC	EN	IN	190	929	**25**	190	**350**	**5**
bicg (Dynamatic)				393	3001	34	393	853	8
	SC			366	3481	31	366	773	7
	SC	EN		295	2875	37	295	671	8
	SC		IN	276	1244	**25**	276	524	**5**
	SC	EN	IN	**269**	**1243**	**25**	**269**	**520**	**5**
gaussian (Dynamatic)				460	3694	35	460	1027	8
	SC			405	3733	38	405	883	7
	SC	EN		312	2974	33	312	656	6
	SC		IN	310	**1183**	**26**	310	459	**5**
	SC	EN	IN	**297**	1231	**26**	**297**	**449**	**5**
gemver (Dynamatic)				1405	10881	36	1405	2955	8
	SC			1285	11793	37	1285	2685	7
	SC	EN		**911**	8669	34	**911**	1851	7
	SC		IN	995	**4342**	**26**	995	1607	**5**
	SC	EN	IN	945	4353	**26**	945	**1561**	**5**
stencil 2d (Dynamatic)				407	3067	32	407	887	8
	SC			363	3021	31	363	782	6
	SC	EN		279	2458	32	279	613	8
	SC		IN	278	**1189**	**26**	278	466	**5**
	SC	EN	IN	**269**	1207	**26**	**269**	**458**	**5**
2mm (Dynamatic)				1117	8912	36	1117	2397	9
	SC			1030	9350	34	1030	2187	7
	SC	EN		746	7139	37	746	1520	8
	SC		IN	785	**3352**	**26**	785	1194	**5**
	SC	EN	IN	**742**	3469	**26**	**742**	**1178**	**5**

but suffers from the area overhead due to this modularity. Similarly, XLS's domain-specific language eases circuit design by abstracting low-level details, but it can introduce redundancy. Each benchmark without memory accesses (XLS cannot handle memory accesses) has two functionally identical implementations in two tools. We also include several standard HLS benchmarks (*matvec, bicg, gemver, kernel 2mm, stencil 2d*) [29]. For each synthesized top-level module, we added a wrapper to simplify the communication between the circuit and its environment. Each module includes a "*go*" input pin to start the execution and a "*done*" output pin that signals when the computation is complete. All designs have been functionally verified using a set of representative input vectors.

Simulate. We synthesize the circuit using Yosys [30]. We use ModelSim [31] to simulate each circuit with random inputs for 4 rounds, each for 25000 cycles.

Suggest. We use the algorithms described in Section V to infer invariants from the set of simulated states. We only extract relations between the FFs that dictate the control status in the design. We use a greedy graph coloring heuristic in the NetworkX library [32] to generate mutual exclusion relations described in Section V-A.

Guarantee. We use the rIC3 model checker [33] to carry out the formal verification of the conjunction of the invariants inferred in the *suggest* phase. rIC3 uses the PDR model checking algorithm [18], [34]. If rIC3 reports that the property is invalid, the states in the counterexample will be added to the set of simulated states. We iterate between the *suggest* and *guarantee* phases until rIC3 confirms that the conjunction of the invariants holds in the circuit.

Optimize. Our baselines are the unoptimized circuit and the circuit optimized only with the `scorr` command (the signal correspondence optimization) available in ABC. We then complement the `scorr` command with our encoding optimization (Section VII-A) and invariants (Section V).

We need to specify our invariants when running the `scorr` command: `scorr` allows declaring a certain PO as a constraint; during the proving step, the algorithm ignores the states where the constraint fails. We construct a logic cone as a constraint from the conjunction of the invariants. We strip away the logic cone after the optimization.

Our optimization metrics. To report the area results, we use ABC to convert the circuit to an AIG network using the `st` command and map the circuit to an FPGA LUT network using `if -K 6`. As with any sequential synthesis optimization, we do not alter the circuit's latency (i.e., the clock cycle count); therefore, we do not report it, as it remains consistent across all designs. Whenever the circuit complexity permits, we apply sequential equivalence checking to formally verify that our modifications preserve the behaviors of the original circuits.

B. *Effectiveness of* PROMISE*: Suggest and Optimize*

Table I reports the ABC synthesis results of PROMISE. The columns grouped with "**Enabled features**" indicate which optimization techniques are applied: Column **EN** indicates whether encoding optimization is applied, **SC** indicates whether signal correspondence [7] (`scorr` in ABC) is applied, and **IN** indicates whether the invariants are used in `scorr`. The columns **AIG results** and **6-LUT results** report the achieved area and delay. The best synthesis result of each single benchmark is highlighted in green.

Design with a network of modular units. The benchmarks labeled with "(Dynamatic)" are generated by Dynamatic [27]. Dynamatic implements handshake modules at the operation level (e.g., a multiplier has its handshake interface) to ensure

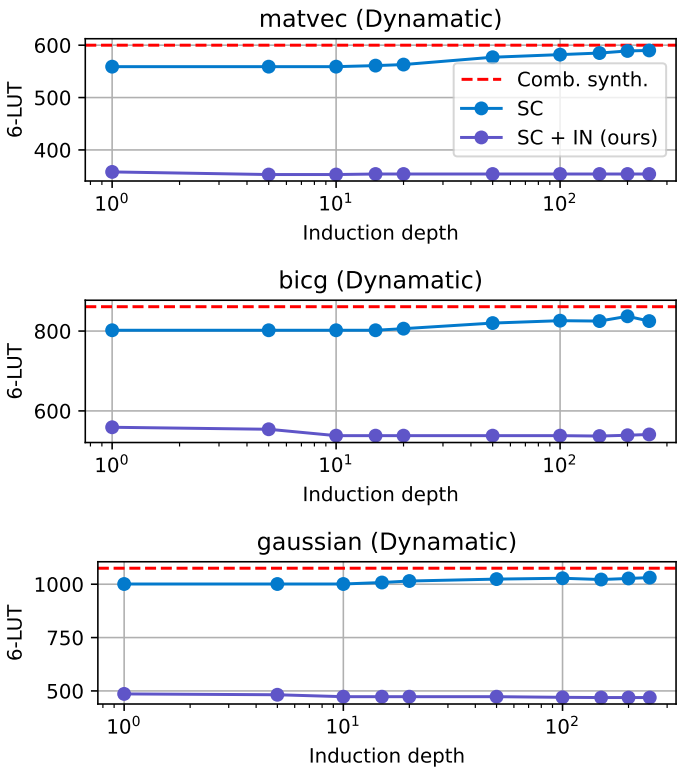


Fig. 6: Effect of using *different induction depths* and *inclusion or exclusion of invariants* when using `scorr` in ABC.

the best composability and latency [35], but has a very large resource overhead [1]. In these benchmarks, signal correspondence alone (i.e., the rows with only **SC**) sees improvement over pure combinational synthesis (i.e., the rows with no techniques). On the other hand, we see a substantial logic reduction when signal correspondence is used with our additions—either encoding optimization (**EN**) or invariants (**IN**); in each benchmark, the best metric is achieved by applying at least one of our optimizations (i.e., **EN** or **IN**).

Design with a single module. The benchmarks labeled "(xls)" are generated by XLS [28]. As a single-module design, the circuit has less redundancy to explore, but invariants are still helpful in the signal correspondence procedure.

Encoding optimization vs. invariant-enhanced signal correspondence. Generally, it is expected to see that encoding optimization reduces the FF count at the cost of a more complex logic. However, circuits with **SC + EN** consistently have fewer AIG or 6-LUT nodes than those with **SC** alone. This is because reducing the FFs also reduces the combinational logic's primary inputs and outputs, which potentially simplifies the logic function. While encoding optimization might increase the maximum logic depth, this effect is offset by the logic saving enabled by the invariants—the rows with **SC + EN + IN** always have the best logic depth.

Increasing induction depth vs. using invariants. Figure 6 describes the effect of inclusion and exclusion of invariants and varying the induction depth (1 to 250) when using `scorr` for benchmarks matvec and bicg. In both benchmarks, increasing the induction depth alone does not improve the area. Surprisingly, the number of 6-LUTs after applying `scorr` *increases* when the induction depth is large. This shows that PROMISE-generated invariants greatly improve the effectiveness of `scorr`.

TABLE II: For benchmark "simple loop (Dynamatic)", comparing the effectiveness of PROMISE's invariants vs. using the reachable set of states as an invariant.

	AIG results			LUT results		
Invariants	**FF**	**AIG**	**Depth**	**FF**	**LUT**	**Depth**
No invariants	45	359	17	45	87	4
Reachable states	57	256	21	57	101	5
Promise's invariants	29	178	15	29	48	3

TABLE III: Scalability of **Suggest** and **Guarantee**

XLS-Produced Circuits

Benchmark	Sim	Proof	Linear equality	Mutual exclusion	Iter.
factorial	17.3	0.1	0.0	0.0	0
iterative division	25.4	0.1	0.0	0.0	0
iterative sqrt	20.6	0.2	0.0	0.0	1
simple loop	17.3	0.1	0.0	0.0	0

Dynamatic-Produced Circuits

Benchmark	Sim	Proof	Linear equality	Mutual exclusion	Iter.
factorial	237.8	1.0	0.4	0.0	1
iterative division	306.9	0.2	0.2	0.0	0
iterative sqrt	350.8	0.4	0.3	0.0	0
simple loop	40.6	0.1	0.0	0.0	0
matvec	257.1	1.6	0.2	0.0	0
bicg	348.8	4.1	0.3	0.0	0
gaussian	365.7	5.7	1.1	0.0	0
gemver	827.1	60.0	4.0	0.0	0
stencil 2d	340.6	12.5	0.7	0.0	0
2mm	700.4	201.3	7.2	0.0	0

The runtimes for simulation (Sim), model checking (Proof), and invariant generation (Linear equality and Mutual exclusion) are measured in seconds.

Effectiveness of PROMISE's invariants vs. reachability. BDD-based reachability analysis produces a set of reachable states—this can be formulated as an invariant (i.e., given the set of all reachable states $\{s_1, s_2, \ldots, s_N\}$, we can format them as a invariant: $(\text{state} = s_1)|(\text{state} = s_2)|\cdots(\text{state} = s_N)$). Table II reports the synthesis result of the benchmark "simple loop (Dynamatic)". The table reports the optimization result after using `scorr`, assisted by no invariants (No invariants), the set of reachable states as an invariant (Reachable states), and PROMISE's invariants (PROMISE's invariants). From the result, using the set of all reachable states as an invariant is less effective than our invariants.

We omit the further comparison with encoding optimization that requires a complete set of reachable states (e.g., Sentovich et al. [26]) due to the poor scalability of the reachability analysis [9]. For one of our medium-sized benchmarks, matvec (Dynamatic), the reachability analysis in ABC could not converge after 48 hours.

C. *Scalability of* PROMISE*: Suggest and Guarantee*

Table III reports the runtime statistics of the property mining procedures (Section V) and time needed to prove the properties (Section VI). Column **Iter.** reports the number of failed proof attempts (that trigger re-execution of the **suggest** phase).

Columns **Mutual exclusion** and **Linear equality** report the total runtime of the techniques introduced in Section V-A and Section V-B. Column **Proof** reports the total runtime of running the rIC3 model checker. Column **Sim** reports the total time spent on circuit simulation. All model checking runs converge in a reasonable time without any abstraction technique applied (often necessary for model checking to converge [1], but not needed in our experiments). The gate-level simulation (event-driven) takes significantly longer to run compared to model checking. This runtime can be greatly reduced by switching to a cycle-accurate simulator.

Scalability of property mining. In benchmarks with data-dependent control flow (e.g., the "factorial" benchmarks generated using Dynamatic), few corrections were done (see Section VI) before our property mining procedure could infer a verifiable invariant. Yet, our property mining procedures are scalable: most property generation takes less than 1 second. All the mutual exclusion properties are generated within 0.1 seconds. The largest benchmark "kernel 2mm (Dynamatic)" only requires 7.2 seconds to generate properties.

IX. Related Work

Dataflow design produced from high-level languages. There has been an increasing interest in HLS tools to produce dataflow processing networks [27], [28], [35]–[41]. Dataflow processing systems are composable and deliver high performance due to their dynamic nature [42]. This paradigm comes with a resource overhead (up to 50% of the dataflow circuit logic are redundant bypassing multiplexers and buffer slots that are never occupied with valid data [1]) and many research works aim at removing it [1], [35], [43]–[46] in them. Unlike these efforts, PROMISE offers a more general solution, independent of specific circuit generation methods.

Encoding optimization. Sentovich et al. [26] focus on optimizing encoding in circuits generated from ESTEREL. They greedily remove one register at a time and require computation and analysis of the entire reachable state space. Computing the reachable state space is typically impractical without problem-specific abstractions. Empirical evidence suggests that induction in the presence of invariants is more scalable than reachability analysis [9]. We efficiently infer encoding optimization opportunities using simulation. Sentovich et al. [20] also describe an FF optimization approach that leverages the knowledge of their circuits; yet, their technique is specific to ESTEREL-produced circuits.

Redundancy removal in sequential circuits. Sequential synthesis techniques like those of Mishchenko et al. [7] and Marakkalage et al. [14] remove redundancy using induction; yet, they do not take advantage of any invariants. Many research efforts explore redundancy removal approaches in a limited setting, such as combinational circuits [47], feedback-free circuits, and a particular redundancy structure [1], [48]. These approaches do not aim to improve the circuits' encoding, and our invariants can be used to improve the effectiveness of their optimization.

Generating invariants for circuits. There exist techniques for automatically deriving inductive invariants for dataflow circuits [8], [9] to improve verification runtime, but they are limited to a set of predefined units. There is a family of model checking algorithms that aim to synthesize an inductive invariant to prove the safety property [18], [34], [49], [50]. These methods are specific to generating an inductive invariant for a single safety property, operate in unreachable states, and the operatorsl used to construct the invariants have either too limited expressivity (i.e., pure Boolean formulas [18]) or are too general and overfit the observed states [49] (i.e., any first-order logic operator). Our method operates on simulation traces, and we aim to infer expressions (e.g., Equation 3 and Equation 4) that commonly appear in the control logic of the circuits generated from hardware compilers.

Property generation from simulation. Using simulation—referred to as *dynamic analysis* in software engineering—to derive loop invariants has been studied in the software verification domain [22]–[24], [51]. Empirical results show that these invariants can support proving the equivalence between the program before and after certain optimization [51]. Execution traces, from both software and hardware, have also been leveraged to infer temporal specifications [52], [53]; they generate more expressive properties than ours (e.g., LTL formulas instead of invariants). However, these properties are not readily applicable to circuit optimization. Inspired by these works, we adopted a similar insight—using simulation to support verification—but specifically for circuit optimization.

X. Conclusion

We presented PROMISE, a framework that utilizes simulation data to detect redundancy in the sequential circuit's state encoding and extract invariants to speed up the circuit verification. PROMISE efficiently detects state encoding optimization opportunities in cases where conventional techniques are prohibitively expensive, and derives linear invariants to make existing sequential synthesis procedures fundamentally more effective. PROMISE synergizes comprehensive simulation-based testing, formal verification, and logic synthesis to uncover new opportunities for more effective and scalable optimization.

Acknowledgement

This work has been supported by the Swiss National Science Foundation (grant number 215747) and the ETH Future Computing Laboratory (donation from Huawei Technologies).

References

[1] J. Xu, E. Murphy, J. Cortadella, and L. Josipović, "Eliminating excessive dynamism of dataflow circuits using model checking," in *Proceedings of the 31st ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterey, CA, Feb. 2023, pp. 27–37. [Online]. Available: https://doi.org/10.1145/3543622.3573196.

[2] S. Chaki and A. Gurfinkel, "BDD-based symbolic model checking," in *Handbook of Model Checking*. Springer International Publishing, 2018, pp. 219–245. [Online]. Available: https://doi.org/10.1007/978-3-319-10575-8_8.

[3] E. Clarke, K. McMillan, S. Campos, and V. Hartonas-Garmhausen, "Symbolic model checking," in *Proceedings of the 8th International Conference on Computer Aided Verification*, New Brunswick, NJ, Jun. 1996, pp. 419–22. [Online]. Available: https://doi.org/10.1007/3-540-61474-5_93.

[4] E. M. Clarke, T. A. Henzinger, and H. Veith, "Introduction to model checking," in *Handbook of Model Checking*. Springer International Publishing, 2018, pp. 1–26. [Online]. Available: https://doi.org/10.1007/978-3-319-10575-8_1.

[5] C. Kern and M. R. Greenstreet, "Formal verification in hardware design: A survey," *ACM Transactions on Design Automation of Electronic Systems*, vol. 4, no. 2, pp. 123–93, Apr. 1999. [Online]. Available: https://doi.org/10.1145/307988.307989.

[6] M. Sheeran, S. Singh, and G. Stålmarck, "Checking safety properties using induction and a SAT-solver," in *Proceedings of the 3rd International Conference on Formal Methods in Computer-Aided Design*, Austin, TX, 2000, pp. 127–144. [Online]. Available: https://doi.org/10.1007/3-540-40922-X_8.

[7] A. Mishchenko, M. Case, R. Brayton, and S. Jang, "Scalable and scalably-verifiable sequential synthesis," in *Proceedings of the 27th International Conference on Computer-Aided Design*, San Jose, CA, Nov. 2008, pp. 234–241. [Online]. Available: https://doi.org/10.1109/ICCAD.2008.4681580.

[8] S. Chatterjee and M. Kishinevsky, "Automatic generation of inductive invariants from high-level microarchitectural models of communication fabrics," *Formal Methods in System Design*, vol. 40, pp. 147–69, 2012. [Online]. Available: https://doi.org/10.1007/s10703-011-0134-0.

[9] J. Xu and L. Josipović, "Automatic inductive invariant generation for scalable dataflow circuit verification," in *Proceedings of the 42nd International Conference on Computer-Aided Design*, San Francisco, CA, Oct. 2023, pp. 1–9. [Online]. Available: https://doi.org/10.1109/ICCAD57390.2023.10323796.

[10] C. A. Furia, B. Meyer, and S. Velder, "Loop invariants: Analysis, classification, and examples," *ACM Computing Surveys*, vol. 46, no. 3, Jan. 2014. [Online]. Available: https://doi.org/10.1145/2506375.

[11] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Sequential circuit design using synthesis and optimization," in *Proceedings 1992 IEEE International Conference on Computer Design*, Cambridge, MA, Oct. 1992, pp. 328–33. [Online]. Available: https://doi.org/10.1109/ICCD.1992.276282.

[12] G. De Micheli, *Synthesis and Optimization of Digital Circuits*. New York: McGraw-Hill, 1994.

[13] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Proceedings of the 22nd International Conference on Computer Aided Verification*, Edinburgh, Jul. 2010, pp. 24–40. [Online]. Available: https://doi.org/10.1007/978-3-642-14295-6_5.

[14] D. S. Marakkalage, E. Testa, W. L. Neto, A. Mishchenko, G. De Micheli, and L. Amarù, "Scalable sequential optimization under observability don't cares," in *Proceedings of 2024 Design, Automation & Test in Europe Conference & Exhibition*, Valencia, Spain, Mar. 2024. [Online]. Available: https://doi.org/10.23919/DATE58400.2024.10546595.

[15] M. Case, A. Mishchenko, and R. Brayton, "Cut-based inductive invariant computation," in *Proceedings of the 17th International Workshop on Logic Synthesis*, Lake Tahoe, CA, Jun. 2008, pp. 253–58. [Online]. Available: https://people.eecs.berkeley.edu/~alanmi/publications/2008/iwls08_ind.pdf.

[16] G. Cabodi, S. Nocco, and S. Quer, "Strengthening model checking techniques with inductive invariants," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 1, pp. 154–58, Jan. 2009. [Online]. Available: https://doi.org/10.1109/TCAD.2008.2009147.

[17] C. van Eijk, "Sequential equivalence checking based on structural similarities," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 7, pp. 814–819, Jul. 2000. [Online]. Available: https://doi.org/10.1109/43.851997.

[18] A. R. Bradley, "SAT-based model checking without unrolling," in *Proceedings of the 12th International Workshop on Verification, Model Checking, and Abstract Interpretation*, Austin, TX, 2011, pp. 70–87. [Online]. Available: https://doi.org/10.1007/978-3-642-18275-4_7.

[19] *ABC: System for sequential logic synthesis and formal verification*, Commit: ca78f5e, berkeley-abc. [Online]. Available: https://github.com/berkeley-abc/abc/tree/ca78f5e6e5308df420ffc5c709e6d37caf97e40b.

[20] E. M. Sentovich, H. Toma, and G. Berry, "Efficient latch optimization using exclusive sets," in *Proceedings of the 34th Annual Design Automation Converence*, Anaheim, CA, Jun. 1997. [Online]. Available: https://doi.org/10.1145/266021.266026.

[21] *Graph coloring*. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Graph_coloring&oldid=1279292851.

[22] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," in *Proceedings of the 21st International Conference on Software Engineering*, Los Angeles, CA, May 1999, pp. 213–224. [Online]. Available: https://doi.org/10.1145/302405.302467.

[23] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest, "Using dynamic analysis to discover polynomial and array invariants," in *Proceedings of 34th International Conference on Software Engineering*, Zurich, Switzerland, Jun. 2012, pp. 683–93. [Online]. Available: https://doi.org/10.1109/ICSE.2012.6227149.

[24] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. V. Nori, "A data driven approach for algebraic loop invariants," in *Proceedings of 22nd European Symposium on Programming*, Rome, Italy, Mar. 2013, pp. 574–92. [Online]. Available: https://doi.org/10.1007/978-3-642-37036-6_31.

[25] *Kernel (linear algebra)*. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Kernel_(linear_algebra)&oldid=1261076439#Computation_by_Gaussian_elimination.

[26] E. M. Sentovich, H. Toma, and G. Berry, "Latch optimization in circuits generated from high-level descriptions," in *Proceedings of the 15th International Conference on Computer-Aided Design*, San Jose, CA, Nov. 1996, pp. 428–35. [Online]. Available: https://doi.org/10.1109/ICCAD.1996.569833.

[27] *Dynamatic*, Commit: 999dc3c, EPFL-LAP. [Online]. Available: https://github.com/EPFL-LAP/dynamatic/tree/999dc3ce2fb95eac1dd39cad441fbdf6b8389aee.

[28] *Xls: Accelerated hw synthesis*, Google, Inc. [Online]. Available: https://github.com/google/xls.

[29] L.-N. Pouchet, *Polybench: The polyhedral benchmark suite*, 2012. [Online]. Available: https://sourceforge.net/p/polybench/wiki/Home/.

[30] *Yosys Open SYnthesis Suite*, Commit: 29cf4a9. [Online]. Available: https://github.com/YosysHQ/yosys/tree/29cf4a919062fe7b6a6f21b946dbec15a3d2114a.

[31] Mentor Graphics, *ModelSim*, 2016. [Online]. Available: https://www.mentor.com/products/fv/modelsim/.

[32] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using NetworkX," in *Proceedings of the 7th Python in Science Conference*, Pasadena, CA, Aug. 2008, pp. 11–15. [Online]. Available: https://www.osti.gov/biblio/960616.

[33] Y. Su, Q. Yang, Y. Ci, T. Bu, and Z. Huang, "The rIC3 hardware model checker," *arXiv preprint arXiv:2502.13605*, Feb. 2025.

[34] N. Een, A. Mishchenko, and R. Brayton, "Efficient implementation of property directed reachability," in *Proceedings of 14th International Conference on Formal Methods in Computer-Aided Design*, Austin, TX, Oct. 2011, pp. 125–34. [Online]. Available: https://dl.acm.org/doi/10.5555/2157654.2157675.

[35] L. Josipović, R. Ghosal, and P. Ienne, "Dynamically scheduled high-level synthesis," in *Proceedings of the 26th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterey, CA, Feb. 2018, pp. 127–36. [Online]. Available: https://doi.org/10.1145/3174243.3174264.

[36] L. Josipović, A. Guerrieri, and P. Ienne, "Dynamatic: From C/C++ to dynamically scheduled circuits," in *Proceedings of the 28th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Seaside, CA, Feb. 2020, pp. 1–10. [Online]. Available: https://doi.org/10.1145/3373087.3375391.

[37] *Vivado design suite user guide: High-level synthesis*, Xilinx Inc., 2018. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug902-vivado-high-level-synthesis.pdf.

[38] Y. Chi, L. Guo, J. Lau, Y.-k. Choi, J. Wang, and J. Cong, "Extending high-level synthesis for task-parallel programs," in *Proceedings of the 29th IEEE Symposium on Field-Programmable Custom Computing Machines*, Orlando, FL, May 2021, pp. 204–213. [Online]. Available: https://doi.org/10.1109/FCCM51124.2021.00032.

[39] L. Guo, Y. Chi, J. Wang, *et al.*, "AutoBridge: Coupling coarse-grained floorplanning and pipelining for high-frequency HLS design on multi-die FPGAs," in *Proceedings of the 29th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Virtual Event, Mar. 2021, pp. 81–92. [Online]. Available: https://doi.org/10.1145/3431920.3439289.

[40] L. Guo, P. Maidee, Y. Zhou, *et al.*, "RapidStream: Parallel physical implementation of FPGA HLS designs," in *Proceedings of the 30th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Virtual Event, Feb. 2022, pp. 1–12. [Online]. Available: https://doi.org/10.1145/3490422.3502361.

[41] A. Elakhras, A. Guerrieri, L. Josipović, and P. Ienne, "Unleashing parallelism in elastic circuits with faster token delivery," in *Proceedings of the 32nd International Conference on Field-Programmable Logic and Applications*, Belfast, UK, Aug. 2022, pp. 253–61. [Online]. Available: https://doi.org/10.1109/FPL57034.2022.00046.

[42] A. Elakhras, A. Guerrieri, L. Josipović, and P. Ienne, "Survival of the fastest: Enabling more out-of-order execution in dataflow circuits," in *Proceedings of the 32nd International Symposium on Field-Programmable Gate Arrays*, Monterey, CA, Mar. 2024, pp. 44–54. [Online]. Available: https://doi.org/10.1145/3626202.3637556.

[43] J. Xu and L. Josipović, "Suppressing spurious dynamism of dataflow circuits via latency and occupancy balancing," in *Proceedings of the 32nd ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterey, CA, Mar. 2024, pp. 188–98. [Online]. Available: https://doi.org/10.1145/3626202.3637570.

[44] R. Nigam, S. Thomas, Z. Li, and A. Sampson, "A compiler infrastructure for accelerator generators," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, Virtual, Apr. 2021, pp. 804–17. [Online]. Available: https://doi.org/10.1145/3445814.3446712.

[45] A. Elakhras, J. Xu, M. Erhart, P. Ienne, and L. Josipović, "ElasticMiter: Formally verified dataflow circuit rewrites," in *Proceedings of the 30th International Conference on Architectural Support for Programming Languages and Operating Systems*, Rotterdam, The Netherlands, Apr. 2025, pp. 293–308. [Online]. Available: https://doi.org/10.1145/3676641.3715993.

[46] L. Josipović, A. Marmet, A. Guerrieri, and P. Ienne, "Resource sharing in dataflow circuits," in *Proceedings of the 30th IEEE Symposium on Field-Programmable Custom Computing Machines*, New York, May 2022, pp. 1–9. [Online]. Available: https://doi.org/10.1109/FCCM53951.2022.9786084.

[47] 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. 2573–86, May 2022. [Online]. Available: https://doi.org/10.1109/TCAD.2021.3108704.

[48] Q. Tan, A. Gupta, and S. Malik, "Usage-based RTL subsetting for hardware accelerators," in *Proceedings of the 41st International Conference on Computer-Aided Design*, San Diego, CA, Dec. 2022, pp. 1–9. [Online]. Available: https://doi.org/10.1145/3508352.3549391.

[49] A. Goel and K. Sakallah, "AVR: Abstractly verifying reachability," in *Proceedings of 26th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Dublin, Ireland, Apr. 2020, pp. 413–22. [Online]. Available: https://doi.org/10.1007/978-3-030-45190-5_23.

[50] M. L. Case, A. Mishchenko, and R. K. Brayton, "Automated extraction of inductive invariants to aid model checking," in *Proceedings of the 7th International Conference on Formal Methods in Computer Aided Design*, Nov. 2007, pp. 165–172. [Online]. Available: https://doi.org/10.1109/FAMCAD.2007.12.

[51] B. Churchill, O. Padon, R. Sharma, and A. Aiken, "Semantic program alignment for equivalence checking," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Phoenix, AZ, Jun. 2019, pp. 1027–1040. [Online]. Available: https://doi.org/10.1145/3314221.3314596.

[52] M. Gabel and Z. Su, "Symbolic mining of temporal specifications," in *Proceedings of the 30th International Conference on Software Engineering*, Leipzig, Germany, May 2008, pp. 51–60. [Online]. Available: https://doi.org/10.1145/1368088.1368096.

[53] W. Li, A. Forin, and S. A. Seshia, "Scalable specification mining for verification and diagnosis," in *Proceedings of the 47th Design Automation Conference*, Anaheim, CA, Jun. 2010, pp. 755–60. [Online]. Available: https://doi.org/10.1145/1837274.1837466.