Automatic Inductive Invariant Generation for Scalable Dataflow Circuit Verification

Jiahui Xu and Lana Josipović
Department of Information Technology and Electrical Engineering, ETH Zurich, Switzerland

***Abstract*—Formal verification via BDD-based reachability analysis has been shown to improve the quality of dataflow circuits produced via high-level synthesis (HLS): it can restrict the generality of the dataflow handshake logic only to provably required constructs and significantly improve their resource requirements. Unfortunately, BDD-based strategies are unscalable for larger circuits. A promising alternative is k-induction, which offers scalability in the presence of suitable inductive invariants. Yet, appropriate invariants are not straightforward to determine: they must provide exclusively relevant information that constrains the induction to a small number of steps without complexing the system under verification. In this paper, we propose a fully automated framework that systematically generates suitable inductive invariants for scalable dataflow circuit verification. Our framework systematically exploits a variety of HLS insights to convey relevant invariant information to the verifier and applies to any dataflow circuit generated from C code. On a set of representative benchmarks, we show that our method significantly outperforms prior BDD-based approaches (i.e., it takes minutes to prove properties that a BDD-based checker cannot prove in days) with only a minor reduction in verification capabilities.**

***Index Terms*—high-level synthesis, dataflow circuit, formal verification, model checking**

I. Introduction

HLS-produced dataflow circuits have performance merits over traditional, statically scheduled circuits when accelerating workloads with unpredictable control flow or memory accesses [1]–[3]. Yet, this gain is not for free: the bidirectional handshake communication signals that enable dataflow circuits to excel also cause a significant and often unacceptable resource cost. Thus, there is a clear need to remove or simplify these signals whenever their flexibility is not needed [4].

A general way to go about such simplifications is to rely on formal methods: techniques that prove that, in no possible situation, a particular handshake signal is required, so that it can be safely omitted without compromising functional correctness. A promising way to achieve this goal is *k-induction*, but it is scalable only in the presence of suitable *inductive invariants* that exclude spurious counter-examples and enable the induction to be solved in a reasonably small number of *k* steps [5]–[7]. Although we might be able to devise these invariants on a case-by-case basis, this does not suffice in the HLS context—we need a systematic way to exploit such invariants in *all* circuits obtained from high-level code.

In this work, we present a methodology for automatically generating inductive invariants targeting HLS-produced dataflow circuits. Our method relies on general observations about the structural patterns and behaviors of dataflow circuits and exploits features of the code they originate from to systematically produce a set of inductive invariants for any dataflow circuit. On a set of dataflow circuits obtained from C code, we demonstrate that our invariants successfully reduce the induction runtime while successfully proving relevant circuit properties; its capabilities significantly exceed those of other formal strategies (i.e., BDD-based model checking [4]).

II. Background

In this section, we describe the dataflow circuits that we aim to optimize via formal methods. We illustrate the benefits of performing such optimizations and contrast several formal methods that could be employed for this purpose.

A. Dataflow Circuits

Dataflow circuits are built from *units* that communicate with their predecessors and successors via latency-insensitive *channels*, composed of data and handshake signals [1], [8]. Once the relevant criteria have been met (e.g., control and memory dependencies have been resolved), units exchange data in the form of *tokens*.

The generation process of dataflow circuits has been the subject of many research works [1], [9]–[12]; without the loss of generality, we here focus on a recent approach for generating dataflow circuits from C code [1]. The circuits we consider organize units into *basic blocks (BBs)*, i.e., straight pieces of code without control flow decisions inside. We consider dataflow circuits that implement sequential programs, i.e., there is a *single* token entering through the entry point, traversing through the intermediate BBs, and exiting through the exit point in a final BB. The circuit is composed of the following units: (1) A *merge* propagates a token non-deterministically to its single output from one of its two data inputs. (2) A *mux* has identical functionality as a merge, except it propagates the input based on an additional condition input. (3) A *branch* propagates the received data token to one of its successors, depending on the value of the received condition token. (4) An *eager fork* distributes a copy of the incoming token to each of the successors as soon as they are ready to receive it. (5) A *join* synchronizes multiple tokens before sending a token to its successor. (6) A *control merge (or cmerge)* is a merge, which has an output that indicates which of the inputs has been taken from the merge. (7) A *buffer* is used to store data tokens, break combinational paths, and increase throughput. (8) An *entry* is an entry point of a dataflow circuit; we consider it as a buffer with one initial token that triggers the circuit's computation. (9) An *exit* is

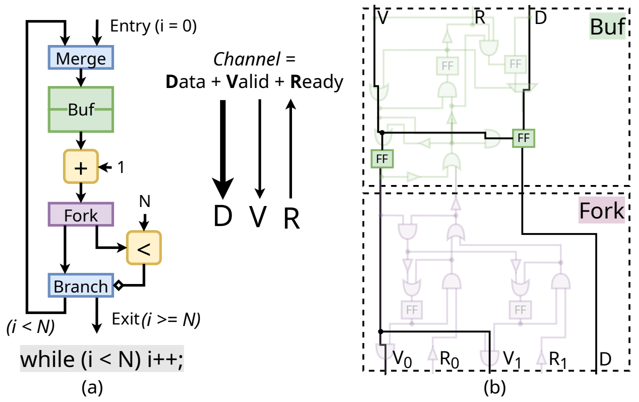


Fig. 1. A simple dataflow circuit that does not need any ready signal in its channels, and many valid signals are equivalent. (a) the schematic that implements the functionality of the simple program below, (b) original and optimized implementations of the *Buf* and *Fork* units, the original implementation is shaded, and the optimized implementation is in black.

an exit point of a dataflow circuit, we consider it as a buffer that stores the circuit's outputs. In general, a buffer insertion does not impact the functionality of the circuit and they can be arbitrarily added without penalizing correctness [1], [13], [14]. There are two caveats: (1) To ensure that the circuit is deadlock-free, each cyclic path has to have at least two buffer slots [14] and there must be no more than one token per cyclic path. (2) To ensure determinism, a buffer must be placed in between the entry units (i.e., merges and muxes) of a BB, and the first eager fork (if any exist) of the same BB [3].

Fig. 1 shows a dataflow circuit implementing the functionality of the code in the figure and built out of units introduced above; all channels between units contain data and handshake signals. In each iteration, the token resides inside the *Buf* and is sent back through the back edge from the left output of the *Branch*, through the *Merge*, and to the same *Buf*. The circuit repeats this for *N* consecutive iterations until the *comparator (<)* evaluates to *false*, which terminates the program by removing the token from the circuit through the right output of the *Branch*. *Buf* is a 2-slot buffer after the merge, hence it honors the two properties above.

B. Eliminating Redundant Handshake Logic

Dataflow circuits equip every channel with bidirectional handshake signals; the circuit is, thus, entirely latency-insensitive. This level of generality is not always necessary. Consider again the example in Fig. 1a: by observing the circuit's behavior, one can immediately notice that there is no unit that stalls its predecessors; thus, the ready signals are redundant and the logic that determines them can be replaced with a constant value of 1 (i.e., all units are always ready); similarly, since the *fork* always triggers the execution of its successors equivalently, the logic to compute all valid signals can be unified. Fig. 1b illustrates the simplified unit implementations using the transformations described above; the complexity reduction with respect to the original implementations (shaded in the figure) is immediately evident.

A recent work [4] exploited BDD-based model checking to generate formal proofs that guarantee that optimizations such as the ones above are correct (i.e., they do not alter any reachable behavior): for every dataflow channel of a circuit obtained from high-level code, this work aims to prove the absence of stalls and the equivalence of valid signals, as in Fig. 1. Circuit modeling required for such an approach is straightforward and results in up to 50% area reduction. Note that there is a fundamental difference between system verification tasks and circuit optimizations (such as the ones we are interested in) in how *property checking techniques* are exploited: in a typical system verification task, every desired property is part of the specification of the system, and any failure implies that the underlying system is incorrect; for the purpose of circuit optimization, it is acceptable to have some statements false—it simply indicates that a particular handshake signal cannot be removed without violating correctness. However, this approach is not scalable, as we will discuss next.

C. Reachability vs. Induction

Unfortunately, BDD-based reachability analysis is known for its scalability issues [7]: even for moderately-sized circuits, no property can be concluded in days [4]. Despite prior efforts to simplify the model by abstracting away data, control flow, and memory constructs, the runtime remains unacceptable; in larger circuits, no useful properties can be concluded.

K-induction is the most prominent technique among the verification methods of safety properties without reachability analysis [5], [15]. A property that can be verified using k-induction satisfies the following [6]: (1) for any k steps starting from the set of initial states, no counter-example can be found; (2) assuming the safety property holds for k consecutive steps, any state obtained after any state transition preserves the property. 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. Therefore, an inductive invariant is a key to constraining the induction depth k, thus ensuring scalability [16]. In the rest of this paper, we devise a set of inductive invariants that enables us to efficiently verify properties such as the ones discussed in Section II-B.

III. Generating Invariants from Dataflow Circuits

This section describes our methods for generating inductive invariants from dataflow circuits. They restrict the unreachable state space seen by an induction engine by leveraging different structural properties of individual units, entire dataflow circuits, and the high-level code they originate from.

A. Localized Invariants

This section describes localized invariants that eliminate unreachable states of the state-holding elements among the units (i.e., buffers and eager forks, as mentioned in Section II-A). These invariants always hold true for these units, regardless of the dataflow circuit they are used in. We will first discuss *forks* and then the *buffers*.

Fig. 2a describes a *fork* in terms of its state registers. We introduce a binary state-variable *sent* for each output of an

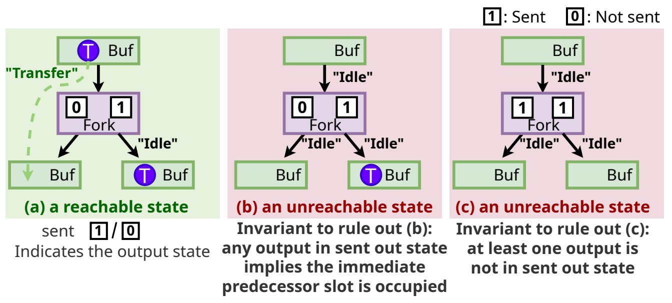
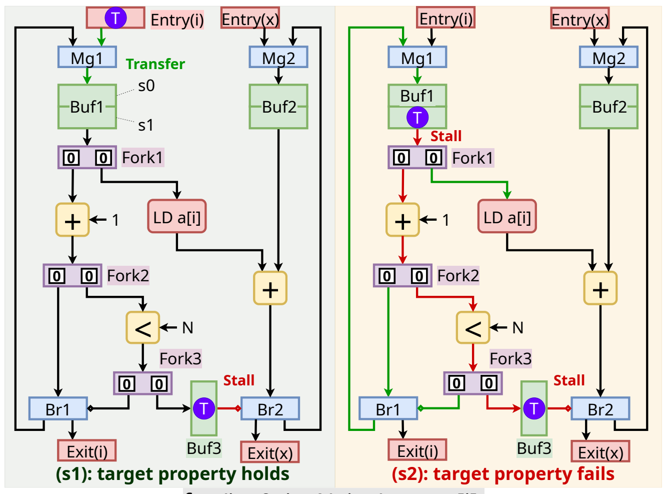


Fig. 2. Using local invariants to rule out unreachable states of an eager fork.



```
for (i = 0; i < N; i++) x += a[i];
```

Fig. 3. A possible counter-example against proving property "Buf_1 never stalled by $Fork_1$" after adding local invariants described in Section III-A.

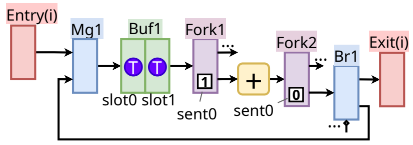


Fig. 4. An in-order graph from the same circuit in Fig. 3.

eager fork; *sent* $= 1$ indicates that the token at the fork input has been issued to its successor through the corresponding channel, and *sent* $= 0$ indicates that there is no token at the fork input or the input token has not been sent yet (which occurs when the successor is not ready to accept the token). For example, the *Fork* in Fig. 2a has already issued a token to the right *Buf*, but not yet to the left *Buf*. Without the addition of invariants, an induction engine would account for all combinations of the *sent* register values in an eager fork, but one can immediately see that some will never occur. For instance, in Fig. 2b, the fork has sent out a token, but there is no token appearing at its input; in Fig. 2c, the fork has issued tokens to both outputs, but the sent registers have not been reset. To rule out such unreachable states, we devise the following invariant *for each eager fork*:

Invariant 1. *For any fork f, the number of outputs that are in* sent *state must be smaller than the total number of fork outputs (N_{out}).*

$$\sum_{j=1}^{N_{out}} f.sent_j < N_{out}. \tag{1}$$

This invariant states that not all outputs of an eager fork can be in a sent-out state simultaneously. Yet, this invariant alone is not sufficient to rule out the state in Fig. 2b: although only one output is in a sent-out state, this situation is impossible, as the token is no longer present in the preceding buffer and can never be issued to the left, thus violating the fork's functionality. To exclude such situations, we require a local relationship between a fork and all the token-holding units from which it can copy tokens. We denote the elementary token-holding unit as a slot s, where the binary state variable $s.full$ denotes whether the slot is full (e.g., in Fig. 1a, *Buf* has two cascading slots). We define the following:

Definition 1. *A path between a pair of dataflow units u and v is a set of non-repeating units and channels starting from u and ending at v. The set of all paths from u to v is denoted as $Paths(u, v)$.*

For instance, Fig. 6 describes a subsection of dataflow circuit, we can identify an ordered set of units $Fork_1$, "+", $Fork_2$, "<", $Fork_3$, Buf_3, Br_2, and all channels in between the units as a path p.

Definition 2. *A slot s belongs to the set of* copied slots *of fork f, denoted as $s \in CopiedSlots(f)$ if the following holds: (1) $Paths(s, f) \neq \{\varnothing\}$ and (2) $\exists p \in Paths(s, f)$, such that the path* does not *contain any slots other than s itself. Similarly, the set of* copying forks *of s is the set of forks such that $\forall f \in CopyingFork(s), s \in CopiedSlots(f)$.*

For example, in the circuit illustrated in Fig. 3, the buffer unit Buf_1 is composed of slot s_0 and slot s_1, the lower slot s_1 is an *copied slot* of $Fork_1$, $Fork_2$, and $Fork_3$; however the upper slot s_0 is not.

Remark 1. *Since we can associate an output state of a fork $f.sent$ with a particular channel, for a path p, we denote $f.sent_k \in p$ if $f \in p$ and output channel k is in the path.*

We propose the following for each fork:

Invariant 2. *For any output state $f.sent$ of a fork f, all slots s such that $s \in CopiedSlots(f)$:*

$$f.sent \implies s.full. \tag{2}$$

This invariant describes that, if some of the outputs of fork f have sent out a token, and others have not, the corresponding token must be held in all copied slots of f. For example, Invariant 2 rules out the possibility of having fork *sent* states in Fig. 2b in any induction counter-examples.

Invariant 3. *For any path p such that p does not contain any slots, the following invariant must hold for all fork sent-out states $f.sent \in p$:*

$$\sum_{f.sent \in p} f.sent \leq 1. \tag{3}$$

This invariant describes that, on every path, a token can be duplicated only once before entering any other slot. For

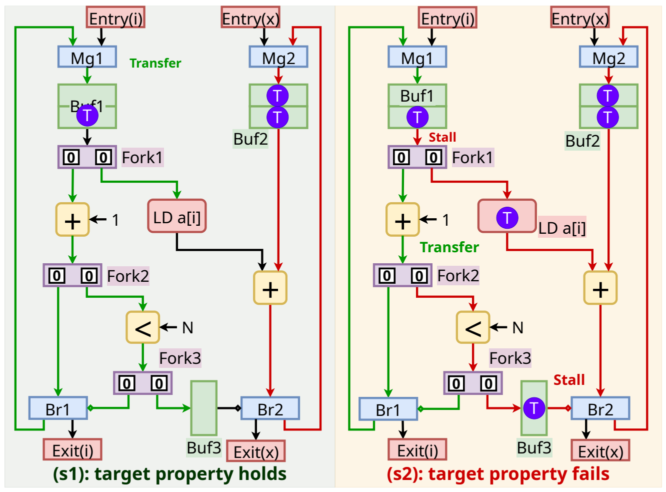


Fig. 5. A possible counter-example for the same property as in Fig 3, after adding the invariant described in Section III-B.

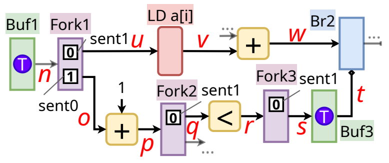


Fig. 6. A pair of recombining paths from the same circuit in Fig. 3. Path 1: Fork1, LD, add, Br2; Path 2: Fork1, add, Fork2, compare, Fork3, Buf3, Br2.

instance, in Fig. 5, only one fork has a *sent-out* state in the path between Buf_1 and Buf_3, and Invariant 3 is honored. We model each buffer as a sequence of slots. Every buffer follows the first-in-first-out principle, and we propose the following invariant *for each buffer unit*:

Invariant 4. *Consider a sequence of slots $s_1, s_2, ..., s_N$ that originate from any buffer unit with N slots.*

$$s_i.full \implies s_{i+1}.full, \forall i \in 1..N-1. \tag{4}$$

This invariant states that, if a given slot inside a buffer is occupied by a token, then its successor slot in the same buffer must also be occupied. For instance, in Fig. 1, Invariant 4 describes that whenever a token occupies the upper slot of Buf, the lower slot must also be occupied.

B. Invariants from Recombining Paths

We characterize the state of a dataflow circuit by the *full* and *sent* states of the slots and forks. Without additional information, an induction engine would account for any combination of these states. Fig. 3 illustrates a dataflow circuit that implements the functionality of the code at the bottom; we aim at removing the ready signal of the channel between Buf_1 and $Fork_1$. The left part of the circuit is analogous to that of Fig. 1; the only difference is that, in every loop iteration, the iterator is also forked into a load to retrieve the value of *a[i]*, to be accumulated to the token that carries the value of x in the rightmost cyclic path. While loading the value of *a[i]* from memory, the control-flow decision is also buffered in Buf_3. The circuit terminates its execution when "<" issues a loop exit control decision, which sinks both i and x to exit.

The target property is honored throughout the execution of

TABLE I
SUMMARY OF FLOW RELATIONS

Unit Type	Relation
Operators (Join)	$\lambda_{in,k} = \lambda_{out}$, for each input k
Eager forks	$\lambda_{in} + sent_k = \lambda_{out,k}$, for each output k When sending token that influences control flow: $\lambda_{in}^{+/-} + sent_k^{+/-} = \lambda_{out,k}^{+/-}$. where $sent_k^{+} = d_{in} \rightarrow sent_k$ and $sent_k^{-} = \overline{d_{in}} \rightarrow sent_k$
Slots	$\lambda_{in} = full + \lambda_{out}$. When holding token that influences control flow: we separately specify $full^{+}$ and $full^{-}$ $\lambda_{in}^{+/-} = full^{+/-} + \lambda_{out}^{+/-}$.
Branches	$\lambda_{in,data} = \lambda_{in,cond} = \lambda_{out,true} + \lambda_{out,false}$ $\lambda_{in,cond}^{+/-} = \lambda_{out,true/false}$
Muxes	$\lambda_{in,sel} = \lambda_{in,true} + \lambda_{in,false} = \lambda_{out,0}$, $\lambda_{in,sel}^{+/-} = \lambda_{in,true/false}$
Merges	$\lambda_{in,0} + \lambda_{in,1} = \lambda_{out}$

the circuit. However, if we attempt to prove it using 1-step induction (even *after* adding the invariants of Section III-A), the solver will generate a counter-example such as the one in Fig. 3 (i.e., the target property holds in s_1 but not in s_2, see Section II-C). Yet, this token assignment is evidently infeasible: whenever $Fork_1$ injects a token to Buf_3 but not yet to LD, a token must be held in Buf_1 until it is eventually injected into LD; meanwhile, exactly one of $Fork_1, Fork_2, Fork_3$ must remember a token has already been sent out, which is not true in this case. We could reason that the state is unreachable because we can identify a pair of *recombining paths* (i.e., paths that originated from the same fork, and recombined at the same join), as illustrated in Fig. 6, such that both paths start from $Fork_1$ and end at Br_2; without knowing anything about the delay of the units, the number of tokens that are issued to one path *must eventually be identical* to the other.

Prior work proposed an automatic method for discovering inductive invariants to systematically rule out unreachable states of this form [7]; For each unit, relations that specify how each unit governs the token transfer λ_{ch} (the number of tokens that have been transferred in a channel *ch* up to a given point in time) in its channels can be derived; the invariants that describe the token count relations of the recombining paths are generated to rule out the unreachable states. We adapt this method to dataflow circuits: a set of relations is derived according to Table I on a per-unit basis; a set of invariants is derived from applying the Gaussian elimination and added to the verification model [7], [17].

As an example, we would like to derive an invariant for the recombining path structure in Fig 6, which is part of the same circuit in Fig 3. All channels in this figure are labeled with a red italic letter. For the eager forks $Fork_1, Fork_2, Fork_3$:

$$\begin{aligned} \lambda_n + Fork_1.sent_0 = \lambda_o,\ \ \lambda_n + Fork_1.sent_1 = \lambda_u, \\ \lambda_p + Fork_2.sent_1 = \lambda_q,\ \ \lambda_r + Fork_3.sent_1 = \lambda_s. \end{aligned} \tag{5}$$

The above relations describe that an eager fork allows an output transfer (i.e., a token issued to a fork output) to happen before the input transfer (i.e., the token sent to all outputs and consumed from the fork's predecessor), up to *sent* many times (which can only be 0 or 1); in other words, a fork output can replicate at most one token (see Section II-A). For token-holding elements *LD* and *Buf3*:

$$\lambda_u = LD.full + \lambda_v,\ \lambda_s = Buf_3.full + \lambda_t. \tag{6}$$

The above relations describe that a slot allows an output transfer to be delayed from an input transfer by buffering the token. For combinational units *Br2*, "+" and "<":

$$\lambda_w = \lambda_t,\ \lambda_o = \lambda_p,\ \lambda_v = \lambda_w,\ \lambda_q = \lambda_r. \tag{7}$$

The above relations describe that the input transfer of these units must be synchronized. After eliminating the λ variables from the system of equations derived from individual units (i.e. the resulting relations hold for any point in time), we obtain the following invariant:

$$\begin{aligned}&\mathrm{LD}.full + \mathrm{Fork}_3.sent_1 + \mathrm{Fork}_2.sent_1 +\\ &\mathrm{Fork}_1.sent_0 = \mathrm{Buf}_3.full + \mathrm{Fork}_1.sent_1.\end{aligned} \tag{8}$$

This invariant requires the token occupation in the two recombining paths in Fig. 6 to be balanced, which eliminates the state s1 in Fig. 3; if one path has more tokens than the other, the eager fork sents must compensate for the difference.

C. Invariants from Bounded Structures

The invariants of Section III-B do not capture the fact that our circuits contain a bounded or even a fixed number of tokens. In this section, we devise invariants that describe these features. Consider that we add the structural invariants from Section III-B to rule out the counter-example in Fig. 3: the solver will generate a different counter-example, as described in Fig. 5: the two tokens in Buf_2 cause a stall on the path with the LD, thus breaking the target property.

Throughout the execution of the program, since there is exactly one token representing the value of *x* (recall that the circuit implements a sequential program, see Section II-A) and there are no eager forks that can duplicate the value of *x* along the paths that the token *x* can traverse, it is impossible to have two tokens reside in the two slots in Buf_2 simultaneously. To systematically rule out an incorrect number of tokens, we need to identify circuit structures in which the token count remains constant. Fig. 4 details a subsection of the circuit in Fig. 5 that has a fixed number of tokens: the slots in this part of the circuit hold the value of the variable *i*; the number of tokens in this structure is always equal to its *number of initial tokens* and *the number of copies produced by the forks* since there is no way to inject new tokens or remove tokens from this part of the circuit. Such structures are generally found in the dataflow circuits compiled from sequential programs [18]; we define them as *in-order graphs*.

Definition 3. *An in-order graph (IOG) of a dataflow circuit is a subset of units and channels, such that: (1) there exists one and only one entry unit and the entry unit can reach all other units on the IOG, (2) for each fork on the IOG, only one of its output channels can belong to this graph, (3) for each merge and mux on the IOG, all their input data channels must belong to the graph.*

For example, the dataflow circuit in Fig. 5 contains two IOGs: (1) the one that is described in Fig. 4 and (2) the IOG that consists of *Entry(x)*, *Mg2*, *Buf2*, *the adder "+"*, *Br2*, and *Exit(x)*. For every IOG, we propose the following:

Invariant 5. *For any in-order graph IOG, the following relation is invariant over the number of slots that are occupied (s.full) and the output states of forks (f.sent):*

$$\sum_{s \in IOG} s.full = T_{init} + \sum_{f \in IOG} f.sent. \tag{9}$$

For any IOG, $T_{init} = 1$. For example, in the IOG that accommodates value of x in Fig. 5, we could formulate Invariant 5 as

$$\begin{aligned}&Entry_x.full + Buf_2.slot_0.full +\\ &Buf_2.slot_1.full + Exit_x.full = 1,\end{aligned} \tag{10}$$

from which we can have the same conclusion as before, i.e., it is impossible to occupy both of the slots in Buf_2 with tokens, which rules out the state s0 in Fig. 5. We developed a procedure for in-order graph enumeration based on a breadth-first search; we generate Invariant 5 for all of them.

To exclude unreachable circuit states, in addition to the total number of tokens on an IOG, it is important to reason about the distribution of tokens in particular slots. For every IOG, we formulate the following invariant for every pair of slots:

Invariant 6. *For any two slots $s_i, s_j \in IOG$, the following relation must hold:*

$$\begin{aligned}&(s_i.full) \wedge (s_j.full) \implies \exists f, p : (f \in p) \wedge (f.sent) \wedge\\ &(p \subset IOG) \wedge (p \in Paths(s_i, s_j) \wedge s_i \in CopiedSlots(f) \vee\\ &p \in Paths(s_j, s_i) \wedge s_j \in CopiedSlots(f)),\end{aligned} \tag{11}$$

where f is a fork in p.

Invariant 6 describes that, if two slots on an IOG are both full, there must be a path connecting them; the slot at the beginning of this path must be duplicated by a *copying fork* (i.e., $s \in CopiedSlots(f)$, see Section III-A). For example, in the IOG in Fig. 4, Invariant 6 rules out the possibility of having $slot_1$ and $Entry$ both occupied by tokens, as there is no copying fork to duplicate tokens from $Entry$ (notice that this state is not ruled out by Invariant 5).

D. Invariants for Token Ordering

Our invariants so far described rules for token counts and distributions. However, these rules do not account for the incorrect ordering of tokens, whose values directly influence the control flow in unreachable states. We therefore devise the corresponding invariants in this section.

Fig. 7 illustrates a dataflow circuit that accumulates the squared values of the iterator in every loop iteration. The circuit begins its execution by injecting the initial values of the iterator and accumulator into the loop; a dataless control token is also injected through a *cmerge* unit CMg_1. In every iteration, CMg_1 generates a decision token, forked through $Fork_2$, and indicates from which input Mux_1 and Mux_2 should take tokens. The iterator token *i* is stored in Buf_2, and is forked into $Pipeline_1$ (a unit of latency 4) to calculate the squares; at the same time, *i* triggers the execution of the decider – an abstract component that produces non-

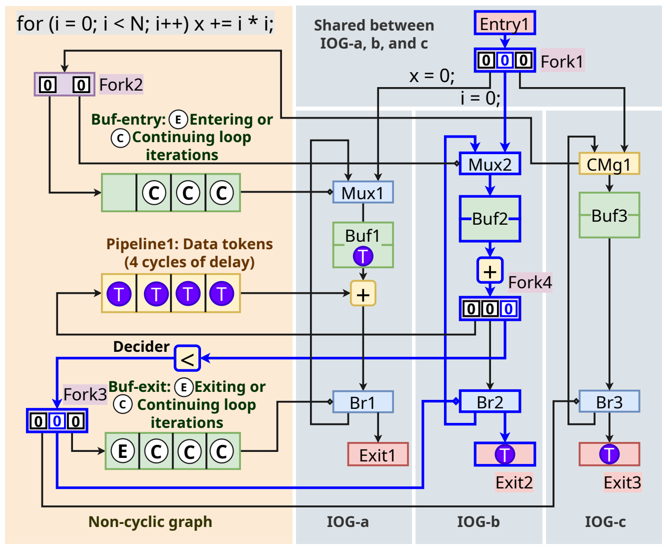

for (i = 0; i < N; i++) x += i * i;
Shared between IOG-a, b, and c
Entry1
Fork1
Fork2
x = 0;
i = 0;
Buf-entry: E Entering or C Continuing loop iterations
Mux1
Mux2
CMg1
Buf2
Buf3
Buf1
Pipeline1: Data tokens (4 cycles of delay)
Fork4
Decider
Fork3
Buf-exit: E Exiting or C Continuing loop iterations
Br1
Br2
Br3
Exit1
Exit2
Exit3
Non-cyclic graph
IOG-a
IOG-b
IOG-c

Fig. 7. Interaction between different in-order graphs. The non-cyclic graph is composed of the units that pass tokens in between different in-order graphs.

deterministic control decisions instead of actual comparison [4]. The decider dictates whether the cyclic paths (the three IOGs a, b, and c) should repeat the loop iteration or dispatch the tokens into the exit units. Condition C indicates to continue the loop iteration; E indicates entering the loop for Mux and exiting for Br. The accumulator token x is stored in Buf_1 and initially has to wait for the first token from $Pipeline_1$ for 4 cycles before it can start accumulating values; while waiting, the control decisions are buffered in Buf_{entry} and Buf_{exit}, which keep track of the control decisions that token x has to follow. After the first square arrives, in every iteration, a new square is accumulated into x through the adder, and the tokens in Buf_{exit} (from 4 cycles before) decide if token x should continue the loop iteration or exit.

Without suitable invariants, an induction engine will account for sequences of control-flow decisions that are impossible for sequential programs. Consider the circuit in Fig. 7 but, instead of having ”C, C, C” in Buf_{entry}, assume a sequence ”C, E, C”; all other tokens remain at the same location and values. In this case, after the token repeats a loop iteration through Mux_1, Buf_{entry} expects a token to enter the loop through Mux_1, but the token has already entered (see Buf_1); as no further token will enter, the Mux_1 select token will never change to C, and the token in Buf_1 will never advance; the verifier concludes that the circuit deadlocks and misses the opportunity to verify any property (e.g., the output channel of $Pipeline_1$ never stalls). Yet, this scenario is clearly nonsensical: Buf_{entry} can receive only a single E token during the circuit execution and this token must have already been consumed by the mux (otherwise, there would be no token in Buf_1).

We propose invariants to eliminate unreachable states associated with this value-ordering problem: we restrict the value sequence of the slots in the path between CMg and Mux_1 only to those that are actually possible in a sequential program execution. To this end, we introduce a binary variable $s.value$, which represents the token value stored in slot s.

Definition 4. *Given a path p, the value function $Values(p)$ returns an ordered set $\{s_{first}.value, ..., s_{last}.value\}$, representing an ordered sequence of* unique *tokens (i.e. the tokens duplicated by eager forks are removed) on p.*

Remark 2. *The first element in a sequence (s_{first}) is the value of the last token that was injected into the path.*

Invariant 7. *$\forall CMg_{entry}, Mux_{entry} \in BB_{entry}$ in the header BB of the outermost loop, all paths $p_{cm} \in Paths(CMg_{entry}, Mux_{entry})$, the sequence can be represented using regular language as follows:*

$$Values(p_{cm}) := \{C * E?\}, \tag{12}$$

where ”$*$” indicates that the preceding symbol repeats arbitrarily many times and ”?” indicates that the preceding symbol occurs 0 or 1 time.

This invariant describes that only the last valid value in the path (i.e., first to be consumed by the Mux) between CMg and Mux can be a loop entry (E).

Additionally, once a token enters the outermost loop, no further token can be injected into the same CMg from the entry point into the loop header BB:

Invariant 8. *For all entry units of the dataflow circuit Entry, and all paths $p_{ec} \in Paths(Entry, CMg_{entry})$:*

$$(Values(p_{cm}) \neq \{\varnothing\}) \implies (Values(p_{ec}) = \{\varnothing\}). \tag{13}$$

We must also restrict the order of the token values that control the behavior of the *Branch*. In Fig. 7, instead of having ”E, C, C, C” in Buf_{exit}, assume a sequence ”C, C, C, E”. In this case, the sequence in Buf_{exit} expects Br_1 to first let the program exit from the loop, and then repeats the loop iteration three more times. However, after the token exit from IOG_a, no token in Buf_{entry}, Buf_{exit}, and $Pipeline_1$ can be consumed, pointing the verifier to deadlock and precluding circuit simplifications. To rule this out, we propose the following invariants to restrict the value sequence of the slots in the path between Decider and Br only to those that are actually possible in sequential program execution:

Invariant 9. *In the BB_{exit} that is the exit BB of the outermost loop, the decider Dec_{exit}, all branch nodes Br_{exit}, and the paths $p_{db} \in Paths(Dec_{exit}, Br_{exit})$. The sequence in the path can be represented in regular language:*

$$Values(p_{db}) := \{E?C*\}. \tag{14}$$

This invariant describes that the outermost loop exit should always be the last decision to be produced by a decider. Consider the IOG_{Dec} that contains Dec_{exit}, we construct new subgraphs IOG'_{Dec} that are the union of all units and channels of IOG_{Dec} and p_{db}.

The following describes, if any slot holds the outermost loop exit decision, then no other slot can send tokens to it.

Invariant 10. *For all slots that hold outermost loop exit condition $s_{db} \in p_{db}$, all slots that are s_{db}'s an-*

TABLE II
MODEL STATISTICS & INVARIANT VERIFICATION RUNTIME

Benchmark	#Slots	#Sents	#Units	#Ch.	No Recomb. Paths (s)	All Invar (s)
fir	46	19	86	98	0.05	1.04
matrix power	67	34	150	170	0.07	931.98
matvec	65	41	147	172	0.12	79.89
bicg	83	48	177	204	0.11	5960.88
2mm	248	142	557	665	3.29	TO(72h)
3mm	270	180	660	782	3.51	TO(72h)

cestors $s_{ancestor} \in IOG'_{Dec}$ *and all paths* $p_{ab} \in Paths(s_{ancestor}, s_{db})$ *such that* $p_{ab} \subset IOG'_{Dec}$*, the following must hold:*

$$(s_{db}.value = E) \implies (Values(p_{ab}) = \{\varnothing\}). \quad (15)$$

For example, in Fig. 7, consider IOG_{Dec} shown in blue and containing a decider Dec; buffer Buf_{exit} receives tokens from IOG_{Dec}. We see a condition E inside Buf_{exit}: since no ancestor of Buf_{exit} holds a token that can be sent to Buf_{exit}, Invariant 10 is honored.

To account for token duplication across the decider (i.e., a token is held by a decider's predecessor, and not yet issued to all its successors), we include the following constraint:

Invariant 11. *Consider every slot* s*, path* p_{sd} *such that* $p \in Paths(s, Dec), p \subset IOG_{Dec}$*, and* p_{sd} *contains no other slots than* s*, for all* $s_{db} \in p_{db}$ *the following must hold:*

$$(s_{db}.value = E \wedge p_{sd} \neq \{\varnothing\}) \implies (p_{sd} = \{E\}). \quad (16)$$

With these invariants, we exclude token sequences that are not possible in any sequential program.

IV. EVALUATION

In this section, we evaluate the effectiveness of our inductive invariant-based strategy in verifying and optimizing dataflow circuits obtained from high-level code.

A. Methodology and Benchmarks

We incorporate our strategy in a verification framework [4] targeting dataflow circuits produced by Dynamatic, an open-source HLS compiler that translates C/C++ into dataflow circuits [19]. This framework automatically generates an abstract circuit model based on SMV [20] and a set of properties to prove (e.g., valid signal equivalence and the absence of stalls for every channel, as discussed in Section II-B). The original framework verifies these properties using BDD-based model checking in nuXmv [21]. Whenever a channel property evaluates to *true*, the channel logic is simplified accordingly; if the property is *false*, the channel remains intact.

We use the same flow for our strategy; the only difference is that, instead of BDD-based model checking, we employ k-induction enhanced with the inductive invariants from Section III. Note that BDD model checking classifies *all* properties as *true* or *false*; our induction-based strategy may result in an *undecided* property if the induction depth is too small to verify it. In such cases, we do not optimize the corresponding channel logic. Hence, our goal is to have a short runtime, a minimal number of *undecided*, and a maximal number of *true* properties that allow circuit simplifications.

TABLE III
PROPERTY VERIFICATION RUNTIME & SYNTHESIS RESULTS

Bench.	Techn.	Inv	k	Check Time(s)	#T/F/U	LUT (%Red)	FF (%Red)
fir	**No Opt.**	-	-	-	0/0/68	513	508
	BDD	-	-	7.8	53/15/0	276(-46%)	267(-47%)
	Ind	∅	1	0.3	17/12/39	503(-2%)	500(-2%)
	Ind	all	1	1	35/12/21	383(-25%)	379(-25%)
	Ind	∅	5	0.9	20/15/33	495(-4%)	495(-3%)
	Ind	all	5	1.5	52/15/1	288(-44%)	267(-47%)
	Ind	∅	10	3.4	20/15/33	495(-4%)	495(-3%)
	Ind	all	10	1.6	53/15/0	276(-46%)	267(-47%)
matrix power	**No Opt.**	-	-	-	0/0/128	830	415
	BDD	-	-	243.7	45/83/0	493(-41%)	364(-12%)
	Ind	∅	1	0.8	15/33/80	585(-30%)	409(-1%)
	Ind	all	1	4.1	36/33/59	752(-9%)	380(-8%)
	Ind	∅	5	2.3	17/77/34	567(-32%)	405(-2%)
	Ind	all	5	7.3	44/77/7	735(-11%)	370(-11%)
	Ind	∅	10	6.3	17/82/29	567(-32%)	405(-2%)
	Ind	all	10	10.8	44/82/2	735(-11%)	370(-11%)
matvec	**No Opt.**	-	-	-	0/0/128	781	587
	BDD	-	-	55.1	96/32/0	373(-52%)	280(-52%)
	Ind	∅	1	0.8	20/19/89	768(-2%)	581(-1%)
	Ind	all	1	5.5	46/19/63	634(-19%)	494(-16%)
	Ind	∅	5	3.7	22/31/75	753(-4%)	578(-2%)
	Ind	all	5	23.8	55/31/42	552(-29%)	425(-28%)
	Ind	∅	10	16.8	22/32/74	753(-4%)	578(-2%)
	Ind	all	10	51.6	77/32/19	463(-41%)	360(-39%)
bicg	**No Opt.**	-	-	-	0/0/152	904	844
	BDD	-	-	2202.2	58/94/0	711(-21%)	678(-20%)
	Ind	∅	1	1.4	20/29/103	889(-2%)	838(-1%)
	Ind	all	1	9.1	47/29/76	798(-12%)	751(-11%)
	Ind	∅	5	4.9	22/68/62	889(-2%)	835(-1%)
	Ind	all	5	26.4	50/68/34	749(-17%)	716(-15%)
	Ind	∅	10	15.6	22/85/45	889(-2%)	835(-1%)
	Ind	all	10	53.8	50/85/17	749(-17%)	716(-15%)
2mm	**No Opt.**	-	-	-	0/0/489	3061	2794
	BDD	-	-	TO(72h)	0/0/489	3061(-0%)	2794(-0%)
	Ind	∅	1	14.7	52/61/376	2997(-2%)	2720(-3%)
	Ind	all	1	2750.3	160/61/268	2470(-19%)	2259(-19%)
	Ind	∅	5	92.2	57/106/326	2924(-4%)	2618(-6%)
	Ind	all	5	7592.6	185/106/198	2301(-25%)	2117(-24%)
	Ind	∅	10	472.3	57/161/271	2924(-4%)	2618(-6%)
	Ind	all	10	20677	186/161/142	2285(-25%)	2084(-25%)
3mm	**No Opt.**	-	-	-	0/0/596	2706	1907
	BDD	-	-	TO(72h)	0/0/596	2706(-0%)	1907(-0%)
	Ind	∅	1	21.8	49/91/456	2677(-1%)	1901(-0%)
	Ind	all	1	742.6	230/91/275	2356(-13%)	1720(-10%)
	Ind	∅	5	150.5	51/122/423	2694(-0%)	1898(-0%)
	Ind	all	5	2867.9	263/122/211	2312(-15%)	1679(-12%)
	Ind	∅	10	745.5	51/165/380	2694(-0%)	1898(-0%)
	Ind	all	10	11523.8	263/165/168	2312(-15%)	1679(-12%)

We verify our set of invariants using 1-step induction *prior to verifying our circuits*; this allows us to use our invariants as system assumptions when proving the properties of interest. This approach differs from prior works on induction [7] that verify the invariants *in conjunction* with the properties of interest. The reason is the fundamental difference of the verification goals, discussed in Section II-C: unlike in standard verification systems, a *false* property is perfectly acceptable here; it just indicates the inability to simplify a particular logic construct. Thus, it is sensible to first prove that the entire set of inductive invariants is correct and to reason about the system properties separately—combining the two would require repeating the verification process while excluding *false* properties until the invariant correctness has been proven, thus unnecessarily prolonging the verification process for the same final outcome. The verification of most invariants is extremely fast, as reported in Table II, column *No Recomb. Paths*, showing the time needed to verify all invariants except those of Section III-B (i.e., recombining paths). Including these invariants increases the verification time drastically (column *All Invar*). Although we here prove these invariants for completeness, these proofs could easily be omitted as they originate directly from the well-known composability of latency-insensitive units that makes dataflow circuits correct by construction [8], [22].

We report the verification results and runtime of nuXmv on a consumer laptop. Each verification run is timed out after 72 hours (in case it has not terminated prior to that time limit). We report the post-place-and-route area results (i.e., the number of FPGA LUTs and FFs) using Vivado [23].

Since all the optimization strategies we consider focus on the datapath of dataflow circuits, we consider exclusively the datapath resources and we omit those of the memory interface. To ensure that our circuit modifications have not compromised circuit behavior, we additionally verify them in ModelSim [24] through functional simulation to confirm that our optimizations did not modify the circuit's functionality in any way.

Our benchmarks are typical HLS kernels that originate from a standard HLS suite [25] and have been previously used for evaluating the effectiveness of dataflow circuits in HLS [3], [4], [12], [26]. Table II details the characteristics of our benchmarks: *#Slots* and *#Sents* are the total number of slots and sent-out states in eager forks; *#Units* and *#Ch.* are the total number of units and channels in the data path. As we will see next, these characteristics will significantly impact the scalability and effectiveness of our optimization.

B. Results: BDD vs. Induction

We here compare our induction-based strategy with a BDD-based approach in terms of verification runtime, the number of verified properties, and area optimization effectiveness.

Table III compares different verification techniques, as indicated in column *Techn.*: *No Opt* corresponds to dataflow circuits that are not optimized via formal verification, *BDD* is BDD-based model checking [4], and *Ind* refers to k-induction-based model checking. For experiments with *Ind*, in column *k*, we report the maximum allowed induction depth; in column *Inv*, we indicate whether we included the invariants from Section III as *all* or $\varnothing$. We report the property checking results in column *#T/F/U*: (T) is the number of properties that are honored by the circuit; only they can be used for optimization, (F) is the number of properties that are falsified by counter-examples, and (U) is the number of properties that cannot be concluded within the induction depth. The final two columns show the LUT and FF usage and improvement.

For the designs that *BDD* is able to handle, the verifier proves all properties (i.e., *undecided* count is always zero) and the circuits can be aggressively optimized: the LUT and FF count is significantly (i.e., up to 52%) smaller than in the non-optimized circuits. However, the checking time quickly increases with benchmark complexity; for the two most complex benchmarks (*2mm* and *3mm*), the verifier times out and the circuits cannot be optimized at all. In contrast, all induction-based verifications successfully terminate: the benchmarks that were verifiable with *BDD* can now be verified faster, and those where *BDD* was unsuccessful now terminate within a reasonable runtime. As expected, the number of *undecided* properties varies with different induction characteristics; we will further explore these variations in the following sections. Consequently, *BDD* is sometimes able to optimize the circuits more aggressively than its induction-based counterparts (e.g., in *matrix power*, the best induction-optimized design achieves a LUT reduction of 32%, whereas the *BDD* reduction is 41%). Still, the minor reduction in optimization capabilities in smaller benchmarks is acceptable for the significant runtime savings (in the *matrix power*, the small reduction in LUT savings comes at a 100$\times$ verification time speedup); most importantly, in larger benchmarks, where area reduction typically matters the most, induction achieves optimized designs (i.e., with up to 25% LUT and FF savings) which were impossible with *BDD*. This points to the importance of employing induction for scalable circuit verification.

V. Related Work

Automatically generating inductive invariants has been the subject of many research works [7], [15], [27], [28]; to our best knowledge, the most recent work that targets hardware verification is Chatterjee et al. [7]. This work focuses on deadlock freedom verification of communication fabrics and has presented a systematic strategy for generating suitable inductive invariants, rule out unreachable states, and verify properties using 1-step induction. Despite the similarity between the units that are used to model communication fabrics and the dataflow units, the purposes that these platforms serve are very different; dataflow units are more complex and diverse. For instance, dataflow circuits contain eager forks instead of lazy forks to increase the throughput by allowing operations to execute more out-of-order. As a result, our desired properties require additional rules to prove.

Sequential synthesis aims at reducing circuit cost without alternating functionality, which shares a similar goal with our approach [15], [29]. Yet, these approaches are mostly based on either pure k-induction or mining inductive invariants from gate-level descriptions.

VI. Conclusion

In this work, in order to support k-induction-based model checking on dataflow circuits derived from high-level code, we have developed a strategy for generating inductive invariants. Our invariants rule out unreachable states that violate the properties of individual units, of the entire data flow circuits, and of the high-level code from which they are derived. By enabling a more scalable verification strategy, we reduce the resource requirements of larger dataflow designs that previous techniques could not handle. This makes this HLS paradigm more affordable and widely applicable.

VII. Acknowledgements

We thank Satrajit Chatterjee and Alan Mishchenko for their insightful feedback and help. This work has been supported by the Swiss National Science Foundation (project number 200021_215747).

REFERENCES

[1] 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.

[2] L. Josipović, A. Guerrieri, and P. Ienne, "Synthesizing general-purpose code into dynamically scheduled circuits," *IEEE Circuits and Systems Magazine*, vol. 21, no. 1, pp. 97–118, May 2021.

[3] L. Josipović, A. Guerrieri, and P. Ienne, "From C/C++ code to high-performance dataflow circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 7, pp. 2142–55, Jul. 2022.

[4] J. Xu, E. Murphy, J. Cortadella, and L. Josipović, "Eliminating excessive dynamism of dataflow circuits using model checking," Monterey, CA, Feb. 2023, pp. 27–37.

[5] M. Sheeran, S. Singh, and G. Stålmarck, "Checking safety properties using induction and a sat-solver," in *Formal Methods in Computer-Aided Design*, Berlin, Heidelberg, 2000, pp. 127–144.

[6] P. Bjesse and K. Claessen, "Sat-based verification without state space traversal," in *Formal Methods in Computer-Aided Design: Third International Conference, FMCAD 2000 Austin, TX, USA, November 1–3, 2000 Proceedings 3*. Springer, 2000, pp. 409–426.

[7] 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. 147–169, 2012.

[8] J. Cortadella, M. Kishinevsky, and B. Grundmann, "Synthesis of synchronous elastic architectures," in *Proceedings of the 43rd Design Automation Conference*, San Francisco, CA, Jul. 2006, pp. 657–62.

[9] M. Budiu, P. V. Artigas, and S. C. Goldstein, "Dataflow: A complement to superscalar," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, Austin, TX, Mar. 2005, pp. 177–86.

[10] S. A. Edwards, R. Townsend, and M. A. Kim, "Compositional dataflow circuits," in *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design*, Vienna, Sep. 2017, pp. 175–84. [Online]. Available: http://doi.acm.org/10.1145/3127041.3127055

[11] L. P. Carloni, K. L. McMillan, and A. L. Sangiovanni-Vincentelli, "Theory of latency-insensitive design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 9, pp. 1059–76, Sep. 2001.

[12] A. Elakhras, A. Guerrieri, L. Josipović, and P. Ienne, "Unleashing parallelism in elastic circuits with faster token delivery," in *Proceedings of the 22nd International Conference on Field-Programmable Logic and Applications*, Belfast, UK, Aug. 2022.

[13] D. Bufistov, J. Cortadella, M. Kishinevsky, and S. Sapatnekar, "A general model for performance optimization of sequential systems," in *Proceedings of the International Conference on Computer-Aided Design*, San Jose, CA, Nov. 2007, pp. 362–69.

[14] L. Josipović, S. Sheikhha, A. Guerrieri, P. Ienne, and J. Cortadella, "Buffer placement and sizing for high-performance dataflow circuits," in *Proceedings of the 28th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Seaside, CA, Feb. 2020, pp. 186–96.

[15] M. L. Case, A. Mishchenko, and R. K. Brayton, "Automated extraction of inductive invariants to aid model checking," in *Formal Methods in Computer Aided Design (FMCAD'07)*, 2007, pp. 165–172.

[16] D. Beyer, M. Dangl, and P. Wendler, "Boosting k-induction with continuously-refined invariants," in *Computer Aided Verification*, D. Kroening and C. S. Păsăreanu, Eds., 2015, pp. 622–640.

[17] F. Krückeberg and M. Jaxy, "Mathematical methods for calculating invariants in petri nets," in *Advances in Petri Nets 1987*, G. Rozenberg, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, pp. 104–131.

[18] L. Josipović, A. Guerrieri, and P. Ienne, "Speculative dataflow circuits," in *Proceedings of the 27th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Seaside, CA, Feb. 2019, pp. 162–71.

[19] 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.

[20] E. Clarke, K. McMillan, S. Campos, and V. Hartonas-Garmhausen, "Symbolic model checking," in *Computer Aided Verification*, Berlin, Heidelberg, Jun. 1996, pp. 419–22.

[21] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, "The nuXmv symbolic model checker," in *Computer Aided Verification*, Vienna, Austria, Jul. 2014, pp. 334–42.

[22] J. Cortadella, M. Galceran-Oms, and M. Kishinevsky, "Elastic systems," in *Proceedings of the 10th ACM/IEEE International Conference on Formal Methods and Models for Codesign*, Jul. 2010, pp. 149–58.

[23] *Vivado Design Suite*, Xilinx Inc., 2020. [Online]. Available: https://docs.xilinx.com/v/u/2019.2-English/ug901-vivado-synthesis

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

[25] L.-N. Pouchet, *Polybench: The polyhedral benchmark suite*, 2012. [Online]. Available: http://www. cs. ucla. edu/pouchet/software/polybench

[26] C. Rizzi, A. Guerrieri, P. Ienne, and L. Josipović, "A comprehensive timing model for accurate frequency tuning in dataflow circuits," in *Proceedings of the 22nd International Conference on Field-Programmable Logic and Applications*, Belfast, UK, Aug. 2022.

[27] 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. 154–158, 2009.

[28] M. Case, A. Mishchenko, and R. Brayton, "Cut-based inductive invariant computation," *at IWLS*, 2008.

[29] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Computer Aided Verification*, Berlin, Heidelberg, 2010, pp. 24–40.