Automatic Inductive Invariant Generation for Scalable Dataflow Circuit Verification

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

2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD) | 979-8-3503-2225-5/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICCAD57390.2023.10323796

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. We also demonstrate the effectiveness of our approach over prior induction-based techniques by proving up to 4× more properties in the same runtime.

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

I. Introduction

High-level synthesis (HLS) tools generate RTL designs from high-level programming languages like C/C++ and promise to liberate designers from low-level hardware details [1], [2]. HLS-produced *dataflow circuits* have performance merits over traditional, statically scheduled circuits when accelerating workloads with unpredictable control flow or memory accesses [3], [4]. 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 [5].

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 [6]–[8]. Although these invariants may be straightforward to devise 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 effectively proving relevant circuit properties; its capabilities significantly exceed those of other formal strategies (i.e., BDD-based model checking [5]).

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 [3], [9]. 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 studies [3], [10]–[13]; without the loss of generality, we here focus on a recent approach for generating dataflow circuits from C code [3]. The circuits we consider organize units into *basic blocks* (BBs), i.e., straight pieces of code without control flow decisions inside. In line with standard HLS tools [1]–[4], we consider dataflow circuits that implement sequential programs with a single thread, 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; it contains an internal register per output to memorize whether data has already been issued to this output, which ensures that the same token is not sent twice to the same output. (5) A *join* synchronizes multiple tokens before sending a token to its successor; it is typically used in arithmetic units to ensure the presence of all inputs prior to computing. (6) A *control merge* (or *cmerge*) is a merge that 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 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 [3], [14], [15]. There are two caveats: (1) To ensure that the circuit is deadlock-free, each cyclic path has to have at least two buffer slots [15] 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 [4].

Fig. 1 shows a dataflow circuit implementing the functionality of

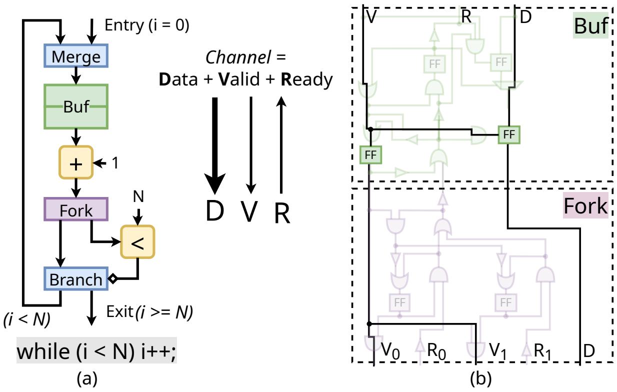


Fig. 1. Removing redundant dataflow logic. The dataflow circuit in Fig. 1a implements the functionality of the program below. Since this circuit does not need any ready signals and has many equivalent valid signals, its implementation can be simplified, as shown in Fig. 1b; the original implementation is shaded and the optimized implementation is in black.

the code in the figure and built out of units introduced above; all channels between units contain data and handshake signals (i.e., valid and ready). 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 and are, thus, entirely latency-insensitive. This level of generality is not always necessary. Consider again Fig. 1a: by observing the circuit's behavior, one can immediately notice that no unit 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 shows the simplified implementation of the *Buf* and *Fork* using the transformations described above; the complexity reduction with respect to the original, general implementations (shaded in the figure) is immediately evident.

A recent work [5] 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 the example of Fig. 1. Circuit modeling required for such an approach is straightforward and the results are very promising (i.e., up to 50% reduction in area).

Note that there is a fundamental difference between system verification tasks and circuit optimizations (e.g. 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 system specification, and any failure implies that the underlying system is incorrect; for optimization purposes, it is acceptable to have some statements false—it simply indicates that a particular handshake signal could not be removed without penalizing correctness.

C. Reachability vs. Induction

Unfortunately, BDD-based reachability analysis is known for its scalability issues [8]: even for moderately size dataflow circuits, it cannot conclude a single property in days [5]. 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 one of the most widely used hardware verification techniques that does not rely on reachability analysis [6], [16], [17]. K-induction proves a property that satisfies the following [7]: (1) for any k steps starting from the set of initial states, no counter-example can be found; (2) assuming the 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 proving non-trivial properties. *Inductive invariants* are properties that rule out unreachable states and enable the induction proof of the target property to be solved with a small k; they can be proven using 1-step induction and, thus, scalably verified [18]. Therefore, they are a key mechanism for making k-induction efficient.

In the rest of this paper, we devise a set of inductive invariants by leveraging the insights from the characteristics of the dataflow units and circuits (Section III-A and III-B) and the high-level programs they implement (Section III-C and III-D); this enables us to efficiently verify properties such as the ones discussed in Section II-B.

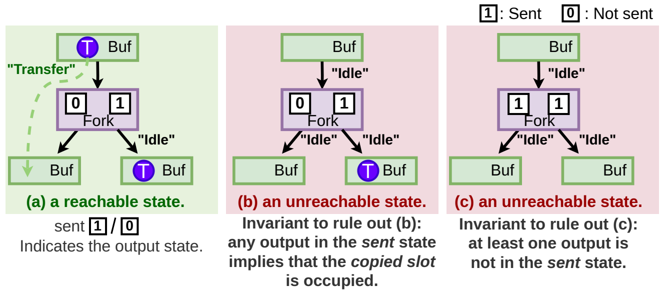


Fig. 2. Using local invariants to rule out unreachable states of an eager fork.

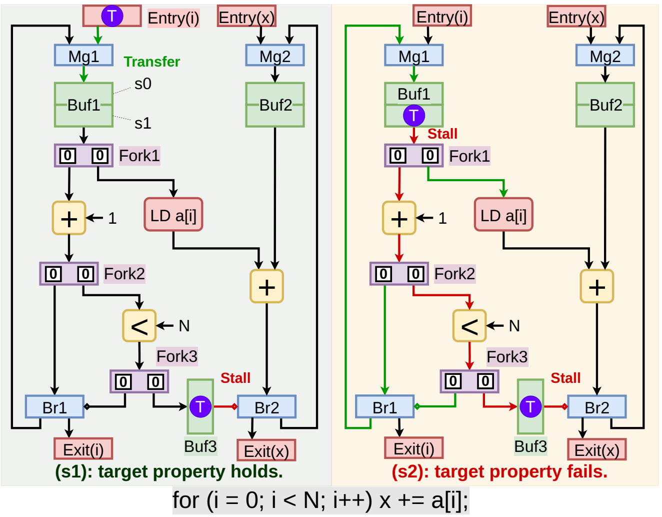
III. Generating Invariants for Dataflow Circuits

This section describes our methods for generating inductive invariants for dataflow circuits. These invariants establish local and global relations between the state-holding units in the circuit. They restrict the unreachable state space seen by a model checker 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 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



(s1): target property holds.

(s2): target property fails.

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

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

not been reset. To rule out such unreachable states, we devise the following invariant *for each fork*:

Invariant 1. *For any fork f, the following invariant must hold:*

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

where N_{out} is the number of outputs of fork f.

This invariant states that not all outputs of an eager fork can be in a *sent* state simultaneously. Yet, this invariant alone is not sufficient to rule out the state in Fig. 2b: although only one output is in a *send* 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. To this end, 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* can be seen as 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, in the circuit in Fig. 3, 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.*

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 a *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 invariant for *each fork*:

Invariant 2. *For any output state $f.sent$ of a fork f and 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. It rules out the situation of Fig. 2b because the upper *Buf* is not full.

Invariant 3. *For any path p such that p does not contain any slots, the following invariant must hold for all fork* sent *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 instance, in Fig. 4, only $Fork_1$ is in *sent* state in the path between Buf_1 and Buf_3, and Invariant 3 is honored.

A buffer contains a sequence of slots and honors a first-in-first-out principle. We propose the following *for each buffer*:

Invariant 4. *For the sequence of slots $s_1, s_2, ..., s_N$ that is originated 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 slot inside a buffer is full, then the next slot in the same buffer must also be full. For instance, if the upper slot of *Buf* in Fig. 1 is full, then its lower slot must be full.

B. Invariants from Reconvergent Paths

We characterize the state of a dataflow circuit by the *full* and *sent* states. 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 (LD) 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 buffered in Buf_3. Whenever the circuit terminates, it sinks both *i* and *x* to exit.

The target property is honored throughout the execution of the circuit. However, if we attempt to prove it using 1-step induction (even *after* adding the invariants of Section III-A), the induction engine will generate a counter-example such as the one in Fig. 3 (i.e., the target property holds in the first state s1 but not in the second state s2, 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 *reconvergent paths* (i.e., paths that originated from the same fork, and reconverged at the same join), as illustrated in Fig. 4, such that both paths start from $Fork_1$ and end at Br_2. For any pair of reconvergent paths, the number of tokens that are issued to one path *must eventually be identical* to the other.

Prior work [8] proposed an automatic method for discovering inductive invariants to systematically rule out unreachable states of this form; it introduces an *auxiliary* state variable λ_{ch} to denote the number of tokens that have been transferred in a channel *ch* up to a given point in time. How each unit governs the token transfer λ in its in-/output channels can be formulated as separate rules; invariants that describe the token count relations between reconvergent paths are generated from these rules after eliminating λ using the Gaussian elimination method. We adapt this method to dataflow circuits: Table I

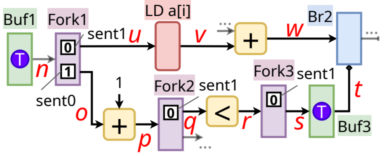


Fig. 4. A pair of reconvergent paths from the circuit in Fig. 3. Path 1: Fork1, LD, add, Br2; path 2: Fork1, add, Fork2, compare, Fork3, Buf3, Br2.

describes a set of relations for each unit; we automatically derive a set of invariants and add them to the verification model.

As an example, we would like to derive a structural invariant for the reconvergent path structure in Fig 4, which is part of the same circuit in Fig 3. All channels in this figure are labeled with a red italic letter. We could formulate the following equations for the units in Fig. 4. 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 rules out the state s1 in Fig. 3 by requiring the count of full slots and *sent* states in eager forks in the two reconvergent paths in Fig. 4 to be balanced.

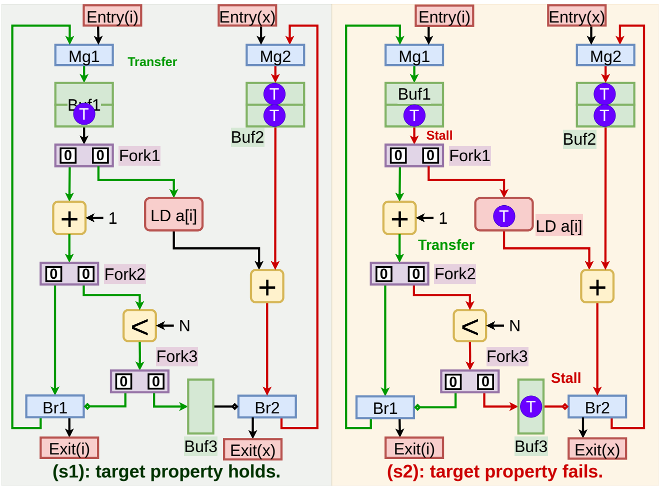
C. Invariants from Bounded Structures

The invariants of Section III-B ruled out many structurally unreachable states. However, they 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. 6 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 [19]; we define them as *in-order graphs*.

TABLE I
SUMMARY OF FLOW RELATIONS

Unit Type	Relation
Join	$\lambda_{in,k} = \lambda_{out}$, for each input k.
Fork	$\lambda_{in} + sent_k = \lambda_{out,k}$, for each output k. When sending tokens that determine the control flow: $\lambda_{in}^{+/-} + sent_k^{+/-} = \lambda_{out,k}^{+/-}$, where $sent_k^{+} = d_{in} \rightarrow sent_k$ and $sent_k^{-} = \overline{d_{in}} \rightarrow sent_k$.
Slot	$\lambda_{in} = full + \lambda_{out}$. When the token it holds determines the control flow, we separately specify $full^{+}$ and $full^{-}$: $\lambda_{in}^{+/-} = full^{+/-} + \lambda_{out}^{+/-}$.
Branch	$\lambda_{in,data} = \lambda_{in,cond} = \lambda_{out,true} + \lambda_{out,false}$, $\lambda_{in,cond}^{+/-} = \lambda_{out,true/false}$.
Mux	$\lambda_{in,sel} = \lambda_{in,true} + \lambda_{in,false} = \lambda_{out,0}$, $\lambda_{in,sel}^{+/-} = \lambda_{in,true/false}$.
Merge	$\lambda_{in,0} + \lambda_{in,1} = \lambda_{out}$.



(s1): target property holds.

(s2): target property fails.

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

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. 6 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:

> **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

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

which implies that it is impossible to occupy both of the slots in Buf_2 with tokens. This rules out the state s1 in Fig. 5. We developed a procedure for IOG enumeration based on a breadth-first search; we

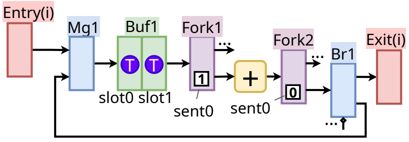


Fig. 6. A part of the circuit from Fig. 3 that forms an in-order graph.

generate Invariant 5 for all of them.

To exclude unreachable dataflow 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:*

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

where f is a fork in p.

Invariant 6 describes that, if two slots on an IOG are both full, then (1) there must be a path p connecting them, (2) the slot s at the beginning of p must be copied by a fork f on this path, (3) s belongs to the set of copied slots of f (see Section III-A). For example, in the IOG in Fig. 6, Invariant 6 rules out the situations of having $slot_1$ and *Entry* both occupied by tokens, as there is no fork to duplicate tokens from *Entry* (note that Invariant 5 does not rule out this state).

D. Invariants for Token Ordering

Our invariants so far describe rules for token counts and distributions. Yet, they do not account for token *values*, which is critical when reasoning about control flow decisions in a pipelined circuit. We devise the corresponding invariants in this section.

Fig. 7 illustrates a dataflow circuit that accumulates the squared values of the loop iterator. The execution starts by injecting the initial values of the iterator *i* and accumulator *x* into the loops; 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 data input Mux_1 and Mux_2 should consume tokens. The iterator *i* is stored in Buf_2 and 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 unit that produces non-deterministic control decisions instead of actual comparisons [5]. The decider dictates whether the cyclic paths (the IOGs a, b, and c) should repeat the loop iteration or dispatch the tokens to the exits. Condition C indicates to continue the loop iteration; E indicates entering the loop for Mux and exiting for Br. Accumulator *x* is stored in Buf_1 and initially has to wait for the first token from $Pipeline_1$ for 4 cycles; while waiting, the control decisions are buffered in Buf_{entry} and Buf_{exit}, which keep track of the control decisions that *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. For example, in Fig. 7, instead of having "C, C, C" in Buf_{entry}, assume a sequence "C, E, C"; all other tokens remain at the same location and have the same 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 input select token of Mux_1 will remain to be E, and the token in Buf_1 will never advance; the induction engine concludes that the circuit deadlocks and misses the opportunity to verify useful properties (e.g., the output channel of $Pipeline_1$ never stalls). Yet, this scenario is 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 not be a token in Buf_1).

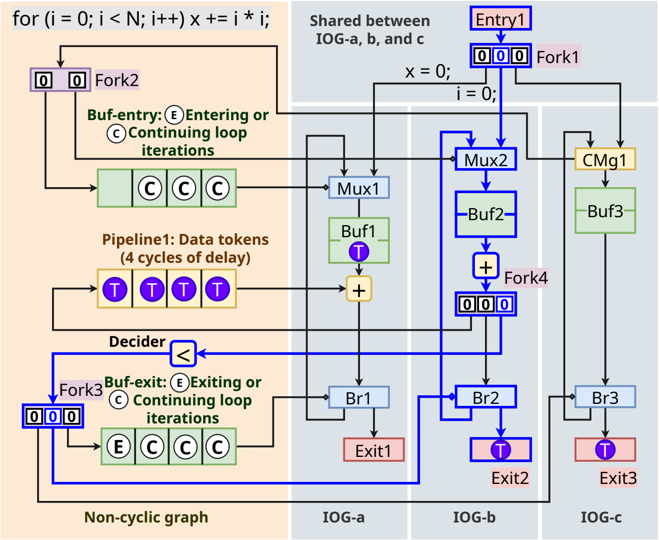


Fig. 7. Interaction between different in-order graphs. The non-cyclic graph is composed of the units that pass tokens in between different in-order graphs.

We propose invariants to eliminate unreachable states associated with this value-ordering problem: we restrict the token 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 $s_i.value$, which represents the value of the token stored in the 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. *For all $CMg_{entry}, Mux_{entry} \in BB_{entry}$ and all $p_{cm} \in Paths(CMg_{entry}, Mux_{entry})$, the sequence can be represented using regular language as follows:*

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

where BB_{entry} is the header BB[1] *of the outermost loop, "$*$" 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 (condition 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:

[1] A loop header BB is the entry to the loop that is targeted by the back edge; the exit BB is the BB that has an outgoing edge from the loop [20].

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

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

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 example, 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 exits from IOG_a, no further token can be consumed from Buf_{entry}, Buf_{exit}, and $Pipeline_1$, 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 all paths between Decider and Br only to those that are actually possible in sequential program execution:

Invariant 9. *For all $Dec_{exit}, Br_{exit} \in BB_{exit}$ and all $p_{db} \in Paths(Dec_{exit}, Br_{exit})$, the sequence in the path can be represented using regular language as follows:*

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

where BB_{exit} is the exit BB of the outermost loop.

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 invariant describes that, if any slot holds the outermost loop exit decision, then no other slot can send data tokens to this slot.

Invariant 10. *For all slots that hold outermost loop exit condition $s_{db} \in p_{db}$, all slots that are s_{db}'s ancestors*[2] *$s_{ancestor} \in IOG'_{Dec}$ 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; we construct IOG'_{Dec} by adding Br_1, Buf_{exit} and their channels to IOG_{Dec}. Buf_{exit} receives tokens from IOG_{Dec}. We see a condition E inside Buf_{exit}: since no ancestor of the slot that holds condition E in Buf_{exit} on IOG'_{Dec} that holds a token that can be sent to Buf_{exit}, therefore Invariant 10 is honored.

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

Invariant 11. *Consider every slot s and path p_{sd} such that $p_{sd} \in Paths(s, Dec), p_{sd} \subset IOG_{Dec}$, and s is the only slot in p_{sd}. For all $s_{db} \in p_{db}$, the following must hold:*

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

For example, if Buf_{exit} holds a token with value E and Buf_2 also holds a token, then in this case Buf_2 must hold a (duplicated) token that the decider Dec evaluates as E.

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. The research artifact (our code and benchmarks) is publicly available [21].

A. Methodology and Benchmarks

We incorporate our strategy in a verification framework [5] targeting dataflow circuits produced by Dynamatic, an open-source HLS compiler that translates C/C++ into dataflow circuits [22]. This framework automatically generates an abstract circuit model based on SMV [23] and a set of properties to prove (discussed in Section II-B). The original framework verifies these properties using BDD-based model checking in nuXmv [24]. Whenever a dataflow channel property evaluates to *true*, the channel logic is simplified; if the property is *false*, the channel remains intact.

TABLE II
Model Statistics & Invariant Verification Runtime

Benchmark	#Slots	#Sents	#Units	#Ch.	No Reconv. 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)

For a fair comparison, 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 cannot optimize the corresponding channel logic so that we do not compromise functional correctness. 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.

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 [8] 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-B: 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 Reconv. Paths*, showing the time needed to verify all invariants except those of Section III-B (i.e., reconvergent 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 ommitted as they originate directly from the well-known composability of latency-insensitive units that makes dataflow circuits correct by construction [9], [25].

We report the verification results and runtime of nuXmv on an AMD Ryzen 7 CPU at 1.90 GHz. Each verification run is timed out after 72 hours (if 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 (v2019.1) and targeting a Kintex-7 Xilinx FPGA with a target clock period of 4 ns [26].

Since all the optimization strategies we consider focus on the datapath of dataflow circuits, we consider exclusively the datapath resources and omit those of the memory interface; others optimized and evaluated this orthogonal aspect [27]–[29]. To ensure that our modifications have not compromised circuit behavior, we perform functional verification in ModelSim through simulation: on a set of representative stimuli, we confirm that the number of clock cycles for

[2] Unit a is an ancestor of unit u if $Paths(a, u) \neq \{\varnothing\}$

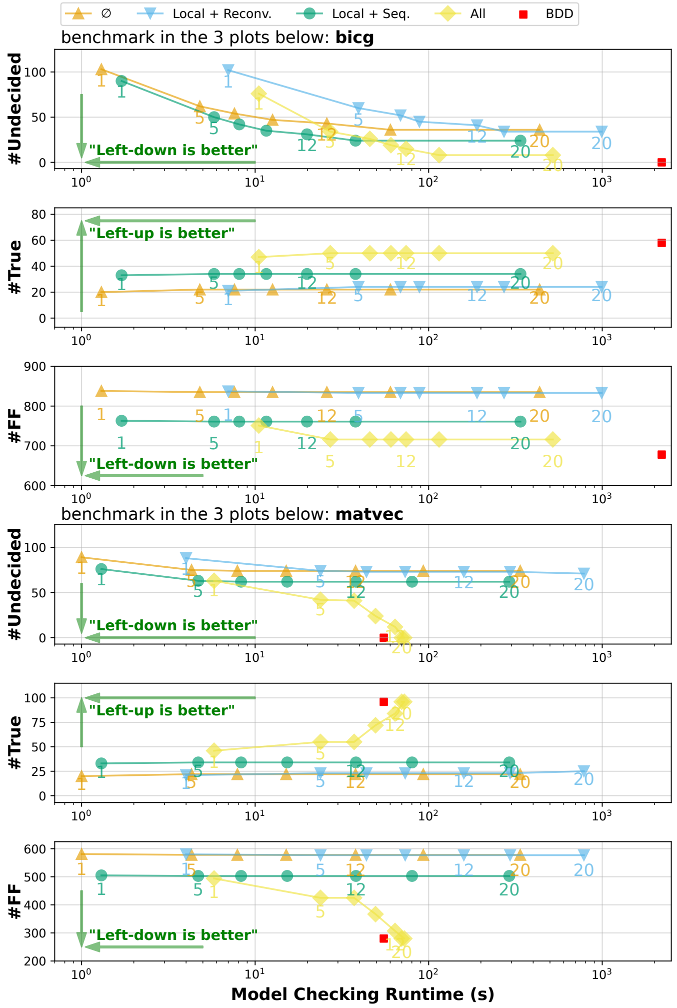


Fig. 8. Effect of different induction depths k and different added sets of invariants on the number of undecided properties (#Undecided), number of true properties (#True), the number of registers (#FF), and model checking runtime in seconds for benchmarks *bicg* and *matvec*.

the circuit's execution remains identical (i.e., our optimizations do not impact performance) and that the computed outputs are correct (i.e., our optimizations do not modify the circuit's functionality).

Our benchmarks are typical HLS kernels that originate from a standard HLS suite [30] and have been previously used for evaluating the effectiveness of dataflow circuits in HLS [4], [5], [13], [31]. Table II details the characteristics of our benchmarks: *#Slots* and *#Sents* are the total number of slots and *sent* 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

In this section, we compare our induction-based strategy with a BDD-based approach [5] in terms of verification runtime, the number of verified properties, and area optimization effectiveness.

Table III compares different verification techniques, as indicated in the *Techn.* column: *No Opt* corresponds to generic dataflow circuits that are not optimized via formal verification, *BDD* is BDD-based model checking [5], 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 proven to hold in all reachable states; only they can be used for circuit 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 k. 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., $\#Undecided = 0$) 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* example above, the small reduction in LUT savings comes at a 100× 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.

C. Results: Induction Depth Analysis

In this section, we explore the effects of induction depth k on its verification capabilities.

In Fig. 8, we evaluate property checking and synthesis results under different invariants and with different k. We consider *matvec* and *bicg* as these benchmarks are moderately sized: they are small enough to be handled by *BDD* and large enough to see the differences when varying different induction parameters. The curve labeled with $\varnothing$ (in orange) corresponds to induction with no invariants; the curve *Local + Reconv.* (in blue) corresponds to *local invariants* (Section III-A) and *invariants from reconvergent paths* (Section III-B), which qualitatively corresponds to a prior invariant generation strategy [8]; the curve *Local + Seq.* (in green) corresponds to all proposed invariants but *excluding* the invariants from reconvergent paths; the curve *All* (in yellow) corresponds to all proposed invariants; the red square corresponds to BDD-based model checking. The used induction depth k is labeled next to the points. For the utilization results, Fig. 8 only includes the register count (FF) since the trend of LUTs and FFs are similar (seen from Table III). In all y-axis metrics, we desire to be as close as possible to the *BDD* point (achieving the largest FF savings, no undecided properties, and proving all true properties); in the x-axis, we aim to be as far to the left as possible from the *BDD* point (i.e., reducing verification runtime).

Fig. 8 shows that, by naively increasing the induction depth in $\varnothing$ (i.e., no invariants), the run-time increases exponentially with very few or no improvements in the number of proven properties, undecided properties, nor the synthesis results. This shows that, without the addition of invariants to restrict the unreachable state space, only ”*simple*” properties can be concluded, and little benefit can be seen by naively increasing the depth k. With the inclusion

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%)

of all proposed inductive invariants, many of the properties can be proved already in $k = 1$; in the case of *matvec*, when many candidate properties are true, we can see the number of proven properties (*#True*) increases sharply with increasing *k*, and finally the number of proven properties reaches that of *BDD* at $k = 15$. This indicates the relevance of invariants, which are our main contribution, to reaping the benefits of k-induction-based verification.

D. Results: Effectiveness of Inductive Invariants

Section IV-B and IV-C have demonstrated the key role of the inductive invariants from Section III in induction-based verification; the question is whether *all* these invariants are actually useful and needed. We investigate this aspect in this section.

As shown in Fig. 8, verification in the absence of invariants is typically the fastest; adding invariants while maintaining a constant *k* increases the runtime. Consider, for instance, the top graph and the points for $k = 1$; the order from left (fastest) to right (slowest) is: no invariants (orange), partial invariants (green, blue), and all invariants (yellow). However, it is important to note that verification with all invariants systematically achieves the best results: it proves the largest number of *true* properties and reduces the FF count the most. What is more, its effectiveness rapidly increases with the increase of *k* (see *matvec*: the yellow curves quickly converge to the red points of *BDD*); in contrast, verification without and with partial invariants does not improve with the increase of *k* (i.e., the curves are flat and the results stay nearly the same despite the significant runtime increase). This suggests that these systems lack critical information to prove anything beyond the most trivial properties and indicates the relevance of employing *all invariants* we propose in Section III.

V. RELATED WORK

With the increased interest in using dataflow circuits in HLS [3], [10], [13], [32], many authors explored a variety of dataflow optimizations [19], [29], [31], [33]–[35]; all these efforts complement our work and could benefit from removing redundant dynamism using our strategy. Some HLS frameworks consider dataflow designs in coarser granularities to support task-level parallelism [1], [36]; our work could provide useful insights for the verification of these designs, e.g., by analyzing their reconvergent paths. Several works use formal verification for HLS transformations [37], [38]; in contrast, we apply formal verification to optimize HLS-produced circuits.

Automatically generating inductive invariants has been the subject of many research studies [8], [16], [18], [39]. To our best knowledge, the most recent work that targets hardware verification is Chatterjee et al. [8] (discussed in Section III-B). Compared with their strategy, we exploit the regularities in dataflow circuits generated from high-level programs to take full advantage of the benefit of k-induction (their approach qualitatively corresponds to the *Local + Reconv.* curves in Fig. 8; our solution yields better results, since we explore the techniques discussed in Section III-C and III-D).

Property-directed reachability (PDR) is a more recent model-checking algorithm [40]. Its goal is to incrementally construct an inductive invariant that implies the target safety property. Our strategy could enhance this approach by directly providing an already pruned state space for the PDR algorithm to complete the proof.

Many techniques discussed in this paper are rooted in the modeling, analysis, and verification of Petri nets [8], [14], [15], [41]. Our work could provide insights into what important behaviors to preserve when building Petri net models for verification purposes. Several approaches [42], [43] use Petri nets to remove redundant logic in a particular class of latency-insensitive systems (i.e., loops with a fixed computational rate and no control flow). Our approach is more general as it supports control flow constructs that appear in high-level languages (e.g., nested loops, conditional execution).

Sequential synthesis aims at circuit optimization without altering sequential behavior, which shares a similar goal with our approach [16], [17]. Yet, these approaches are based on either pure k-induction or mining inductive invariants from gate-level descriptions. As seen in Section IV-C, without invariants, k-induction cannot conclude useful properties for optimizing dataflow circuits.

VI. CONCLUSION

In this work, we present an automated strategy for generating inductive invariants to support k-induction-based verification on HLS-produced dataflow circuits. Our invariants rule out unreachable states that violate the structural properties of individual units, of the entire dataflow 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. 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] *Vitis HLS*, Xilinx Inc., 2023. [Online]. Available: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls

[2] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems," *ACM Transactions on Embedded Computing Systems*, vol. 13, no. 2, pp. 24:1–24:27, Sep. 2013.

[3] 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.

[4] 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.

[5] 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.

[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.

[7] P. Bjesse and K. Claessen, "SAT-based verification without state space traversal," in *Proceedings of the 3rd International Conference on Formal Methods in Computer-Aided Design*, Austin, TX, Nov. 2000, pp. 409–26.

[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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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*, Austin, TX, Nov. 2007, pp. 165–72.

[17] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Proceedings of the 22nd International Conference on Computer Aided Verification*, Edinburgh, UK, Jul. 2010, pp. 24–40.

[18] 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, 2009.

[19] 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.

[20] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 1st ed. Addison-Wesley Longman Publishing Company, 1986.

[21] J. Xu, "Research Artifact for ICCAD'23: Automatic Inductive Invariant Generation for Scalable Dataflow Circuit Verification," Aug. 2023. [Online]. Available: https://doi.org/10.5281/zenodo.8210941

[22] 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.

[23] 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.

[24] 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 *Proceedings of the 26th International Conference on Computer Aided Verification*, Vienna, Austria, Jul. 2014, pp. 334–42.

[25] J. Cortadella, M. Galceran-Oms, and M. Kishinevsky, "Elastic systems," in *Proceedings of the 8th ACM/IEEE International Conference on Formal Methods and Models for Codesign*, Jul. 2010, pp. 149–58.

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

[27] L. Josipović, P. Brisk, and P. Ienne, "An out-of-order load-store queue for spatial computing," *ACM Transactions on Embedded Computing Systems*, vol. 16, no. 5s, pp. 1–19, Sep. 2017.

[28] L. Josipović, A. Bhattacharyya, A. Guerrieri, and P. Ienne, "Shrink it or shed it! Minimize the use of LSQs in dataflow designs," in *Proceedings of the 18th IEEE International Conference on Field Programmable Technology*, Tianjin, Dec. 2019, pp. 197–205.

[29] J. Liu, C. Rizzi, and L. Josipović, "Load-store queue sizing for efficient dataflow circuits," in *Proceedings of the 21st IEEE International Conference on Field Programmable Technology*, Hong Kong SAR, China, Dec. 2022, pp. 1–9.

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

[31] C. Rizzi, A. Guerrieri, P. Ienne, and L. Josipović, "A comprehensive timing model for accurate frequency tuning in dataflow circuits," in *Proceedings of the 32nd International Conference on Field-Programmable Logic and Applications*, Belfast, UK, Aug. 2022.

[32] R. Li, L. Berkley, Y. Yang, and R. Manohar, "Fluid: An asynchronous high-level synthesis tool for complex program structures," in *Proceedings of the 27th International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 2021, pp. 1–8.

[33] 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.

[34] J. Cheng, L. Josipović, G. A. Constantinides, P. Ienne, and J. Wickerson, "Combining dynamic & static scheduling in high-level synthesis," in *Proceedings of the 28th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Seaside, CA, Feb. 2020, pp. 288–98.

[35] J. Cheng, L. Josipović, J. Wickerson, and G. A. Constantinides, "Dynamic inter-block scheduling for HLS," in *Proceedings of the 32nd International Conference on Field-Programmable Logic and Applications*, Belfast, UK, Aug. 2022.

[36] 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*, Virtual Event, May 2021, pp. 204–13.

[37] F. Faissole, G. A. Constantinides, and D. Thomas, "Formalizing loop-carried dependencies in Coq for high-level synthesis," in *Proceedings of the 27th IEEE Symposium on Field-Programmable Custom Computing Machines*, San Diego, CA, Apr. 2019, pp. 315–15.

[38] Y. Herklotz, J. D. Pollard, N. Ramanathan, and J. Wickerson, "Formal verification of high-level synthesis," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–30, 2021.

[39] 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.

[40] 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.

[41] J. Cortadella, "Combining structural and symbolic methods for the verification of concurrent systems," in *Proceedings of 1998 International Conference on Application of Concurrency to System Design*, Fukushima, Japan, Mar. 1998, pp. 2–7.

[42] J. Carmona, J. Julvez, J. Cortadella, and M. Kishinevsky, "Scheduling synchronous elastic designs," in *Proceedings of the 9th International Conference on Application of Concurrency to System Design*, Augsburg, Germany, May 2009, pp. 52–59.

[43] B. Xue, S. K. Shukla, and S. S. Ravi, "Minimizing back pressure for latency insensitive system synthesis," in *Proceedings of the 8th ACM/IEEE International Conference on Formal Methods and Models for Codesign*, Grenoble, France, 2010, pp. 189–98.