Eliminating Excessive Dynamism of Dataflow Circuits Using Model Checking

Jiahui Xu
ETH Zurich
Zurich, Switzerland

Emmet Murphy
ETH Zurich
Zurich, Switzerland

Jordi Cortadella
UPC Barcelona
Barcelona, Spain

Lana Josipović
ETH Zurich
Zurich, Switzerland

ABSTRACT

Recent HLS efforts explore the generation of dynamically scheduled, dataflow circuits from high-level code; their ability to adapt the schedule at runtime to particular data and control outcomes promises superior performance to standard, statically scheduled HLS solutions. However, dataflow circuits are notoriously resource-expensive: their distributed handshake mechanism brings performance benefits in some cases, but causes an unneeded resource overhead when general dynamism is not required. In this work, we present a verification framework based on model checking to systematically reduce the hardware complexity of dataflow circuits. We devise a series of formal proofs that identify the absence of particular behavioral scenarios and use this information to replace the generic dataflow logic with simpler and cheaper control structures. On a set of benchmarks obtained from high-level code, we demonstrate that our technique significantly reduces the resource requirements of dataflow circuits (i.e., it results in LUT and FF reductions of up to 51% and 53%, respectively), while still reaping all performance benefits of dynamic scheduling.

CCS CONCEPTS

• **Hardware** → **Model checking**; **Datapath optimization**; • **Computer systems organization** → *Data flow architectures.*

KEYWORDS

High-level synthesis, dataflow circuits, model checking

ACM Reference Format:
Jiahui Xu, Emmet Murphy, Jordi Cortadella, and Lana Josipović. 2023. Eliminating Excessive Dynamism of Dataflow Circuits Using Model Checking. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '23), February 12–14, 2023, Monterey, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3543622.3573196

1 INTRODUCTION

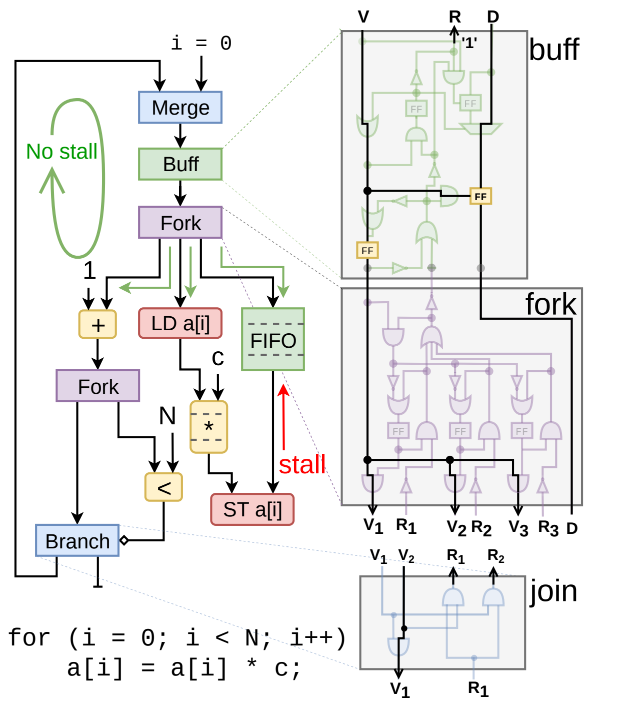
Dataflow circuits [1, 2] have recently been explored as an alternative to standard, statically scheduled HLS solutions due to their ability to achieve high throughput in irregular and control-dominated programs. In contrast to the circuits produced by classic HLS techniques (in which operations are triggered through a centralized, preplanned controller), dataflow circuits are built out of units that communicate using point-to-point pairs of handshake signals; data is propagated from unit to unit as soon as memory and control dependencies allow it and stalled by the handshaking mechanism otherwise [3, 4]. This distributed control mechanism effectively implements a dynamic schedule, adapted at run-time to particular data and control outcomes; it enables dataflow circuits to support new behaviors and achieve high parallelism in situations where standard HLS produces conservative results [4–6].

The same distributed handshake mechanism that provides dataflow circuits with the dynamism and scheduling flexibility that various applications require also causes a notable resource cost [5, 7]: the two-way communication network spans through the entire circuit and introduces local synchronizing, steering, and replication logic practically among all neighboring dataflow operators. While these individual constructs are fairly simple, they collectively represent a significant portion of the circuit's resources and may even degrade the circuit's critical path. In some cases, this overhead is acceptable as it is the reason for the superiority of dynamic scheduling. However, there are many situations where the computational paths never really profit from the flexibility of dataflow computation; the complete and generic dynamism becomes an expensive overkill that reduces the advantages of dynamic scheduling and limits the complexity and variety of programs that this paradigm can support. In such situations, removing the expensive dynamism would be profitable; yet, any such simplification must guarantee that all relevant circuit behaviors and correctness are preserved.

In this work, we present a general strategy to rip off redundant control logic from dataflow circuits. We build an HLS-based framework to automatically translate dataflow circuits, obtained from high-level programs, into models describing their sequential behavior. We develop an extensible set of formal properties, based on model checking, to prove the presence or absence of particular behaviors in all possible execution scenarios. We use the results of these proofs to restrict the flexibility of generic dataflow control logic while neither limiting the circuit's behavior nor compromising its correctness. We show that a small set of simple properties already achieves significant circuit simplifications: the generic control logic templates are only kept where they are actually required, whereas the remaining logic is reduced to simpler and more performance-efficient structures. We envision that our framework can serve as a foundation to develop even more powerful circuit optimization strategies based on formal verification.

The rest of this paper is organized as follows: Section 2 illustrates the expensiveness of dataflow computation and the need to reduce this cost. In Section 3, we describe the dataflow circuits and model checking techniques that our work relies on. In Section 4, we present a formal description of dataflow circuit constructs and characteristics. We use these insights in Section 5 to develop a set of formal proofs that verify the presence of particular behavioral

FPGA '23, February 12–14, 2023, Monterey, CA, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9417-8/23/02.
https://doi.org/10.1145/3543622.3573196



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

Figure 1: Expensive dataflow logic is not always needed. The dataflow circuit on the left is built out of generic dataflow constructs that communicate with a set of handshake signals; a detailed description of three of them is shown on the right (shaded logic), and other units follow the same trends. However, in many situations, the generality of this logic is unneeded: if one can *prove* the absence of particular behaviors in *every possible* execution scenario, the logic could be simplified accordingly. In this example, none of the units except the FIFO ever experiences a stall from its successor and the related logic can be omitted (e.g., the fork, buffer, and join on the right can be reduced to much simpler logic, shown in black).

properties. We discuss the scalability of our approach in Section 6, present our complete verification framework in Section 7, and evaluate it in Section 8.

2 DO WE ALWAYS NEED EXPENSIVE DATAFLOW LOGIC?

The dataflow circuit on the left of Figure 1 implements the functionality of the code in the figure (i.e., resizing the elements of a vector). The initial value of the iterator *i* is injected into the circuit through the merge to trigger the computation; it is incremented while circulating through the loop. In every iteration, the iterator is forked to memory to retrieve the value of *a[i]*, which is then sent to a pipelined multiplier to calculate the resized value of the vector element and stored back into memory; the stored addresses are accumulated in the FIFO during the long-latency multiplication, to ensure the best possible throughput and performance [8].

Although not explicitly shown in the circuit, all dataflow units are equipped with bidirectional handshake signals: a *valid* signal indicates data validity and triggers the successor units, whereas a *ready* signal indicates the availability of a unit to accept data from its predecessors. The gate-level description of the fork, buffer, and part of the branch (i.e., join) is shown on the right. Other units contain similar logic constructs, which implement general latency-insensitivity and can handle any data arrival and dispatch times.

Although all units are fully equipped with a latency-insensitive interface, the *only* instance of a stall occurs between the store and the FIFO (i.e., the store stalls the incoming addresses until the multiplier produces the matching data). All other dataflow units never exploit their latency insensitivity and their implementation can be simplified accordingly, as shown by the fork, buffer, and join in the figure: as there is never any backpressure, the ready signals and the logic to compute them can be entirely omitted; as the fork dispatches data to all successors simultaneously, the valid computation can be reduced to a wire. Other units are amenable to similar simplifications; the behavior of the simplified circuit is identical to the original one, but at a significantly reduced size and complexity. The intuition behind these simplifications is clear, however, any attempt must guarantee that the simplified implementations are correct for any possible circuit execution and for any subtle dynamic change. In the rest of this paper, we present a methodology to prove such behavioral properties via model checking and use this information to systematically simplify generic dataflow logic.

3 BACKGROUND

In this section, we describe the characteristics of the dataflow circuits generated from the high-level code. We discuss the concept of model checking for verifying safety properties and understanding if some circuit transformations can be performed without risk.

3.1 Dataflow Circuits

Dataflow circuits are built out of *units* that are latency insensitive and communicate with the predecessor and successors using *channels*, composed out of handshake signals [1, 2]. As soon as the relevant data and control dependencies allow, the data *token* is sent across the channel (that is, data is exchanged).

Recent works have addressed generating dataflow circuits from high-level programs [3, 4]; without loss of generality, we here consider an approach that translates C/C++ code into synchronous dataflow circuits [4]. What is relevant here is that this strategy organizes dataflow units into *basic blocks* (BBs), representing straight pieces of code with no conditional execution; all control flow (i.e., conditional statements) form control flow edges that connect different BBs into a control-flow graph (CFG). Whenever a BB has multiple successor BBs, a condition that is computed within it determines which successor (with all of its dataflow units) will execute.

Apart from standard computational blocks, dataflow circuits require specialized units to appropriately steer data from their producers to their consumers: (1) A *join* is used inside operators that need to synchronize multiple operands (e.g., the adder, comparator, and branch of Figure 1). (2) A *fork* replicates data to its multiple outputs (as shown for the iterator in the same figure). (3) A *branch* sends a data to one of its successors based on the condition computed in its BB (in the previous example, it decides whether to start a new iteration or exit the loop depending on the comparison of the iterator with the loop bound). (4) A *merge* receives data from one of its predecessors (i.e., that in the preceding active BB). All other units are a composition or extension of these basic constructs.

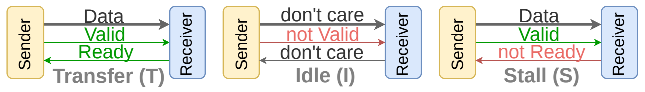


Figure 2: A typical handshake communication protocol. A *valid* signal indicates that the sender is sending a *token* to the receiver, and the *ready* signal indicates the receiver's availability to accept it. The figure shows three possible states on the communication *channel* between the sender and receiver, dictated by *valid* and *ready* values.

3.2 Handshake Communication Protocol

Latency-insensitive communication enables dataflow circuits to make scheduling decisions during runtime. Figure 2 shows an example of a typical latency-insensitive protocol [1] with its three possible channel states, namely **Transfer**, **Idle**, and **Stall**. The state of a channel can be identified from the value of the handshake signals *valid* and *ready*. The channel is said to be in the **Transfer** state when the sender has valid data and the receiver is ready to receive it ($valid \land ready$), the **Idle** state when the sender does not have valid data ($\neg valid$), and the **Stall** state when the sender has valid data but the receiver does not accept it ($valid \land \neg ready$).

All dataflow units are equipped with internal logic that produces the appropriate valid and ready values so that they can communicate with their predecessors and successors following this protocol; the generic logic supports all three states as, in principle, each of them could occur in every channel. Yet, as we have seen in Figure 1, this is not always the case; there are many situations where some of these states never occur and the generality of the dataflow logic that supports them is not required. We explore this insight in this paper: we aim to *prove* the presence or absence of particular behaviors (i.e., particular communication states) and use this information to simplify the dataflow circuit implementation.

3.3 Verifying Properties Using Model Checking

Model checking is the most widely used technique for the automatic formal verification of finite transition systems [9, 10]. During the model verification process, the design and specifications are usually represented as finite automata, and the properties to check are described using temporal logic. The reachable states of the system are then explored to verify the properties. If a particular property is not honored in the model, a counterexample trace is generated in the form of a sequence of states.

Model checking can be performed using different algorithmic techniques [9]: (1) *explicit model checking*, in which all reachable states are enumerated, and (2) *symbolic model checking*, that uses symbolic representations of the set of states rather than explicit enumeration. In this work, we opt for symbolic model checking, as it allows us to conveniently describe all possible channel states and data transfers in a dynamically scheduled dataflow system. Symbolic model checking is typically based on manipulating binary decision diagrams (BDDs): when verifying safety properties, the model checker explores the entire state space in a way similar to a breadth-first search. Yet, this approach is unscalable and impractical when verifying larger systems [9]. This issue can be mitigated via *compositional model checking* [11, 12]: systems are decomposed into subsystems that are individually verified, while the remainder of the system is appropriately abstracted away. We will exploit this technique in our work to ensure the scalability of our approach.

```
MODULE join(pValid0, pValid1, nReady0)
DEFINE valid0 := pValid0 & pValid1;
DEFINE ready0 := nReady0 & pValid1;
DEFINE ready1 := nReady0 & pValid0;
```

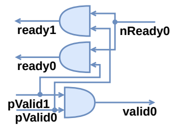


Figure 3: An SMV description of a join, obtained directly from its gate-level description on the right. The unit is fully specified with a predefined firing policy, latency, II, and capacity.

4 MODELING AND ABSTRACTION OF DATAFLOW CIRCUITS

Our goal is to identify the absence of particular dataflow circuit behaviors in *all* execution scenarios and use this information to restrict the generality of the dataflow logic. For this purpose, we devise a circuit model that captures all behavioral and timing properties of the original dataflow circuit and allows us to reason about its behavior irrespectively of particular data values.

4.1 Constructing Arbitrary Dataflow Units

We describe a dataflow graph as a system of transitions: all data is abstracted away and all dataflow channels simply represent token exchanges among neighboring dataflow units.

We characterize each dataflow unit with the following properties: (1) *Firing policy.* Each dataflow unit produces one or multiple tokens following a predefined rule, as described in Section 3.1. (2) *Capacity.* This value indicates the maximal number of tokens that a dataflow unit can hold [8]; it is determined by the number of register slots of the unit (e.g., sequential computational stages or the number of FIFO or buffer slots). (3) *Initiation Interval (II).* This value indicates the rate with which each unit input can accept tokens from its predecessor (i.e., the number of clock cycles between accepting two consecutive tokens). (4) *Latency.* This value indicates the number of clock cycles that a unit requires to produce the output(s) after receiving its input(s). Typically, it corresponds to the unit's sequential stage count. For most dataflow units, these properties are known and predefined; we obtain them by directly translating the dataflow unit RTL description into the corresponding transition relations, while omitting the datapath computation. An example of a join description in a typical verification modeling language [10] is shown in Figure 3; it immediately derives from its gate-level description shown in the figure.

Yet, in some cases, these properties depend on particular data or control outcomes and may be variable or unknown: (1) The II of some units may vary with the timing or data values of particular events (e.g., a memory interface may stall incoming memory requests due to memory congestion or hazard detection). (2) Unit latency may vary with the processed data (e.g., a memory interface may hold a memory request longer due to a dependence; a variable-latency computational unit may take a variable number of cycles to compute the result depending on the input data [13]). (3) The control flow decisions determine the firing of particular units, i.e., a unit will fire only when the control flow determines the execution of the BB it belongs to (see Section 3.1); they are typically statically undeterminable and depend on the actual data values.

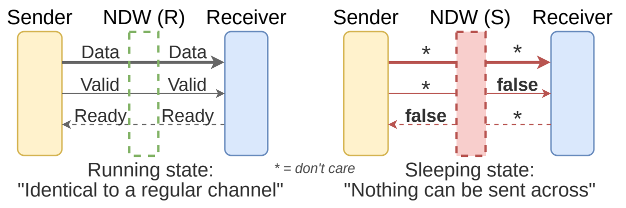


Figure 4: A non-deterministic wire (NDW). In its *running* state, an NDW behaves like a regular channel and directly propagates the valid and ready signal to the receiver and sender, respectively. In its *sleeping* state, the NDW blocks its predecessor from sending any tokens across the channel. We use this construct to model units with variable or nondeterministic properties (e.g., an unknown initiation interval of the receiver or an arbitrary delay of a sender).

We incorporate these behaviors in our model by representing each variability with a nondeterministic construct that captures *all possible* values of the unknown parameters: (1) A *nondeterministic wire (NDW)* models an arbitrary stall on any dataflow channel. The specification of an NDW is shown in Figure 4: it stalls for an arbitrary number of cycles but specifies that, at some point, the stall will disappear, thus ensuring that the unit eventually consumes the token. (2) A *nondeterministic decider (NDD)* produces a non-deterministic boolean value (i.e., true or false); by producing any sequence of boolean values, it models all possible control flow sequences. A decider is shown in Figure 5: whenever it receives its dataless input token, it outputs a token with a randomly generated boolean value.

We employ these constructs as follows: (1) We place an NDW at each unit input that may exhibit variable II; the NDW dictates how long a unit will stall the data coming from the predecessor and, thus, the rate with which the data will be consumed. Assuming an unbounded NDW, it captures any possible stall time (i.e., any possible II). (2) We place an NDW at the output of every variable latency unit; it dictates the time a token will reside inside the unit and determines the moment the unit will output the token. (3) We insert an NDD at the output of each dataflow unit computing the BB conditions; we rely on classic compiler analysis to identify these places as condition operands of branch instructions [14]. The output value of the NDD replaces the control flow condition computed in the original circuit, is forked to all branches of the BB, and regulates the firing of their outputs (thus determining the next BB to execute).

The nondeterministic constructs above model all possible values of the property they describe; typically, this is a superset of the behaviors that can actually occur in execution. This may lead to overconservative assumptions when verifying properties (e.g., by accounting for delays, IIs, or control flow sequences that never happen in practice). Thus, we constrain these values whenever the information to do so is available (e.g., if an II or latency of a unit are within a predetermined interval, we assign the corresponding bounds to the nondeterministic construct). This restricts the complexity of our model while still capturing all achievable situations.

4.2 Use Case: Describing a Dynamic Memory Interface

To give the reader an intuition of the usage of the non-deterministic constructs above, we here describe our abstract model of a *load-store*

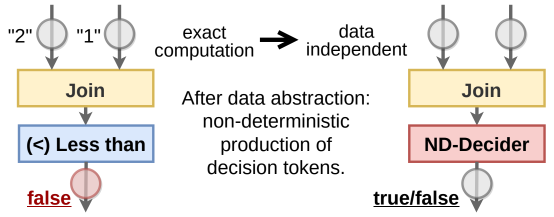


Figure 5: A non-deterministic decider (NDD). This construct receives dataless tokens and produces a token with a non-deterministic boolean value; we provide it to units with conditional behavior (e.g., branch) and use it to model any possible control flow sequence.

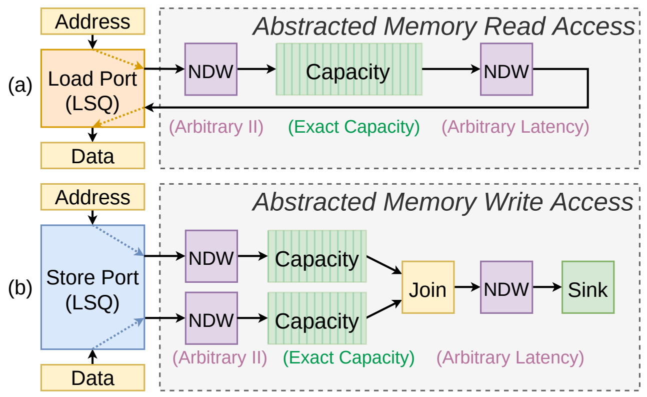


Figure 6: Memory access abstraction for dataflow circuits. A memory access may have variable latency or create circuit stalls, depending on the memory access patterns and possible memory collisions. We use the nondeterministic constructs of Section 4.1 to capture these variabilities and model all possible memory behaviors.

queue (LSQ) [15]. An LSQ is a typical constituent of the dataflow circuit memory interface: it resolves memory dependencies at runtime, allows independent accesses to execute out-of-order, and stalls requests only when there is an address collision; this flexibility is one of the main reasons for the performance superiority of dataflow circuits over the corresponding static designs [8].

Although the exact LSQ functionality is irrelevant here, it is important to understand its mechanism for handling load and store requests. A load request (i.e., address) is sent by the circuit to the LSQ as soon as it has an empty queue slot to hold it; when all memory dependencies are resolved, the request is issued to memory. Eventually, the memory returns the data which is then sent back to the circuit. Similarly, a store address and data are sent from the circuit into the queue when a slot is empty; possibly, at different times. Once they are both available and all address collisions are resolved, the store request is issued to memory.

The firing principle of the load and store access is fairly simple (i.e., return a data token for each address token; send a pair of address-data tokens to memory). However, it is important to note that the timing of these firings is data-dependent: (1) If the LSQ is full of pending requests (arriving from, possibly, various loads and stores in the circuit), all incoming tokens are temporarily stalled (i.e., the II of accepting requests may vary with the LSQ occupancy), (2) The duration of a request's residence in the LSQ differs based on its dependencies on other requests (i.e., the latency changes with actual load-store address collisions). (3) In case of loads, the memory hierarchy may return data in a variable time (depending on the memory organization and data availability). Our memory model must capture *all* these behaviors and variabilities to allow us to reason about the circuit's behavior in *any possible case*.

To this end, we model the load and store LSQ access as shown in Figure 6: A load LSQ access contains an NDW representing its variable II; it is followed by a FIFO of the LSQ capacity, indicating the maximal number of memory requests that the circuit can deposit into the LSQ before receiving back any data; the output NDW captures the latency variability due to hazards and memory latency. The store consists of similar constructs: the NDWs at the address and data input port, as well as the FIFOs of the LSQ capacity, serve the same purpose as in the load; the internal join indicates that the address and data must be available prior to issuing them to memory. The output NDW models the latency variability; once the request is accepted by memory, the token is discarded into a sink. Note that this model allows us to describe every load and store separately and in a composable manner, even though they might insist on the same LSQ, as all variabilities due to various load/store interactions are captured by the constructs above.

This example illustrates the richness of our model in describing a variety of behaviors that may occur in a dynamically scheduled dataflow circuit; it helps us to reason about all reachable circuit behaviors independently of the actual data values and enables us to perform provably safe and correct circuit transformations.

5 PROPERTIES FOR SIMPLIFYING DATAFLOW CIRCUITS

Our dataflow circuit model from the previous section accurately describes all possible circuit behaviors (i.e., it captures all states that the circuit can achieve for any possible data and control outcome). We can use this model to prove that particular circuit properties always hold and simplify the dataflow logic accordingly. In this section, we describe our definition of two simple but powerful formal properties that we aim to verify.

5.1 Proving the Absence of Backpressure

The backpressure mechanism of a latency-insensitive protocol is one of the key enablers of dynamic scheduling, as it supports arbitrary computation stalls which never compromise correctness. Yet, this generic mechanism is also expensive, as each dataflow unit needs to appropriately handle backpressure arriving from its successors and produce the appropriate backpressure signal for its predecessors; we already observed these overheads in Section 2 and Figure 1. As discussed in that example, in practice, only channels where stalling events occur need to be equipped with the backpressure mechanism; if we can prove that backpressure *never* occurs in a particular channel, we can safely remove this redundant logic.

According to Figure 2, if we can show that **Stall** ($valid \land \neg ready$) is an unreachable state for a particular channel, then only the **Transfer** ($valid \land ready$) and **Idle** ($\neg valid$) states need to be considered when constructing the sender and receiver dataflow units that the channel connects. To determine whether the stall state is reachable in a channel formed by a pair of *valid* and *ready* signals, we formulate a simple safety property using temporal logic:

$$\mathbf{G}\ (valid \rightarrow ready). \tag{1}$$

This property indicates that, for a given channel, it should always be the case (i.e., it globally holds) that whenever the sender has a valid token, the successor is able to receive it; in other words, a stall never occurs. If this property holds, the value of the ready signal is either *true* in the case of transfer state or *don't care* in the case of the idle state; thus, the predecessor can simply ignore the actual ready signal value and assume that it is always true. We can therefore remove the logic associated with producing the ready signal in the receiver (and, possibly, simplify the ready computation in the sender, as it now receives a constant true value).

We simplify the circuit by identifying the absence of backpressure as follows: we traverse all dataflow channels and check the validity of the property above. In case it is true, the ready signal entering the data sender from the receiver can be disconnected and replaced with a constant value of 1; the logic to compute the ready signal in the receiver can be removed accordingly. This property can be checked in all channels and in any order, as the channel transformation will never change the circuit's behavior and, thus, never compromise the result of any other property check.

5.2 Proving Trigger Equivalence

The valid signal in a latency insensitive protocol represents the movement of the tokens; it triggers the execution of a dataflow unit so, in contrast to the ready signal, it cannot be entirely removed (as the unit would never trigger) nor replaced with a constant (as the unit would trigger continuously and at incorrect times). However, its generation and distribution to the unit can be simplified: if we can prove that multiple valid signals are equivalent, we can remove the logic associated with their production or synchronization [16].

Consider again the example in Figure 1: the branch in the bottom of the figure contains an internal join whose ready computation logic can be optimized by proving the absence of backpressure, as discussed in the previous section; after this optimization, the *and* gate to produce its valid signal remains intact. If one can prove that the two input valid signals always have the same value, the *and* gate can be omitted, one of the input valid signals disconnected (thus also simplifying the logic of the producer, not visible in the figure), and the other propagated directly to the join output. Ultimately, the join is reduced to a single wire shown in black.

We formulate the temporal logic expression stating that two signals $valid_0$ and $valid_1$ are equivalent in all reachable states as:

$$\mathbf{G}\ (valid_0 \leftrightarrow valid_1). \tag{2}$$

If this property holds, one of the equivalent valid signals can be used to trigger both of the receivers and the logic associated with the production of the other can be omitted.

We simplify the circuit via trigger equivalence proving as follows: for each pair of channels of each BB, we verify the property above; if it holds, we connect a single valid signal to the concerned pair of receivers. Note that there is no formal requirement to perform this check per BB—intuitively, units of different BBs execute at different times as they are triggered by different control flow decisions, thus they are rarely equivalent and their comparison would be futile. In Section 8, we also explore the effect of reducing the equivalence checking only to channels which have the same source or destination dataflow node.

5.3 Extending the Set of Proofs

The properties we discussed in this section are rather straightforward, but extremely effective, as they target the main sources of resource overheads that dataflow circuits exhibit [4]; as we will demonstrate in Section 8, verifying these properties can lead to significant logic simplifications and resource savings. It is important to note that our model and verification framework are in no way limited to the properties we presented here; one could easily build upon them to formulate advanced proofs and reason about more complex behavioral and timing patterns.

For instance, one could perform an even more aggressive investigation of the timing relations of particular non-equivalent valid signals. This information could be used to reconstruct their behavior using a single and cheaper control structure (e.g., a shift register or finite-state machine) that would replace the distributed valid signal producers and centrally control an entire circuit portion [17]. Although beyond the scope of this work, the insights presented in this section as well as our easily extensible verification framework (which we will detail in Section 7) can serve as a foundation for such explorations and optimizations.

6 ENSURING SCALABILITY

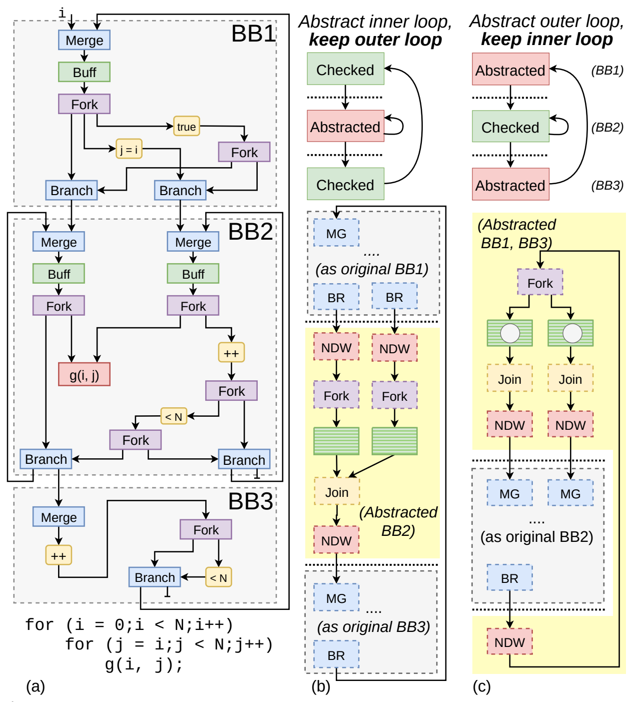
The fine-grain dataflow model can become very complex for circuits describing larger programs, and the reachable state space grows exponentially with the model size—as we will later see, this may have a detrimental effect on the model checking runtime. We here discuss our strategy to ensure the scalability of our approach.

Many of the states that the dataflow circuit exhibits are irrelevant for proving a particular property (e.g., many reachable behaviors inside one loop are invisible to and thus independent of the optimization of another loop). This is the intuition behind *compositional model checking* (see Section 3), which decomposes the model into parts that can be checked independently; removed parts are abstracted into an equivalent or slightly more general model.

We apply this insight in our work: we collapse a portion of the dataflow circuit into a single *supernode*, characterized with the properties of Section 3, where each property is derived or generalized from the properties of the encapsulated dataflow units. This supernode will accurately represent the behavior of the encapsulated circuit but at a significantly reduced model complexity, therefore enabling us to quickly model check other circuit portions. In the rest of this section, we detail our decomposition strategy.

6.1 Decoupling Circuit Regions

Dataflow circuits obtained from software programs group dataflow units into a CFG (see Section 3.1). This organization suggests intuitive ways to organize the circuit into regions whose properties can be verified in isolation: (1) The CFG can be decoupled into independent loop nests (i.e., strongly connected components of the CFG); as they execute sequentially, their properties are largely independent and can be verified separately. (2) Each loop nest can be decoupled based on nesting levels (e.g., innermost loop and outer loop); as the units of the same loop typically exhibit similar properties (e.g., have the same computing rate), it is sensible to group them in a single region. (3) Within each loop BB, larger straight computational paths can be declared as independent regions; as above, units in a long chain of operators typically have many behavioral similarities, so it is desirable to verify them together.



```
for (i = 0;i < N;i++)
    for (j = i;j < N;j++)
        g(i, j);
```

Figure 7: Ensuring scalability by compositional circuit verification. We decompose the circuit into regions whose properties can be verified independently of others; we abstract the complexity of other regions into simpler supernodes that have the same properties as the circuit section they encapsulate.

Figure 7a shows a dataflow circuit describing a nested loop, with units organized into BBs, decoupled into two regions (top of Figure 7b and c): One region will contain the inner loop (i.e., the units and channels of BB2, as well as the channels sending data from BB2 back to itself through the inner back edge). The other region will contain all nodes and channels of BB1 and BB3, corresponding to the outer loop. Depending on the complexity of function g in BB2, it may be decoupled into a separate region; we here assume that it remains in the same region as the remainder of BB2.

The regions obtained with this strategy receive inputs from at most one BB and send outputs to at most one BB (either equivalent to the sender, e.g., in case of a single computational path of a BB, or to a different BB, in case of a loop or loop nest). This implies that either all or no region inputs will be triggered (as the data is transferred through a straight datapath or following a single control flow edge); similarly, the region will either produce all of its outputs or none of them. Consider the region of BB2: as long as the loop repeats, all data remains within the region; once the condition to exit the loop is met, both data items exit the loop through their respective branch nodes. We will exploit this feature when abstracting regions into supernodes, as we will discuss next.

6.2 Abstracting Regions into Supernodes

The main challenge of decomposition is to ensure that the supernode representing the abstracted region maintains all relevant properties of the circuit portion it encapsulates. To this end, we model each supernode with the same (or a superset of) properties as those of dataflow units, as discussed in Section 4.1.

(1) *Firing*: To maintain producer-consumer relations among region inputs and outputs, we connect with a dataflow channel all data-dependent region live-ins and live-outs; this indicates that the input must arrive before the region can produce the dependent output. Consistently with standard dataflow circuit construction policies, whenever a live-in connects to multiple live-outs, we employ a fork to distribute it appropriately; if a live-out receives values from multiple live-ins, we synchronize them with a join.

(2) *Latency*: As a supernode can encapsulate complex behaviors, the exact latency may be difficult to determine (e.g., a loop with a statically unknown number of iterations). Analogously to the procedure with variable-latency units, we model the latency of our supernode with NDWs at the region outputs, thus capturing any possible latency values within the region.

(3) *II*: Similarly to the case above, we place NDWs at the region inputs to model any possible II. In more trivial cases, the II can be calculated based on the individual IIs of the encapsulated units (e.g., in a straight datapath, the unit with the highest II is the one that constrains the II of the entire region [8]).

(4) *Capacity*: The capacity of the region depends on the number of tokens that the region can hold. Unlike latency and II, assuming unbounded capacity is not feasible; instead, we can determine the maximal number of tokens that a region can hold: it is determined by the longest number of sequential stages (i.e., buffers or sequential computational units) from any of the region inputs to any of its outputs. We place FIFOs of this capacity on all input-output paths of the region (i.e., on every channel connecting a fork and a join), thus ensuring that all paths can sustain this maximal capacity.

Figures 7b and 7c show the supernodes representing the identified regions. In Figure 7b, the NDWs at the BB2 supernode input represent any stall and the NDW at its output any latency caused by the repetition of BB2 (i.e., the inner loop). The forks and joins maintain all data connections through the supernode (e.g., the paths from both branches of BB1 to the merge of BB3). The FIFOs are sized according to the maximal number of tokens BB2 can hold; in this example, the capacity of the buffers in BB2. The supernode of the outer loop (Figure 7c) is constructed following the same strategy.

It is interesting to note that all conditional execution within regions constructed as above can be abstracted away, as it never determines the triggering of the constructs of another region; all variability caused by the region's internal control flow is captured by its latency, II, and capacity constructs. In the example of Figure 7, the exit condition of BB2 does not impact the verification results of the outer region: the variable latency of the BB supernode captures any possible number of BB2 loop iterations and any possible time of triggering the outer region. Similarly, when verifying the inner region, the NDW constructs of the abstracted outer loop capture any possible reentry scenario into the inner loop, including the one in which a reentry never occurs (i.e., BB3 terminates the execution by sending tokens to the exit point). Other region organizations may require a placement of NDDs in places where control flow diverges (e.g., a condition of one region determines which of multiple succeeding regions should be triggered next). For simplicity, our strategy avoids the placement of NDD constructs; as we will later demonstrate, our approach is very effective in reducing the model checking times of larger circuits.

6.3 Checking Individual Regions

Once the regions and their supernode abstractions are defined, we can verify any property (e.g., the ones from Section 5) within each individual region while abstracting all other regions into supernodes; we repeat this procedure for every region. Of course, as the supernodes are more general than the exact circuit model, some properties that are provable in a complete model may no longer hold and the resulting circuit simplifications may be more conservative; this is a typical tradeoff of compositional model checking and we will explore it in Section 8. What is important to note is that our compositional approach will never introduce *false positives*, as it still accurately models all states that are reachable in the original circuit; thus, the resulting circuit transformations will always be correct and its final behavior unmodified.

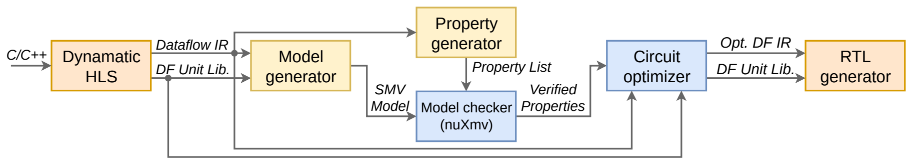
7 PUTTING IT ALL TOGETHER

The flow of our optimization framework is illustrated in Figure 8. The research artifact is publicly available [18].

The frontend to our flow is a dataflow circuit description that we obtain from Dynamatic [19], an open-source HLS compiler that translates C/C++ programs into dataflow circuits. Although we demonstrate our strategy on Dynamatic, as it is most recent and readily available, our work is in no way limited to this HLS strategy and can be directly applied to any dataflow approach [3–5].

Dynamatic produces an intermediate representation of the dataflow circuit in the form of a netlist that describes the interconnect of the employed dataflow units; its generic unit implementations are predefined in a unit library. Our *model generator* translates these units into the corresponding SMV [10] module while omitting all datapath computations and representing all data-dependent behaviors with nondeterministic constructs (Section 4). It translates the dataflow netlist directly into the corresponding SMV netlist to perform full circuit verification or decomposes larger circuits with multiple independent or nested loops (Section 6). Based on the dataflow netlist, the *property generator* automatically generates a list of properties that we are interested in checking. We currently aim to prove the absence of backpressure and the equivalence of trigger signals, as detailed in Sections 5.1 and 5.2, respectively, yet our generator can be easily extended to incorporate other property checks as well. Apart from equivalence checking within the entire BB, we explore the reduction of checking to channels with a single source or destination node (Section 5.2). We provide the complete SMV circuit description and the property list to the NuXmv model checker [20], a standard tool for analyzing synchronous finite-state and infinite-state transition systems. NuXmv verifies the specified properties and informs us of their validity. We utilize the BDD backend in NuXmv to perform reachability analysis. Our *circuit optimizer* uses this information to simplify the circuit prior to transforming it to its optimized RTL description.

The circuit simplifications that our current property checks imply require only signal modifications in the dataflow circuit netlist: (1) If the backpressure among a sender and a receiver is proven absent, we disconnect the ready signal between them and provide the



```
// instantiate fork; declare connection
VAR fork_1 : fork_1_3 (buff_1.valid0,
add_1.ready1, load_1.ready0, FIFO.ready0);
// checking fork has equivalent valids
INVARSPEC fork_1.valid0 <-> fork_1.valid1;
// checking fork never stalled by the adder
INVARSPEC fork_1.valid0 -> add_1.ready1;
```

Verifiying properties for the circuit in the Figure 1

Figure 8: Our optimization framework based on model checking. The input to our framework is a dataflow circuit obtained from C/C++ code; we automatically create its symbolic model and a set of properties to verify using model checking (see the code snippet on the right). The output of our flow is a simplified and smaller dataflow circuit, equivalent in performance and functionality to the original one.

sender with a constant value of 1. (2) When multiple valid signals are proven equivalent, we connect one of the equivalent signals to all receivers and disconnect their original senders. The dataflow unit library remains intact by our framework; it is up to the logic optimizer of a standard FPGA flow [21] to use our netlist transformations to simplify the dataflow units during logic synthesis (e.g., by optimizing out the disconnected dataflow logic). Enhancing our framework with more complex properties may require particular unit modifications as well; in such cases, one could simply extend the input unit library with multiple customized unit implementations and allow the circuit optimizer to choose the most appropriate one based on the verification results.

8 EVALUATION

In this section, we describe the effectiveness of our strategy in eliminating redundant dynamism and reducing the resource requirements of dataflow circuits.

8.1 Benchmarks and Methodology

Our benchmarks are a collection of kernels that others have employed to evaluate dynamic scheduling in HLS [4, 5, 22]; some have been devised specifically to show the benefits of dynamic scheduling, whereas others originate from standard HLS suites [23, 24]: (1) *histogram* and *matrix power* are typical examples where dynamic HLS outperforms static thanks to a load-store queue [15] that resolves memory dependencies at runtime, (2) *if loop mul* and *if loop add* contain irregular and unpredictable control flow that prevents static HLS from achieving high throughput; in contrast, dataflow circuits adapt their execution dynamically to particular control flow outcomes. (3) *fir, iir, sumi3mem, matvec, 2mm, 3mm* are regular kernels that can be equally pipelined by static and dynamic HLS; despite equivalent performance, general dataflow circuits are significantly more resource expensive than the corresponding static designs (especially *2mm* and *3mm*, which use an LSQ, as we will later discuss). Our benchmarks are publicly available [18].

We synthesize the benchmarks with the standard Dynamatic flow [19] and optimize with our framework described in Section 7. We perform functional verification of all designs in ModelSim [25] and use it to obtain the clock cycle count; we present our area and timing results post-place-and-route with Vivado [21], targeting a Kintex-7 Xilinx FPGA. We measure the model checking runtime of nuXmv on an AMD Ryzen 7 PRO 5850U CPU at 1.90 GHz.

8.2 Comparison with Generic Dataflow Circuits

In this section, we compare our circuits with the generic and complete dataflow designs obtained by Dynamatic. We are interested in evaluating the resource reductions thanks to our ability to remove redundant dynamism, as well as ensuring that our methodology maintained the correctness and performance of the original circuits.

Table 1 compares the resources and performance of the original dataflow designs (*No Opt.*), designs we obtain by model checking the entire circuit (*Full Model*), and designs obtained through compositional and reduced model checking (*Reduced Model*). We discuss the *Reduced Model* in Section 8.4. As expected, the clock cycle count of our designs is identical to that of Dynamatic, as our transformations guarantee that the circuit's behavior and timing remain intact. Yet, our designs are significantly smaller, as they retain the generic handshake communication only where it is beneficial. The same reduction typically also causes a critical path (CP) reduction, as many combinational paths are removed or shortened; this leads to a reduction in total execution time. All our designs require fewer resources than the *No Opt.* designs; our savings are, naturally, larger in benchmarks where dynamism is unneeded (e.g., *fir*, *matvec*). When latency sensitivity is beneficial (e.g., *if loop add*), we maintain some of it for performance benefits, but reduce its total cost.

The cost of the memory interface is notable in benchmarks that require an LSQ (see four benchmarks in Table 1, where we report the LSQ resources next to those of the respective kernel). This expensiveness has been observed by others [15, 26] and is orthogonal to our approach. Although this expensive LSQ makes our total resource savings small (e.g., negligible total LUT reduction in *2mm*), it is important to note the computational kernel savings follow the same trends as in all other benchmarks (e.g., 18% for *2mm*). Our target here was to apply model checking to simplify the computational kernels, which we have consistently and successfully achieved.

8.3 Effectiveness of Simplification Properties

In this section, we compare the effectiveness of the two properties that we employ to simplify our circuits (see Section 5). In Figure 9, we plot the computational kernel resources obtained by trigger equivalence (orange) and backpressure (blue) in isolation, as well as their combination (i.e., the green bars are the *Full Model* from Table 1), relative to the Dynamatic resources. Our results show that trigger equivalence is less effective in reducing the area than backpressure absence. This is likely due to the fact that some of the same optimization opportunities are already identified by the logic synthesizer [21] in all design points; yet, the fact that we achieve further reductions from our baseline indicates that formal verification is more powerful. Naturally, verifying both properties simultaneously achieves the best results; including more properties in our framework (see Section 5.3) could further increase the benefits of using formal verification for circuit optimization.

Benchmark	Method	LUTs (Kernel + LSQ)	LUT Red.	FFs (Kernel + LSQ)	FF Red.	DSPs	Cycles	CP (ns)	Exec Time (us)†	Exec Red.	Check Time(s)	Checks§	State Vars§
fir	No Opt.	513	-	508	-	3	1017	3.7	3.7	-	-	0	109
	Full	281	-45%	267	-47%	3	1017	3.4	3.5	-5%	134	1026	109
	Reduced	286	-44%	268	-47%	3	1017	3.5	3.5	-5%	15	109	109
iir	No Opt.	885	-	1104	-	6	5008	3.5	17.7	-	-	0	132
	Full	438	-51%	638	-42%	6	5008	3.1	15.4	-13%	154	2132	132
	Reduced	505	-43%	674	-39%	6	5008	3.4	16.9	-5%	10	168	132
if loop mul	No Opt.	910	-	1057	-	5	3522	5.4	19.2	-	-	0	154
	Full	757	-17%	892	-16%	5	3522	4.8	16.9	-12%	1248	1283	154
	Reduced	758	-17%	893	-16%	5	3522	4.9	17.3	-10%	186	120	154
if loop add	No Opt.	1042	-	1248	-	4	5522	5.4	30	-	-	0	158
	Full	855	-18%	1084	-13%	4	5522	4.5	24.9	-17%	1454	1284	158
	Reduced	824	-21%	1084	-13%	4	5522	4.7	26	-13%	327	120	158
sumi3 mem	No Opt.	537	-	462	-	6	1021	4.1	4.1	-	-	0	125
	Full	307	-43%	266	-42%	6	1021	3.4	3.5	-15%	313	1022	125
	Reduced	311	-42%	267	-42%	6	1021	3.5	3.6	-12%	36	106	125
histogram	No Opt.	820 + 10511	-	996 + 2485	-	2	1167	6.5	7.6	-	-	0	106
	Full	668 + 10513	-19% (-1%*)	810 + 2485	-19% (-5%*)	2	1167	6.5	7.6	0%	42	949	106
	Reduced	677 + 10662	-17% (0%*)	808 + 2485	-19% (-5%*)	2	1167	6.3	7.3	-4%	6	125	106
matvec	No Opt.	781	-	587	-	3	946	4.7	4.5	-	-	0	166
	Full	380	-51%	278	-53%	3	946	3.6	3.4	-24%	3779	2034	166
	Reduced	382	-51%	282	-52%	3	946	3.6	3.4	-24%	603	211	166
matrix power	No Opt.	934 + 6640	-	914 + 1974	-	5	1549	6.5	10.1	-	-	0	153
	Full	820 + 6650	-12% (-1%*)	866 + 1975	-5% (-2%*)	5	1549	6.2	9.6	-5%	2375	2211	153
	Reduced	835 + 6636	-11% (-1%*)	866 + 1977	-5% (-2%*)	5	1549	6.1	9.4	-7%	5	186	93/67
2mm	No Opt.	3375 + 33526	-	2351 + 5782	-	12	5608	7.7	43.2	-	-	0	542
	Full‡	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	11240	542
	Reduced	2764 + 34249	-18% (0%*)	1807 + 5791	-23% (-7%*)	12	5608	7.9	44.1	2%	4921	725	275/204
3mm	No Opt.	3549 + 46492	-	1464 + 8432	-	9	8400	6.9	57.7	-	-	0	583
	Full‡	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	11149	583
	Reduced	3288 + 46005	-7% (-1%*)	1276 + 8431	-13% (-2%*)	9	8400	6.8	56.9	-1%	442	845	182/185/170

** total red. (with LSQ included); † Exec Time = CP × Cycles; ‡ Timed out after 10h; § Model checking scales linearly with Checks and exponentially with State Vars*

Table 1: Resources and timing of dataflow circuits optimized with our strategy (*Full* and *Reduced*), compared with standard dataflow circuits (*No Opt.*). The *Full* designs are obtained by model checking the entire circuit. In the *Reduced* designs, we decompose benchmarks with multiple loops and reduce the number of equivalence checks. In all benchmarks, our strategy significantly reduces the resource requirements of the computational kernel without a performance penalty. Four benchmarks use an LSQ at the memory interface, which makes up a significant portion of the circuit's resources; this overhead is orthogonal to our strategy, which optimizes handshake logic in the computational kernel. The model checking time of the *Full* designs is extensive (in the final two benchmarks, the checker times out after 10 hrs without concluding a single property). The *Reduced* model successfully reduces the checking time, without significantly degrading the optimization quality.

8.4 Scalability Analysis

An exhaustive optimization is not feasible for more complex circuits (for the final two benchmarks, our model checker times out after 10 hours). We here evaluate techniques to make our approach scalable.

The final row of each benchmark in Table 1 (i.e., *Reduced Model*) shows the area and performance values of the circuits optimized with our approach while applying decomposition (Section 6) on benchmarks with multiple loops and reducing the number of equivalence checks to signals of the same source or destination in all benchmarks (Section 5.2); the resource results are also plotted in Figure 9 (rightmost bars). Table 1 reports the total number of checks (*Checks*) and number of state variables (*State Vars*, reported for every decomposed region); model checking complexity scales linearly with *Checks* and exponentially with *State Vars*.

Our strategy significantly improved the checking time and made it feasible for all explored benchmarks without notably reducing the design quality. The results of the final benchmarks indicate that our decomposition approach is effective in decoupling independent regions and our supernodes accurately describe the abstracted circuit portions, without over-generalizing their behaviors.

8.5 Comparison with Static HLS Circuits

Dynamically scheduled circuits typically require significantly more resources than their static counterparts [5, 7]; we here explore the effectiveness of our strategy in reducing this resource gap.

Figure 10 visualizes the datapath resources (LUTs, FFs) and total execution time (product of clock cycles and CP) of the complete and optimized dataflow solutions, normalized to the corresponding Vivado HLS designs [27] (all in red dashed line at value of 1). In all benchmarks, our strategy brings the dataflow designs closer in resources to static designs; the reduction is most notable in benchmarks where dynamic scheduling is not useful and our strategy removes dataflow logic more aggressively. In most cases, the performance improves as well, as our designs typically have a lower critical path; this increases the performance benefits of dynamic scheduling in irregular benchmarks (e.g., *if loop mul*) and reduces the performance gap between static and dynamic designs otherwise. Despite the significant area savings thanks to our strategy, dataflow circuits are still more expensive than their static counterparts, mainly due to the following: (1) Dataflow circuits require expensive LSQs, as discussed before; (2) Our current framework uses Dynamatic as its frontend; the intermediate representation of the circuits produced by this tool is known to be more conservative and complex than that of Vivado HLS [28], which gives the static circuits an immediate advantage. These effects are orthogonal to our work and could be improved by advanced memory optimizations and compiler analyses. We here made a significant advancement in making dynamic scheduling more competitive with static HLS in both area and execution time and demonstrated the relevance of applying formal methods in hardware design and optimization.

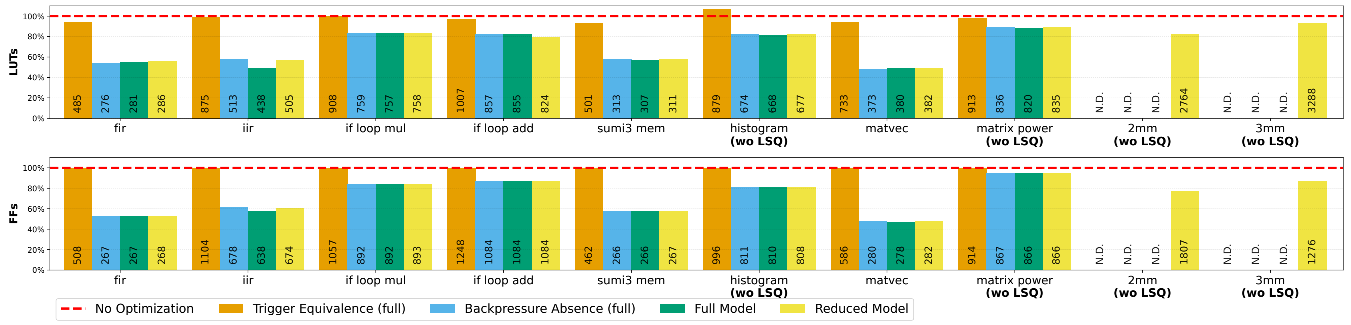


Figure 9: Resource of optimized dataflow circuits produced by Dynamatic using our strategy. All designs are normalized with respect to the design generated by Dynamatic without modification. To demonstrate the effectiveness of our approach to simplify the datapath, we have excluded from the synthesis reports the hierarchy that Vivado identifies as part of LSQ.

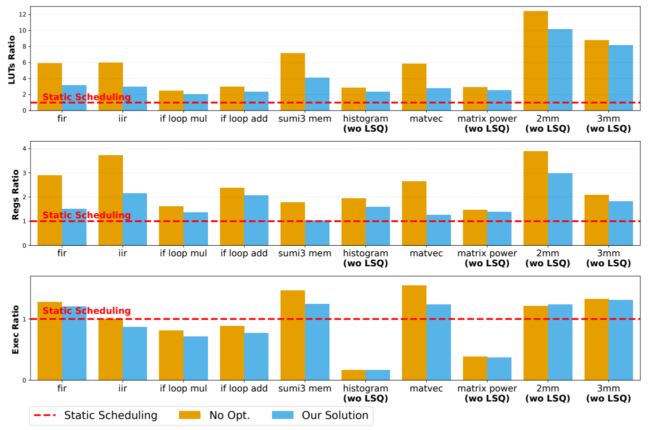


Figure 10: Resources (LUTs, FFs) and execution time (product CP × clock cycles) of the original (*No Opt*) and optimized (*Our Solution*) dataflow circuits, normalized to the corresponding static HLS designs. Our work makes all dataflow circuits more competitive in both area and execution time to static HLS. In *iir*, *if loop mul*, *if loop add*, *histogram*, and *matrix power*, our solutions are Pareto-optimal (best performance overall at smaller area than prior dataflow work).

9 RELATED WORK

Different latency-insensitive protocols [1, 29, 30] have been explored for constructing synchronous and asynchronous dataflow circuits and many efforts investigated their generation from high-level programs [3, 4, 31, 32]. The expensiveness of dataflow computation has often been discussed in the HLS context: recent works develop buffering schemes for frequency regulation [8, 22], share functional units for area savings [7], and reduce the complexity of the memory interface [26]. We profit from all of these techniques in this work to obtain state-of-the-art dataflow circuits from high-level code. Yet, the resulting circuits still implement generic latency-insensitivity and suffer from its overheads; thus, they can be directly improved by our approach. Cheng et al. [5] observed this problem and replaced sections of HLS-produced dataflow circuits with the corresponding static designs; this approach is effective at a coarse grain, but it fails to identify fine-grain logic-level optimization opportunities in the dynamically scheduled circuit regions. Our work seizes this opportunity: we construct formal proofs to systematically restrict the generality of the fine-grain dataflow logic and simplify the circuits accordingly.

Recent efforts aim to formally verify the HLS process [33, 34]. Others employ formal methods to optimize HLS-produced circuits: Cheng et al. use an SMT-based solver to improve the memory arbitration in HLS [35] and employ Petri nets to determine pipeline initiation intervals [36]. Multiple works employ various forms of Petri nets to model and optimize the performance of dataflow pipelines [8, 37, 38], whereas Geilen et al. employ model checking for buffering coarse-grain dataflow graphs [39]. Our work complements these efforts: we aim to prove particular behavioral properties of HLS-produced dataflow circuits and use them to improve their hardware implementation. State-exploration techniques such as sequential logic synthesis [40] could be used to prove some of the properties we explore in this work. Yet, they typically target exact gate-level descriptions; model checking allows us to abstract and generalize circuit behaviors (e.g., by employing nondeterministic constructs when critical data or control decisions are undeterminable), thus allowing us to reason about a wider variety of behavioral properties in HLS-produced dataflow circuits.

10 CONCLUSION

Despite an increased interest in generating dataflow circuits from high-level code, this HLS paradigm is still largely unpopular due to its resource expensiveness. In certain cases, this overhead is justified, as it is the reason for the superiority of dataflow circuits over their statically scheduled counterparts; however, there are many cases where their dynamism is just an unnecessary overhead. In this work, we propose an optimization framework that discovers and proves particular behavioral scenarios in HLS-produced dataflow circuits via model checking. Ascertaining that particular situations never occur allows us to restrict the generality of dataflow logic only to relevant and reachable behaviors and, thus, to reduce its complexity. In programs that require dynamic scheduling, our strategy maintains all performance benefits at a significantly reduced area overhead; otherwise, it makes dataflow circuits resource- and timing competitive with those produced by static HLS. This new avenue of using formal verification to optimize HLS-produced circuits is a promising step in making dataflow circuits obtained from software programs practical and widely usable.

REFERENCES

[1] 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.

[2] L. P. Carloni and A. L. Sangiovanni-Vincentelli, "Performance analysis and optimization of latency insensitive systems," in *Proceedings of the 37th Design Automation Conference*, Los Angeles, CA, Jun. 2000, pp. 361–67.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] C. Kern and M. R. Greenstreet, "Formal verification in hardware design: a survey," *ACM Transactions on Design Automation of Electronic Systems*, vol. 4, no. 2, pp. 123–93, Apr. 1999.

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

[11] E. Clarke, D. Long, and K. McMillan, "Compositional model checking," in *Proceedings of the Fourth Annual Symposium on Logic in computer science*, Pacific Grove, California, USA, Jun. 1989, pp. 353–362.

[12] S. Berezin, S. Campos, and E. M. Clarke, "Compositional reasoning in model checking," in *Compositionality: The Significant Difference*, May 1998, pp. 81–102.

[13] A. K. Verma, P. Brisk, and P. Ienne, "Variable latency speculative addition: A new paradigm for arithmetic circuit design," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, Munich, Mar. 2008, pp. 1250–55.

[14] The LLVM Compiler Infrastructure, 2018. [Online]. Available: http://www.llvm.org

[15] 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. 125:1–125:19, Sep. 2017.

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

[17] 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.

[18] J. Xu, *Research Artifact for ISFPGA'23: Eliminating Excessive Dynamism of Dataflow Circuits Using Model Checking*, Dec. 2022. [Online]. Available: https://doi.org/10.5281/zenodo.7458595

[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] 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.

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

[22] 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, pp. 375–83.

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

[24] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks, "MachSuite: Benchmarks for accelerator design and customized architectures," in *Proceedings of the IEEE International Symposium on Workload Characterization*, Raleigh, NC, October 2014.

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

[26] 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 IEEE International Conference on Field Programmable Technology*, Tianjin, Dec. 2019, pp. 197–205.

[27] *Vivado Design Suite User Guide: High-Level Synthesis*, Xilinx Inc., 2018. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug902-vivado-high-level-synthesis.pdf

[28] 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, pp. 253–61.

[29] 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.

[30] 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.

[31] R. Townsend, M. A. Kim, and S. A. Edwards, "From functional programs to pipelined dataflow circuits," in *Proceedings of the 26th International Conference on Compiler Construction*, Austin, TX, Feb. 2017, pp. 76–86.

[32] J. Sparsø, "Current trends in high-level synthesis of asynchronous circuits," in *Proceedings of the 16th IEEE International Conference on Electronics, Circuits, and Systems*, Yasmine Hammamet, Dec. 2009, pp. 347–50.

[33] Y. Herklotz, Z. Du, N. Ramanathan, and J. Wickerson, "An empirical study of the reliability of high-level synthesis tools," in *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, May 2021, pp. 219–23.

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

[35] J. Cheng, S. T. Fleming, Y. T. Chen, J. Anderson, J. Wickerson, and G. A. Constantinides, "Efficient memory arbitration in high-level synthesis from multi-threaded code," *IEEE Transactions on Computers*, vol. 71, no. 4, pp. 933–46, Apr. 2022, conference Name: IEEE Transactions on Computers.

[36] J. Cheng, J. Wickerson, and G. A. Constantinides, "Probabilistic scheduling in high-level synthesis," in *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines*, May 2021, pp. 195–203.

[37] M. Najibi and P. A. Beerel, "Slack matching mode-based asynchronous circuits for average-case performance," in *Proceedings of the 32nd International Conference on Computer-Aided Design*, San Jose, CA, Nov. 2013, pp. 219–25.

[38] 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.

[39] M. Geilen, T. Basten, and S. Stuijk, "Minimising buffer requirements of synchronous dataflow graphs with model checking," in *Proceedings. 42nd Design Automation Conference, 2005.*, Jun. 2005, pp. 819–824.

[40] R. Brayton and A. Mishchenko, "Scalably-verifiable sequential synthesis," *ERl Technical Report*, 2007.