ElasticMiter: Formally Verified Dataflow Circuit Rewrites

Ayatallah Elakhras[*]
EPFL
Lausanne, Switzerland
ayatallah.elakhras@epfl.ch

Jiahui Xu[*]
ETH Zurich
Zurich, Switzerland
jxu@ethz.ch

Martin Erhart
ETH Zurich
Zurich, Switzerland
merhart@student.ethz.ch

Paolo Ienne
EPFL
Lausanne, Switzerland
paolo.ienne@epfl.ch

Lana Josipović
ETH Zurich
Zurich, Switzerland
ljosipovic@ethz.ch

Abstract

Dataflow circuits have been studied for decades as a way to implement both asynchronous and synchronous designs, and, more recently, have attracted attention as the target of *high-level synthesis* (HLS) compilers. Yet, little is known about mechanisms to systematically transform and optimize the datapaths of the obtained circuits into functionally equivalent but simpler ones. The main challenge is that of equivalence verification: The latency-insensitive nature of dataflow circuits is incompatible with the standard notion of sequential equivalence, which prevents the direct usage of standard sequential equivalence verification strategies and hinders the development of formally verified dataflow circuit transformations in HLS. In this paper, we devise a generic framework for verifying the equivalence of latency-insensitive circuits. To showcase the practical usefulness of our verification framework, we develop a graph rewriting system that systematically transforms dataflow circuits into simpler ones. We employ our framework to verify our graph rewriting patterns and thus prove that the obtained circuits are equivalent to the original ones. Our work is the first to formally verify dataflow circuit transformations and is a foundation for building formally verified dataflow HLS compilers.

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

Keywords: Dataflow circuits, high-level synthesis, formal verification, model checking

ACM Reference Format:
Ayatallah Elakhras, Jiahui Xu, Martin Erhart, Paolo Ienne, and Lana Josipović. 2025. ElasticMiter: Formally Verified Dataflow Circuit Rewrites. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3676641.3715993

[*]Both authors contributed equally to this work.

1 Introduction

The rise of dataflow circuits in HLS. *High-level synthesis* (HLS) compilers convert high-level programs (e.g., C/C++) into circuits and offer application designers an efficient way to program FPGAs [2, 9, 56]. Recent HLS approaches focus on producing dataflow circuits [21, 22, 31], built out of fine-grained operators that communicate with bidirectional handshake signals; the time when data is exchanged among these operators is not predefined by the HLS compiler, but determined at runtime based on particular data and control outcomes. This scheduling flexibility, paired with a variety of performance-enhancing features such as speculation [32] and multithreaded execution [23], offers significant performance advantages over standard HLS compilers in irregular and control-dominated applications, and motivates the further development and enhancements of this HLS approach.

How to verify dataflow circuits? As dataflow-oriented HLS strategies evolve and get more complex, there is an increasing need to verify the produced dataflow circuits and circuit transformations. Although some formal verification strategies for standard HLS compilers apply here as well [48], dataflow circuits exhibit a unique property: due to their latency-insensitivity, two sequentially nonequivalent circuits that produce the same results (possibly, at different times) are considered functionally equivalent and can thus be used interchangeably. Yet, standard sequential equivalence [20, 51, 55] is too restrictive to capture this property—it requires two equivalent circuits to produce the same outputs on every clock cycle and thus cannot capture the latency-insensitivity of dataflow computation.

Consider the example in Figure 1a. Circuit A is a standard sequential circuit that duplicates the value received from its input to the two outputs with a 2-cycle delay, as determined by the *flip-flops* (FFs) on the different paths. Circuits A and B produce the same values on every clock cycle and are, therefore, sequentially equivalent; circuit C is not equivalent to them as the value of the input arrives to the

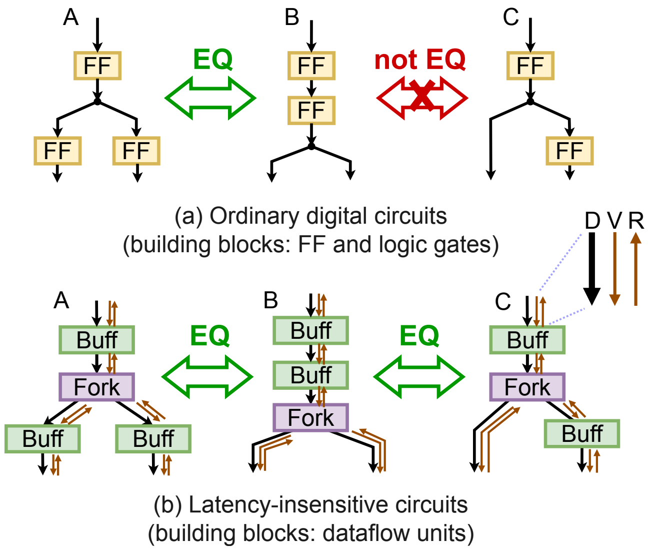

A
B
C
FF
EQ
not EQ
(a) Ordinary digital circuits
(building blocks: FF and logic gates)
D V R
Buff
Fork
(b) Latency-insensitive circuits
(building blocks: dataflow units)

Figure 1. Sequential equivalence vs. latency-insensitive equivalence. Two ordinary circuits are sequentially equivalent if, on every clock cycle, they produce identical data; this is the case for circuits A and B in Figure 1a, but not for circuit C. In contrast, dataflow circuits are considered equivalent if they produce and consume the same sequence of valid tokens, regardless of when the token exchanges take place; this is the case for all three circuits in Figure 1b. This work devises a framework that verifies *latency-insensitive equivalence* and uses it for dataflow circuit simplification.

left output one cycle earlier than in the first two circuits. A conceptually similar dataflow circuit is shown in Figure 1b; the main difference is that all units, including the sequential buffers, communicate with handshake signals to exchange data *tokens*. For a valid token to be exchanged among two units, both handshake signals must be set; data that is not accompanied by a handshake exchange is invalid and thus irrelevant. The three dataflow circuits in Figure 1b are not sequentially equivalent: some of their output wires may have different values at a given point in time. Yet, they all produce and consume the same sequence of valid tokens—replacing one circuit with another only changes the times when the tokens are produced and consumed, and not the token values themselves [10]. Capturing this is beyond the standard definition of sequential equivalence.

In analogy to software code, sequential equivalence is conceptually similar to checking whether two assembly codes always produce the same values in the same number of clock cycles. We aim for a verification strategy that resembles standard software verification: it is the data values that matter and not the exact time when they are computed. We need to rethink the concept of equivalence for dataflow circuits accordingly and devise a general strategy to verify it.

ElasticMiter: an equivalence verifier for dataflow circuits. In this paper, we devise a practical definition of equivalence that captures the latency-insensitivity of dataflow circuits; instead of cycle-by-cycle values, it considers whether two dataflow circuits eventually produce the same sequence of valid tokens. We develop ElasticMiter, an automated model-checking-based approach for verifying dataflow circuit equivalence. ElasticMiter allows us to verify equivalence in general as well as under certain behavioral conditions—a feature that we will exploit to reason about HLS-produced dataflow circuits whose behaviors are constrained by the properties of the originating software code.

A formally verified graph-rewriting system for dataflow optimization. Dataflow circuits tend to suffer from area overheads [33, 60]—it is favorable to replace them with simpler equivalent circuits whenever possible. To this end, we propose a graph rewriting system composed of a series of rewrite patterns to systematically simplify expensive dataflow logic. Our patterns are small and localized—thus, we formally verify them with ElasticMiter in acceptable runtime. The resulting circuits are smaller, faster, and provably equivalent to circuits produced by a state-of-the-art HLS compiler. More importantly, this illustrates the ability of ElasticMiter to perform systematic and practical formal verification of HLS-produced dataflow circuits.

This paper has the following contributions: (1) We build upon the theory of latency-insensitive design [10] to devise a formal definition of latency-insensitive equivalence compatible with model checking. (2) We devise a formal framework for proving that two dataflow circuits are latency-insensitive equivalent regardless of their surroundings. (3) We devise a set of rewrite rules that reduce the area and latency of HLS-produced dataflow circuits and use our framework to prove their correctness. (4) We implement a compiler optimization pass based on the rewrite rules in Dynamatic, an open-source MLIR-based HLS compiler that converts C to dataflow circuits; our solution has achieved 24% mean reduction in latency and 14% reduction in FPGA FF resources.

2 The Need for Dataflow Circuit Optimization and Verification

This section describes the units that form dataflow circuits. We illustrate the need to optimize dataflow steering logic in HLS-produced dataflow circuits. We then outline the formal verification techniques that this paper relies on.

2.1 Background on Dataflow Circuits

Dataflow circuits are networks of units that communicate with latency-insensitive channels [10, 19, 21, 31] containing handshake signals—*valid* and *ready*. A handshake protocol regulates the computation: data propagates through the channel when valid and ready are both set; otherwise, the data is stalled. The units progress the computation and there is

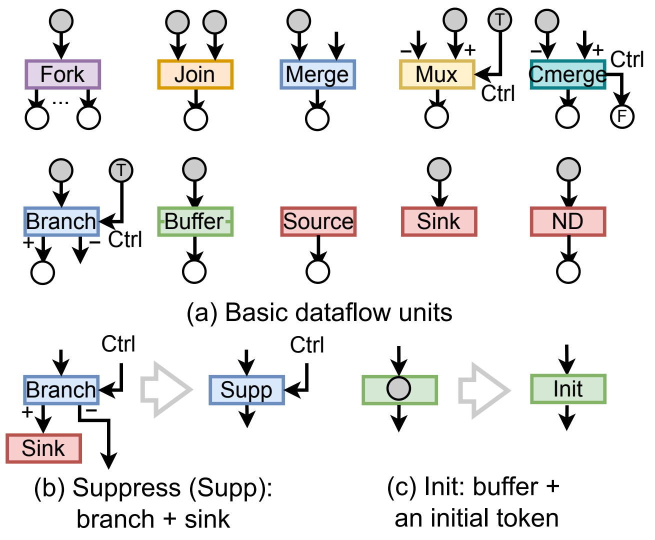


Figure 2. Our dataflow units. Gray tokens are consumed by the units to produce the white tokens.

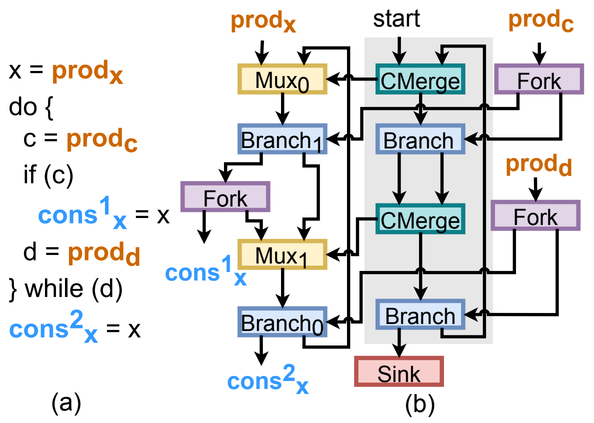


Figure 3. Dataflow circuit coupling data propagation with control flow. The dataflow circuit in (b) implements a part of the code (a) that delivers the variable x from its definition ($prod_x$) to its two uses ($cons_x^1$ and $cons_x^2$). Mux_0 and $Branch_0$ mimic the loop, and Mux_1 and $Branch_1$ mimic the if-then-else. The units with gray backgrounds mimic a sequential program counter and steer data following the program's control flow—this leads to overly complicated and slow circuits.

no centralized scheduler. Many research works discuss the generation of dataflow circuits [6, 19, 21, 31]; we focus on HLS-produced dataflow circuits generated from C code [31].

Figure 2 shows our dataflow units. (1) **Fork** has 1 input and multiple outputs; it replicates the input token and sends it to all successors. (2) **Join** has multiple inputs and 1 output; it synchronizes the tokens of all predecessors before generating the output token. Joins reside in arithmetic and logic units to ensure they operate on valid data. (3) **Merge** has 2 inputs and 1 output; it propagates a token received at any input to its output; it prioritizes one of the inputs if multiple tokens are simultaneously present. (4) **Mux** has 2 data inputs ("+" and "−"), a control input (*Ctrl*) and 1 data output; it sends the "+"-input token to the output if the control token has value *true*, and the "−"-input otherwise; the input that is not selected waits. (5) **Control Merge** (CMerge) is a Merge with a condition output, indicating from which input it propagates a token. (6) **Branch** consumes a condition token and propagates the data input token to the "+" output if the condition is *true* and to the "−" output if the condition is *false*. (7) **Buffer** is the only unit that stores tokens; it breaks combinational loops, eliminates throughput bottlenecks, and can be inserted between any two units without altering the functionality [7, 34, 50]. (8) **Source** has no inputs; it constantly delivers data tokens to its successor. (9) **Sink** has no outputs; it is always ready to take tokens. (10) **Nondeterministic wire** (ND) is an artificial unit that does not appear in the implemented circuits; ElasticMiter uses it to model arbitrary stalls [44, 60], as we will see in Section 4.

We also introduce the following units in Figure 2b and Figure 2c: A Branch with a Sink after the "+"-output is a **Suppress** (Supp). A Buffer with an initial token is an **Init**.

2.2 Dataflow Circuits from Imperative Code Can Miss Optimization Opportunities

Dataflow circuits are generated from imperative code by translating individual operations into dataflow units (e.g., arithmetic units with handshake communication), and adding the dataflow units from Figure 2 to appropriately steer tokens between the operators without altering the data values. Muxes and Branches *conditionally* transfer data between the computational units [30, 31, 41]. This is analogous to control flow in programs: the result of one operation must be fed to another operation only if some condition is true.

Typically, dynamically-scheduled HLS strategies [6, 31] couple data propagation with the control flow. For example, Figure 3b shows a circuit that implements the code snippet in Figure 3a following these HLS strategies (some circuit details, e.g., the Buffers, are omitted for clarity). It propagates the value of variable x from its producer ($prod_x$ executes *once* in the beginning) to two consumers ($cons_x^1$ executes *conditionally* inside a loop, and $cons_x^2$ executes *once* in the end). The propagation starts with Mux_0 that, together with $Branch_0$, implements the do-while loop that is controlled by the condition produced by $prod_d$. Then, it continues with $Branch_1$ that, together with Mux_1, implements the if condition inside the loop, based on the $prod_c$ condition. The select inputs of Muxes are calculated by the network of CMerges (see Section 2.1), highlighted in gray. Essentially, this network mimics a sequential program counter by propagating a dataless token that is injected once upon the start of the execution and continues through the circuit according to the control-flow decisions. The effect of this circuit is that

$cons_{\mathrm{x}}^{1}$ receives a token as long as $prod_{\mathrm{c}}$ and $prod_{\mathrm{d}}$ are producing *true* tokens and $cons_{\mathrm{x}}^{2}$ receives a token only once $prod_{\mathrm{d}}$ produces a *false* token.

Although intuitive and correct, the circuit in Figure 3b is overly complicated and has a limited performance, as we show later: It is essential for $cons_{\mathrm{x}}^{1}$ to be sensitive to both the loop and if conditions; however, it is completely unnecessary and inefficient for $cons_{\mathrm{x}}^{2}$ to wait for the loop to complete its execution—it would be correct to propagate its data directly from $prod_{\mathrm{x}}$, irrespective of any control flow. Nevertheless, it is challenging to decide when to propagate data directly versus when to couple it with the control flow. Some efforts apply control dependence analysis on the high-level input to match the conditions of production and consumption to implement the minimal necessary control flow [15, 22, 28]. Yet, such analysis is nontrivial and is hard to formally verify.

Section 6 presents a set of formally verified circuit rewrites that systematically transform inefficient circuits such as the one in Figure 3b into equivalent and simpler circuits. Yet, we first need to define our notion of equivalence for dataflow circuits and devise a framework to verify them. We tackle these questions first in Section 3, Section 4, and Section 5.

2.3 Formal Verification Using Model Checking

Model checking [4, 17, 53] is an automatic approach for formal verification of temporal properties in finite state machines (FSM). A *model checker* [5, 11] analyzes the *state space* (the set of possible states that the FSM can reach) of the FSM; it provides a formal proof if the property holds, and shows a counterexample otherwise.

ElasticMiter exploits model checking to verify circuit equivalence. We consider properties specified using two *computation tree logic* (CTL) operators—a common extension of Boolean logic to include the notion of time [36, 46]: **AG** (*always globally*) and **AF** (*always finally*). For a proposition p (e.g., p = "signals s_a and s_b are both 1"), a statement **AG** p asserts that p must hold at the initial state, and always hold after any possible state transition; **AF** p asserts that the p must hold at some point in the future; if there exists a possible future in which p does not hold forever, then **AF** p is not honored. As we will see in Section 3, ElasticMiter formulates the properties using these operators (e.g., it checks if some properties always hold, for which **AG** is a good fit).

3 Defining Latency-Insensitive Equivalence

This section presents our definition of the *latency-insensitive equivalence* of two dataflow circuits, building upon the theory of latency-insensitive design [10]. Under the latency-insensitive protocol, we define the behaviors of the circuit in terms of the presence of valid tokens. Identifying valid tokens is protocol-specific (e.g., in SELF [19], a channel is considered to be transmitting a valid token in a given state only when both the valid and ready signals are set.).

Definition 1 (Token sequence)**.**
The token sequence of a channel is an ordered list of valid tokens sent over the channel.

Definition 2 (Equivalent token sequences)**.**
Two token sequences are equivalent if (1) their lengths are equal, and (2) the token values are pairwise equal.

We assume two given *circuits under verification* have matching input and output channels (we call them *elastic primary input channels* (EPIs) and *elastic primary output channels* (EPOs), to distinguish with the terminology of *primary inputs and outputs* (PIs and POs) used for ordinary digital circuits); two circuits with different interfaces are trivially not equivalent. For two circuits having n EPIs ($EPI_0, \ldots, EPI_{n-1}$) and m EPOs ($EPO_0, \ldots, EPO_{m-1}$) each, we define *latency-insensitive equivalence* as follows:

Definition 3 (Latency-insensitive equivalence)**.**
Two dataflow circuits with matching interfaces are *latency-insensitive equivalent* if, for any identical finite token sequences given at the EPIs, they eventually consume and produce identical token sequences at their EPIs and EPOs, respectively.

Verifying that two circuits satisfy Definition 3 has two challenges: *accounting for all possible scenarios where two circuits receive identical sequences* and *checking that the produced sequences are identical.* The following section presents a framework that overcomes these challenges.

4 ElasticMiter: A General Framework for Dataflow Circuit Equivalence Checking

This section presents ElasticMiter: a framework for verifying Definition 3 by embedding the two circuits to be compared within a larger circuit of a general form (the ElasticMiter fabric) and then proving the properties of this circuit.

4.1 The ElasticMiter Fabric

Figure 4 shows an example of the ElasticMiter fabric to prove the equivalence of two circuits. We refer to these as *left-hand side* (LHS) ❺ and *right-hand side* (RHS) ❻. In this example, LHS and RHS are trivially equivalent—both send the token received from EPI_0 to one of the outputs depending on the token received from EPI_1. A similar ElasticMiter fabric can be automatically constructed for any two dataflow circuits with matching interfaces. An ElasticMiter fabric has the following parts: (1) sequence generator, (2) input decoupler, (3) LHS and RHS contexts, (4) LHS and RHS, (5) output decoupler, and (6) sequence comparator, which we elaborate on next.

Sequence generator. The *sequence generator* (i.e., ❶ in Figure 4) ensures that LHS and RHS always receive identical token sequences. For each EPI, a Source (see Figure 2) sends tokens with symbolic values to the Fork, which duplicates the token sequence and sends them to the LHS and RHS (through intermediate units in ❷, ❸, and ❹).

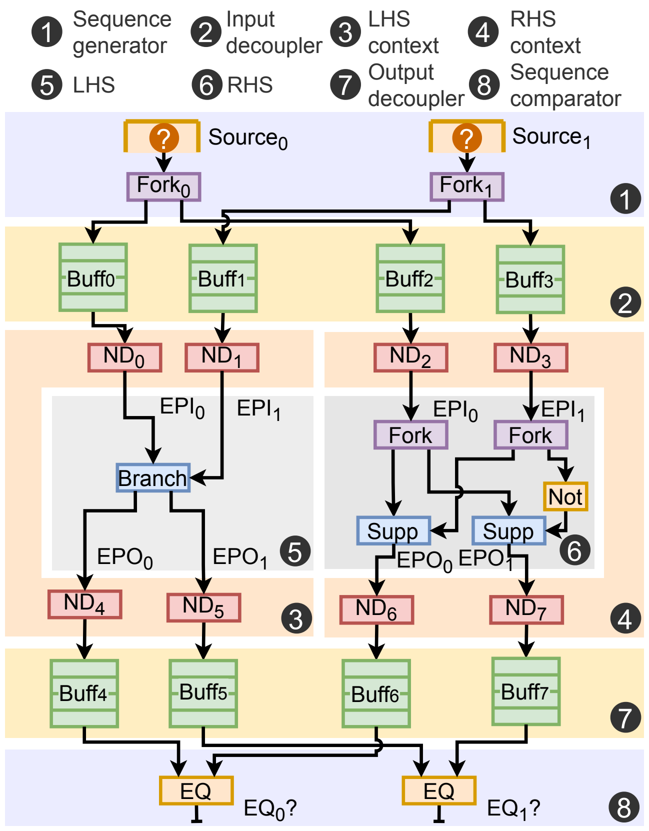


Figure 4. ElasticMiter: an equivalence checking framework for dataflow circuits.

Validating identical token sequences. The *sequence comparator* (❽ in Figure 4) compares the token sequences generated by LHS and RHS. For each EPO, an EQ unit compares every pair of tokens generated by LHS and RHS and outputs a token of value = 1 if the input tokens are identical.

If we directly connected the sequence generator to the EPIs of LHS and RHS (e.g., directly connect $Fork_0$ to EPI_0 without going through $Buff_0$ and ND_0), and the EPOs of LHS and RHS to the sequence comparator (e.g., connect EPO_0 to EQ_0 while skipping $Buff_4$ and ND_4), the circuit would have a similar form as a *miter circuit*, commonly used to verify sequential equivalence [55]. Despite the similarity, this setup without our extensions (i.e., ❷, ❸, ❹, and ❼) does not compare all possible behaviors of the two circuits, as we will see later.

Decoupling LHS and RHS. The *input and output decouplers* (i.e., subcircuits ❷ and ❼ in Figure 4) separate the executions of LHS and RHS via Buffers, enabling ElasticMiter to account for behaviors of LHS and RHS independently: for both LHS and RHS, we insert a Buffer for each EPI after the sequence generator, and for each EPO before the comparator. Without ❷, the sequence generator only sends tokens to LHS and RHS simultaneously, ignoring the cases where LHS and RHS consume tokens at different times. Without ❼, the EQ units only take tokens produced by LHS and RHS at the same cycle, ignoring the cases where LHS and RHS produce tokens at different times. Yet, to check input sequences of any length, Buffers in ❷ and ❼ must have infinite size to ensure that the circuits are always decoupled. Model checkers typically do not support infinite memory [18]. Section 4.3 discusses how to reduce the Buffers to a finite size.

Modeling the circuit context. Equivalent circuits must produce the same output sequence for any arrival times of the input tokens and subject to any backpressure on their outputs—that is, for any *context* that the circuits can be in. Such context does not always need to be perfectly general: in some cases, users may be interested in equivalence only under specific input and output conditions, as we explain in Section 5. In the ElasticMiter fabric, the user can customize *LHS and RHS contexts* in ❸ and ❹ and by configuring the Sources in the sequence generator in ❶ of Figure 4.

Figure 4 shows the most general context—tokens can arrive at different EPIs at any time, and the EPOs can stall at any time and for any duration. To achieve it, we place a nondeterministic wire (ND, see Section 2.1) between each input Buffer and the EPI; it stalls the predecessor Buffer for an arbitrary number of cycles (but eventually propagates the token), thus modeling all possible input latencies. Similarly, an ND is placed between each EPO and the corresponding output Buffer; this models all possible stalls at the EPOs.

4.2 Properties for Checking Sequence Equivalence

This section formulates Definition 3 in terms of the signal behaviors in the ElasticMiter fabric.

Producing and consuming the same sequences. We verify that the following propositions must hold eventually and continue to hold forever: (a) both circuits consume the same number of tokens on every pair of inputs and (b) both circuits produce the same number of tokens on every pair of outputs. We specify proposition (a) in terms of the states of every pair of Buffers in the input decouplers (❷):

$$p_a : b_{LHS,i}.num = b_{RHS,i}.num, \forall b_i \in ❷, \quad (1)$$

where $b_{LHS,i}, b_{RHS,i}$ are the decoupling Buffers of the i-th EPI of LHS and RHS in ❷, and $b.num$ indicates the number of tokens in Buffer b. Since the EQ units consume a pair of tokens, a mismatched number of output tokens will eventually result in "leftover" tokens in the output decoupler. Thus, we specify proposition (b) in terms of every Buffer in ❼:

$$p_b : b.num = 0, \forall b \in ❼. \quad (2)$$

We formulate the following to check that (a) and (b) must hold eventually and continue to hold forever:

$$\textbf{AF AG } (p_a \wedge p_b). \quad (3)$$

Producing tokens with identical pairwise values. To check that the values of each pair of tokens produced by

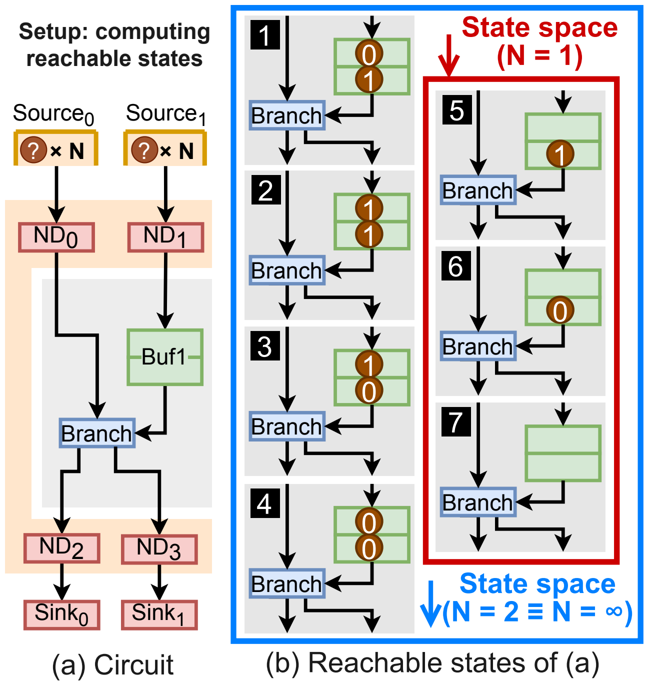


Figure 5. Finite input sequences are sufficient to trigger all possible behaviors. (a) The same circuit as LHS in Figure 4, with a Buffer at the condition input and with the corresponding context and the generator. (b) The state spaces obtained using different token sequence lengths at the input.

LHS and RHS are the same, we check that the EQ units in the sequence comparators (i.e., ❽ in Figure 4) only produce tokens with value = 1; we specify the following formula:

$$\text{AG}\ (EQ_i.valid \longrightarrow EQ_i.data = 1), \forall i \in \{1 \ldots m\}, \qquad (4)$$

where "$a \longrightarrow b$" denotes a implies b.

If both Equations 3 and 4 hold, the circuits are equivalent under the specified context, and one circuit can be replaced by the other; otherwise, they are not equivalent.

4.3 Reducing to Finite-State Model Checking

ElasticMiter can formally verify the equivalence between any two dataflow circuits. There is still a practical challenge: to verify circuit equivalence for input sequences of any lengths, we need *Buffers with infinite slots* in the decouplers (i.e., ❷ and ❼ in Figure 4) to decouple the two circuits and account for all possible behaviors in LHS and RHS. This is unrealistic since it would require infinite memory to represent the state space. However, this section shows that infinite sequences are unnecessary, and suggests a way to determine finite-length sequences that are sufficient for verification.

Using unbounded token sequences is an overkill. Given a dataflow circuit, its all possible states are the reachable states in a general context (i.e., unbounded input sequences with arbitrary latency at EPIs and EPOs). When a circuit receives a sequence with limited lengths, it may only reach a subset of all possible states, leaving some unaccounted for. However, with sufficiently large token sequence lengths, this subset eventually encompasses the entire set of possible states.

For example, Figure 5a describes a dataflow circuit (LHS in Figure 4 with a 2-slot Buffer at EPI_1). Two Sources and Sinks produce and consume token sequences from and to the circuit. To model all possible latency variations, the circuit is equipped with NDs between the Sources and EPIs and between the EPOs and Sinks. For unbounded sequences, each Source outputs an unbounded sequence of tokens with symbolic data. For bounded sequences, each Source holds a counter with initial value equal to N, where N is the length of each input token sequence; the counter counts down when a token is transferred to the receiver, while the Source remains valid until the counter reaches 0.

Regardless of bounded or unbounded sequences, the circuit has a finite number of states and the set of all states can be determined using a reachability analysis tool [18]. Figure 5b reports all 7 states of the circuit (in the blue box). Although unbounded sequences are inapplicable to ElasticMiter, the same set of states is reachable using token sequences with a finite length. Suppose that we use different bounds: When each Source produces sequences with $N = 1$, the states framed with a red box (states 5–7) are all the reachable states; as states 1–4 are not covered, $N = 1$ is insufficient for discovering all states. When each Source produces sequences with $N = 2$, the computed state space is the same as the one using infinite token sequence length. This example indicates that infinite token sequence length is unnecessary.

When building the ElasticMiter fabric, respectively for LHS and RHS, we can determine the minimum token sequence length N sufficient to account for all possible states. Since the sequence generator produces bounded sequences, the Buffers can be sized to N to decouple LHS and RHS.

Determining token sequence length. We employ the following procedure to determine N when building the ElasticMiter fabric. For both LHS and RHS, (1) we independently connect Sources and NDs to EPIs and EPOs to NDs and Sinks (i.e., Figure 5a); (2) we configure the Sources to produce unbounded sequences, and use reachability analysis [11, 18] to determine $states_\infty$—the set of all reachable states reached using unbounded sequences; (3) we then configure the Sources to produce bounded sequences to determine $states_N$—the set of states reached using a sequence length N; we increment the values of N starting from 1 until $states_N \equiv states_\infty$. If LHS and RHS require different values of N; we use the larger one to construct the ElasticMiter fabric.

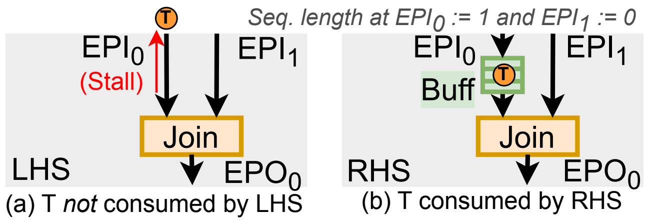


Figure 6. The default context can be too general in some cases. The two circuits are equivalent only if their 2 EPIs receive an identical count of tokens.

When constructing the ElasticMiter fabric, we set the number of slots of each Buffer in the input and output decouplers to N to ensure that the executions of LHS and RHS are always decoupled. We model check the properties in Section 4.2 for every combination of sequence lengths from 0 to N (e.g., if $N = 1$ and the circuits have two EPIs, we separately check for sequence lengths of $(0, 0)$, $(0, 1)$, $(1, 0)$ and $(1, 1)$).

Our strategy ensures that $states_N$ has all the reachable value combinations in the state registers and the primary inputs of the LHS and RHS circuits, and the circuit logic defines all the state transitions. Thus, the model with bounded sequences accounts for all states and all state transitions. Any discrepancy between the produced and consumed sequences of LHS and RHS will be captured by the properties given in Equations 3 and 4.

This concludes the description of ElasticMiter, a strategy for verifying the equivalence between two dataflow circuits.

5 Equivalence Under a Context

Two circuits may not be equivalent in a general sense but can still be used interchangeably within a specific *context*. If the surrounding environment limits their behavior, one circuit can safely replace the other under those constraints. This section explains why contexts are necessary for certain rewrites to be applicable, introduces various contexts, and demonstrates how ElasticMiter incorporates different contexts while verifying contextual equivalence.

5.1 Why Dataflow Circuits May Need a Context?

LHS and RHS in Figure 6 might seem trivially equivalent. Yet, when providing them with sequences of length = 1 at EPI_0 and length = 0 at EPI_1, LHS would not consume any token from EPI_0 since a token is missing at EPI_1 (and the token would be left over in ❷ of Figure 4 forever), whereas RHS would consume a token from EPI_0 and the token would stay in its internal buffer forever; this violates Definition 3 since they consume tokens differently from their inputs despite being given the same token sequences.

Figure 7 shows another example of contextual equivalence: The left circuit (LHS) in the gray-shaded part shows

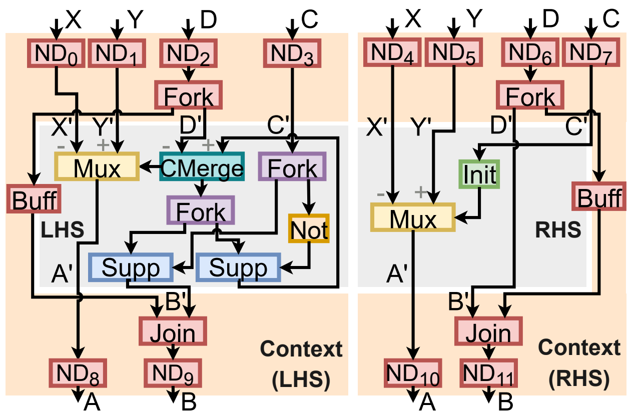


Figure 7. Example of equivalence under a context. The gray-shaded parts model ❺ and ❻ of the ElasticMiter fabric (Figure 4), and the circuits in the orange-shaded part are ❸ and ❹. LHS and RHS are not equivalent in general, but are under the **AT** context shown in the oranged-shaded part, and **SC** and **SV** configurations of ❶, as explained in Section 5.1.

a datapath with a Mux in a loop that propagates to A' tokens arriving at either X' or Y'. The condition of the Mux is calculated using a network composed of a CMerge, Forks, and Suppresses, implementing a dataless structure that mimics the program counter in a loop (analogous to Figure 3): the values of tokens at C' dictate the propagation of tokens that arrived at D' through the cyclic path; accordingly, the CMerge drives the condition of the Mux. Assume one wants to replace this circuit with the one on the right (RHS)—the idea is to send the tokens at D' straight to B', and use an initial token (Init from Section 2.1) followed by a sequence of tokens at C' to calculate the condition of the Mux.

If we plug LHS and RHS in Figure 7 into the ElasticMiter fabric, it will report that the circuits are not equivalent. Sending multiple tokens at D' might result in the CMerge in LHS reordering them (e.g., take a "newer" D' from the left before taking in an "older" D' from the right); thus, the order of tokens outputted at B' might be different from that inputted at D', and the order of tokens outputted at A' will be impacted by the reordering at D'. In contrast, in RHS, the sequence of tokens at B' is always the same as the sequence at D', and the sequence at D' has no impact on the sequence at A'. Additionally, sending an arbitrary count of tokens to D' might result in the RHS having more tokens at B' than the LHS: if the count of tokens at D' is greater than the count of *false* tokens at C', the left Suppress in the LHS will prevent some tokens from reaching B', while, in the RHS, all tokens from D' will be bypassed directly to B'. Furthermore, sending arbitrary values of tokens at C', independent of D', might result in them being consumed in the RHS, while being held back in the LHS: if the very first token at C' happens to be *false* and the second token to be *true*, and if D' receives only one token, the second token of C' will be held back in the LHS (waiting for a token to arrive at D') but will be consumed (stored in the Init) in the RHS. All these scenarios violate Definition 3 and make LHS and RHS nonequivalent.

Although the LHS and RHS of the given examples are, generally, not equivalent, they are equivalent if we constrain the input sequences to prevent the aforementioned scenarios. We leverage the insight that the circuits we are interested in optimizing are produced from imperative code that naturally restricts the circuit's behavior [58]; we infer such restrictions and model them as contexts, as explained in the next section.

5.2 Context of Isolated Dataflow Circuits

Dataflow circuits produced from imperative code have a naturally restricted behavior. For instance, they model a single thread of execution and implement finitely terminating loops that operate as follows: take one token carrying the loop's initialization value, circulate it for some iterations, and finally send out the result, upon loop termination signaled by a *false* token at the loop condition, to the rest of the circuit. Such restrictions impose a **token sequence count (SC)** relation between EPIs, certain **token sequence values (SV)** arriving to some EPIs, and a **token arrival timing (AT)** relation between token arrivals at different EPIs and EPOs.

We model an SC context by configuring the Sources in the sequence generator (❶ in Figure 4). We specify relations that constrain the count of tokens in the sequences arriving at EPI_X and EPI_Y as follows:

$$SC_{equ} : |\mathrm{EPI}_X.seq| = |\mathrm{EPI}_Y.seq| \tag{5}$$

configures EPI_X and EPI_Y to receive the same count of tokens. It models channels in a straight datapath (free of control flow) in a dataflow circuit—they must receive the same count of tokens [22, 31]. It is necessary in the example of Figure 6 to ensure that all inputs of the shown Joins receive sequences of the same length, preventing any discrepancy in consumption between the LHS and RHS. Additionally,

$$SC_{val} : |\mathrm{EPI}_X.seq| = |\{\mathrm{v} \in \mathrm{EPI}_Y.seq \mid \mathrm{v} = val\}| \tag{6}$$

configures EPI_X to receive a count of tokens matching the count of tokens received at EPI_Y with a certain value *val*. It models channels that receive tokens conditionally depending on some control flow decisions [22]. One key usage of such a context is part of modeling the operation of a loop (see the beginning of the section) together with the SV and AT contexts: Specifically, the count of tokens initializing a loop must match the count of tokens terminating the loop and the count of *false* tokens at the loop condition. This context is useful in Figure 7 to force the count of tokens at D' (i.e., the loop initialization) to match the count of *false* tokens at C' (i.e., the loop condition), preventing any tokens at D' from getting blocked in the LHS. The Sources in the sequence generator of the ElasticMiter fabric can be configured to support other SC relations, depending on the user's requirements.

Similarly, we model an SV context by configuring the Sources of the sequence generator (❶ in Figure 4). We constrain the value of a particular token at index i in the sequence of tokens arriving at a particular EPI_X as follows:

$$SV : EPI_X.seq[i] = val. \tag{7}$$

This models the condition of a finitely terminating loop by forcing the last token at a loop condition to take the value *false*. In Figure 7, it forces the last token at C' to be *false*, ensuring that the tokens at C' will be consumed in the same manner in the LHS and the RHS.

Unlike SC and SV contexts that restrict the token sequences, an AT context restricts the time when tokens become available at EPIs. Therefore, it should come after the NDs in ❸ and ❹ of Figure 4 to constrain their effect. We implement AT as a circuit shown in the orange-shaded region of Figure 7, which models a loop circulating a single token at a time: it delays the arrival time of a new token at D' until a token is produced at B'. It does so through the Join that stalls the next token of D' by holding the prior one in a Buffer until a token arrives at B'; thus enabling only a single value of D' to enter the circuit at a time and preventing any reordering from occurring at the CMerge. The provided circuit can be replaced with other circuits for other timing relations, as decided by the user of ElasticMiter.

In general, not every rewrite with input channels coming from one straight datapath will *require* the SC_{equ} context for it to apply. Similarly, not every rewrite with a loop will *require* the SC_{val}, SV, and AT contexts to hold. As we will see in the next section, it depends on how different the structures of the circuits in the LHS and RHS of a rewrite are, and whether this difference in structures could result in them violating some properties of Section 4.2 under arbitrary input counts, values, and latencies.

6 A Graph Rewriting System

HLS-produced dataflow circuits often use excessive steering units [22], as suggested in Figure 3. Prior approaches [22] for optimizing the steering logic do not formally or comprehensively reason about their transformations. We develop a graph rewriting system that takes a conservative HLS-produced dataflow circuit and transforms it into a simpler and faster one using our *provably correct* rewrite transformations. Our rewrites are few and operate on localized *patterns* (i.e., a few dataflow units form a pattern that is replaced by another one), thus, we can verify the latency-insensitive equivalence of each rewrite with ElasticMiter once and before the HLS process. After verification, our rewrites can be applied to any HLS-produced circuit.

6.1 Dataflow Circuit Rewrites

We present a set of rewrites that change the steering logic between computational units without affecting the computational units themselves. We classify them into five categories,

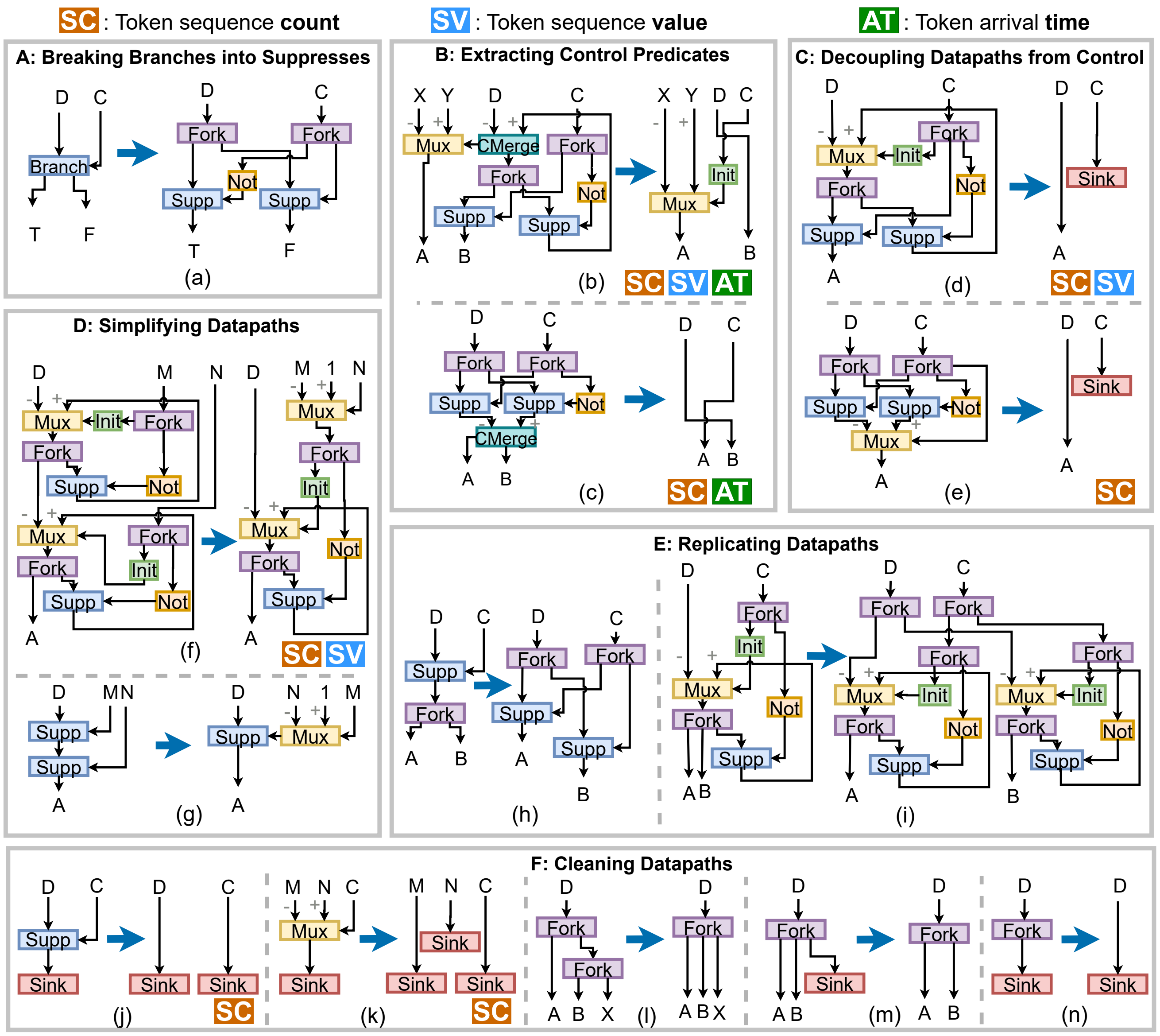


Figure 8. Dataflow circuit rewrites forming a graph rewriting system.

according to their purpose, and include a sixth category that does basic optimizations ensuring that the other rewrites are applicable, as shown in Figure 8. Whenever the pattern of the LHS of one rewrite is found within a larger dataflow circuit, we replace it with the equivalent RHS circuit.

Some of our rewrites are correct for an arbitrary context, whereas some of them require one or more of the contexts described in Section 5.2, as recorded in Figure 8.

Rewrite A: breaking Branches into Suppresses. This rewrite, shown in Figure 8a, applies in the general context. It replaces any Branch with two Suppresses that take the same condition but with opposite signs. It splits the divergence point created by a Branch into two datapaths to allow other rewrites to optimize them separately.

Rewrites B: extracting control predicates. Figure 8b is the rewrite discussed in Figure 7. It generalizes to any number of Muxes driven by the CMerge. It requires the three types of contexts explained in the previous section.

The rewrite in Figure 8c is qualitatively the same but for an if-then-else construct. It requires an AT context identical to that of the orange-labeled region of Figure 7 to prevent the CMerge in the LHS from reordering tokens. Furthermore, it requires the SC_{equ} context of Equation 5 to ensure that the count of tokens arriving at C matches that at D.

Rewrites C: decoupling datapaths from control structures. These rewrites remove circuitry that unnecessarily couples data propagation with the control flow of do-while and if-then-else structures; such couplings appear frequently in C-to-dataflow translation [31]. Figure 8d replaces a Mux in a pointless loop structure with a simple wire that propagates the input D of the Mux to the output A of the Suppress. It requires the SC_{val} of Equation 6 to ensure that the count of tokens at D matches the count of *false* tokens at C. Additionally, it requires the SV context of Equation 7 to force the very last token at C to be *false*, modeling a finitely terminating loop. Figure 8e eliminates a pointless if-then-else, implemented using a Mux and complementary Suppresses. It requires only the SC_{equ} context of Equation 5 to ensure that the count of the tokens arriving at C matches that at D.

Rewrites D: simplifying datapaths. These rewrites simplify datapaths by shifting complexity on the circuitry calculating the conditions. They apply to steering logic with multiple control flow decisions—i.e., nested constructs, such as those in Figures 8f–g, which show a loop nest and a nested if-then-else structures, respectively. Figure 8g applies in a general context. On the contrary, Figure 8f requires the SV context of Equation 7, setting the last tokens of M and N to *false* to model finitely terminating loops. Additionally, it requires two instances of the SC_{val} of Equation 6 to ensure that (1) the count of D matches the count of *false* tokens at M, and (2) the count of M matches the count of *false* tokens at N, modeling the fact that the number of iterations of the outer loop (that has the condition M) matches the number of times the inner loop (that has the condition N) terminates.

Rewrites E: replicating datapaths. These rewrites do not eliminate redundant circuitry, but in fact add more of it to enable other optimizations. They *advance* steering logic over a Fork: Figure 8h advances a Suppress over a Fork, and Figure 8i advances a do-while loop structure over a Fork in the middle of the structure that feeds a Suppress, which in turn completes the loop. They generalize to Forks with an arbitrary number of outputs. Both rewrites apply in a general context. Some added circuitry may end up not being entirely optimized by other rewrites, thus unnecessarily increasing the circuit's area. Therefore, we employ two additional rewrites that we will refer to as Rewrites E'; these simply undo the effect of Rewrites E. We apply them once at the end; otherwise, together with Rewrites E, they could produce a nonterminating sequence of rewrite steps.

Rewrites F: cleaning datapaths. Figures 8j–n show rewrites that remove useless steering components whose presence would make other rewrites inapplicable. These include a Suppress, a Mux, or a Fork that feed a Sink. Besides, it combines a Fork feeding another Fork into one larger Fork. Figure 8j requires the SC_{equ} context of Equation 5 between C and D. Figure 8k requires the SC_{val} context of Equation 5 to ensure that the count of tokens at M matches the count of *false* tokens at C, and the count of tokens at N matches the *true* tokens at C. Figures 8l–n apply in a general context.

6.2 Putting All Rewrites Together

We present the properties of our graph rewriting system and show a working example reducing the datapath of Figure 3b.

Order of rewrites. Our system is not confluent: If two rewrites are simultaneously applicable, choosing one rewrite over the other could result in a different circuit. Hence, we pragmatically order rewrites as follows: Rewrites F, A, E, B, C, and D, and apply them in the following form $(FAEBCD)^*$: we iterate through the ordered set of rewrites, one by one, applying each rewrite once, and repeating until none of the rewrites further apply (we will discuss the termination of our system later in this section). Finally, we apply Rewrites E' once and exit.

Application of rewrites. Our rewrites reduce the number of steering units on every path from any input to any output. We apply them to the datapath of Figure 3b, omitting the network of CMerges (with the gray background in Figure 3b) for simplicity, but it is optimized out similarly by the iterative application of Rewrites F, A, and B. We show the result of each step in Figure 9. None of Rewrites F are applicable, so we start with Rewrite A of Figure 8a, replacing every Branch with two Suppresses, and show the result in Figure 9a. Then, we apply Rewrite E of Figure 8h by advancing a Suppress over a Fork, thus replicating $Supp_0$, resulting in a two-output Fork feeding another two-output Fork. At this point, none of the remaining rewrites are applicable anymore so we start another iteration of our rewrites by applying Rewrite F of Figure 8l to combine the two two-output Forks into one, and show the result in Figure 9b. Following our order, none of Rewrites A or E are applicable, and Rewrites B applies to the network of CMerges, which is omitted for simplicity. Therefore, we apply Rewrite C of Figure 8e, which eliminates Mux_1, $Supp_{01}$, and $Supp_1$. This rewrite feeds the condition of the Suppresses to a Sink; thus results in some Fork outputs feeding Sinks. Additionally, it results in the top Fork feeding directly to the Fork that was at the output of Mux_1. None of Rewrites D apply so we reapply Rewrites F of Figures 8l–n to remove the sinked Fork outputs and combine Forks, and show the result in Figure 9c. Then, Rewrite E of Figure 8i is the first applicable, advancing Mux_0 and $Supp_3$ over a Fork resulting in Figure 9d. In turn, Rewrite C of Figure 8d is the first applicable, eliminating the pointless loop datapath that delayed the data propagation to $cons^2_{x}$, resulting in Figure 9e. The final circuit achieves our goal: a better implementation of the same functionality.

Termination. Although we believe that our system of rewrite rules is finitely terminating when presented with any dataflow graph that might be generated by an HLS compiler—and our experiments confirm this for our benchmarks, as shown in Section 7—a formal proof of termination is nontrivial and beyond the scope of this paper. Our main contribution

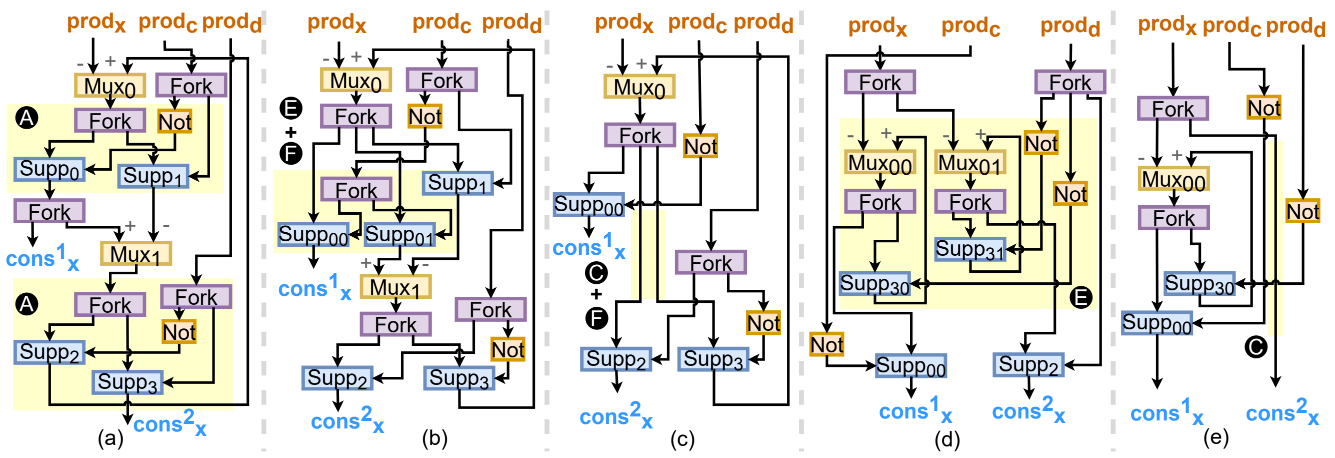
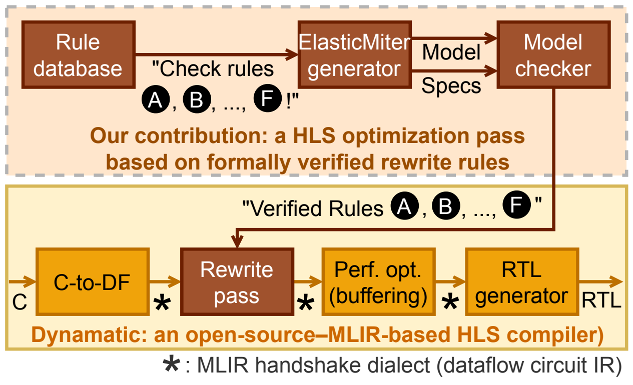


Figure 9. Dataflow circuit transformations. All circuits are equivalent to the one in Figure 3b but with different steering logic. We omit the Mux conditions for simplicity. Each step indicates the applied rewrites among A–F.



Rule database
"Check rules A, B, ..., F!"
ElasticMiter generator
Model
Specs
Model checker
Our contribution: a HLS optimization pass based on formally verified rewrite rules
"Verified Rules A, B, ..., F"
C
C-to-DF
Rewrite pass
Perf. opt. (buffering)
RTL generator
RTL
Dynamatic: an open-source–MLIR-based HLS compiler)
*: MLIR handshake dialect (dataflow circuit IR)

Figure 10. Our complete workflow.

is providing a framework to prove the correctness of dataflow circuit rewrites. The specific rewrite system discussed in this section serves only as an example of a useful system that achieves measurable advantages, as reported in our results.

7 Evaluation

We formally verify the rewrites (Section 6) in an acceptable runtime leveraging ElasticMiter (Section 4). We then employ our verified rewrites in a compiler optimization pass and show experimentally that it improves circuit performance and area. Our research artifact is publicly available [25].

7.1 Methodology

Our framework is shown in Figure 10. It has two parts: (1) Our verifier that takes rules from a predefined database, generates the ElasticMiter fabric (Section 4), and employs a model checker to verify them. (2) Our compiler pass that implements our rewrites (Section 6), integrated into Dynamatic [31], an open-source MLIR-based HLS compiler [42].

Formally verifying our rewrites. We model dataflow circuits, their context, and the formal properties using the SMV language [38]. The SMV models of the units are identical to their RTL implementation; the data signals are abstracted using 1-bit for scalability [60]. We use nuXmv [11] to prove temporal properties and extract the set of reachable states (using BDD-based reachability analysis). We determine the configurations of ElasticMiter (i.e., the token sequence length) using the approach in Section 4.3 and verify latency insensitive equivalence using the properties of Section 4.2.

Evaluating the effectiveness of our rewrites. We implement an MLIR [42] compiler pass that greedily and iteratively applies our rewrite patterns until no match is found. We integrate this pass into the HLS flow of Dynamatic which implements a simple strategy [31] for dataflow circuit generation from C (*C-to-DF* in Figure 10). We place our graph-rewriting pass (*Rewrite pass*) right after the *C-to-DF* step and before buffer placement. We apply a simple buffer placement heuristic: we insert a buffer to break every combinational cycle and, additionally, a buffer with a large depth at every fork output; then, we iteratively reduce the depths only if this reduction has no negative effect on the circuit's latency reported in simulation. We use the circuits produced by Dynamatic in the *C-to-DF* step as a baseline, after passing them through the same buffer placement heuristic. We use Dynamatic's *RTL generator* for all circuits. We synthesize the generated VHDL netlists using Vivado [57] with a clock-period constraint of 4 ns, targeting a Kintex-7 FPGA. We simulate the designs with ModelSim [39] and use a set of test vectors for functional verification. We measure (1) the cycle count obtained from simulation, (2) the clock period (CP) from the postrouting timing analysis, and (3) resource usage (i.e., LUT, FF, and DSP counts) reported from Vivado after placement and routing. Our goal is a smaller LUT and FF usage, and a faster execution time compared to the baseline.

Table 1. Verifying the rewrites in Section 6.1.

Rewrite	Sequence length		Model checking		
	Tokens	Time (s)	Length	Value	Time (s)
Fig. 8a	1	0.2	✓	✓	1.6
Fig. 8b	3	35.2	✓	✓	115.3
Fig. 8c	2	1.2	✓	✓	2.0
Fig. 8d	4	2.1	✓	✓	18.1
Fig. 8e	1	0.2	✓	✓	0.4
Fig. 8f	4	16.3	✓	✓	854.2
Fig. 8g	1	0.2	✓	✓	1.9
Fig. 8h	1	0.2	✓	✓	2.2
Fig. 8i	3	2.1	✓	✓	98.6
Fig. 8j	1	0.1	✓	✓	0.2
Fig. 8k	1	0.2	✓	✓	0.3
Fig. 8l	1	0.2	✓	✓	0.7
Fig. 8m	1	0.2	✓	✓	0.5
Fig. 8n	1	0.1	✓	✓	0.1

7.2 Results: Rewrite Rule Verification

Table 1 reports the results and runtime statistics of verifying the correctness of the rewrites of Figure 8 using ElasticMiter. **Sequence length** reports the token sequence length bound used in the framework (Section 4.3) and the time spent on determining it for each rewrite in column **Rewrite**. **Model checking** reports the model checking results: **Length** and **Value** describe the sequence value and length equivalence (Equations 3 and 4); **Runtime** indicates the total verification runtime. Most verification runs finish within seconds. Rules with loops (Figure 8b, d, and i) have a larger state space and take a longer time to prove. nuXmv has poorer scalability on **AF** properties: for the rule in Figure 8f, the **Value** property only took 50 s, whereas the **Length** property took 800 s to prove. Speeding up model checking is orthogonal to our contribution and ElasticMiter could, if needed, be complemented with existing verification speedup strategies (e.g., abstractions or invariants [12, 13, 18, 58]). Note that verification runtime is not a concern for us: we need to prove a limited number of small rewrites once and then we can optimize any HLS-produced dataflow circuit.

7.3 Results: Effectiveness of Our Rewrite Pass

We evaluate our rewrites using a set of HLS kernels commonly used for dynamically-scheduled HLS [14, 31, 47, 54]. Circuits with a more complex and irregular control-flow structure have more steering logic and thus can benefit more from our rewrites. When our rewrites remove steering units, they reduce area and allow units in independent control structures to run in parallel, thus increasing performance.

Table 2 summarizes the timing and resources of the circuits generated by our approach (Ours) compared to the baseline Dynamatic [31]. Figure 11 graphically represents our results normalized to those of Dynamatic. In most benchmarks, our results Pareto-dominate the baseline: they improve the overall execution time without negatively affecting the area, if not actually reducing it.

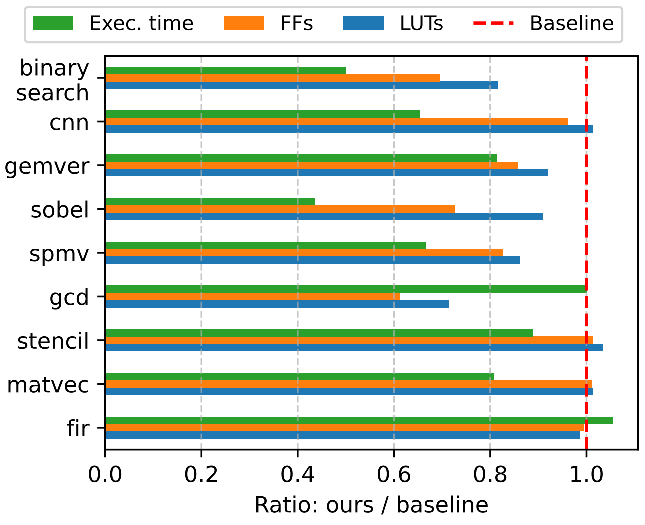


Figure 11. Results of our rewrites normalized to baseline *Dynamatic* circuits [31].

Specifically, **gemver** (30 x 30 matrix dimension), **binary search** (101 vector dimension), and **cnn** (10 x 10 x 10 image dimension and 1 x 1 kernel dimension) feature multiple loop nests in succession. They benefit from the rewrites of Figure 8b, d, and f to varying degrees, depending on the amount of data and memory dependencies. **sobel** (15 x 15 image dimension and 3 x 3 kernel dimension) and **spmv** (10 vector dimension) also feature multiple loop nests along with a few if-then-else structures; thus, they additionally benefit from the rewrites of Figure 8c, e, and g. These rewrites parallelize multiple control structures and achieve tangible advantages on all metrics. **gcd** (computation between 7,966,496 and 314,080,416) also has a few loops and if-then-else structures, and it still benefits from the rewrites; yet, the gain appears mostly as an area reduction and less in cycle count because the parallelism is limited by multiple data dependencies. Finally, **stencil** (30 x 30 input dimension and 3x3 kernel dimension), **matvec** (100 x 100 matrix dimension), and **fir** (1000 vector dimension) have simple control structures; as expected, they do not benefit much from our rewrites. In fact, **fir** witnesses a minor CP degradation, yet it is caused by orthogonal effects: our non-delay-driven buffering heuristic and the place-and-route heuristics of the FPGA implementation tool [57].

Overall, our rewrites produce smaller and faster dataflow circuits that are formally proven equivalent to the original ones. They remove a multitude of unnecessary components and thus improve resources, critical paths, and clock cycles.

8 Related Work

Dataflow circuit optimizations. Prior work explored the generation of dataflow circuits from imperative code [6, 28, 31, 35]; this includes Dynamatic [31], the source of

Table 2. Results of our rewrites (*Ours*) on the circuits generated by *Dynamatic* contrasted to the original circuits ([31]).

Benchmark	**Cycles**		**CP (ns)**		**Exec. time (us)**			**LUTs**			**FFs**			**DSPs**
	[31]	**Ours**	**[31]**	**Ours**	**[31]**	**Ours**		**[31]**	**Ours**		**[31]**	**Ours**		
binary search	282	155	4.4	4.0	1.2	0.6	**-50%**	1,131	924	**-18%**	1,061	738	**-30%**	0
cnn	6,659	5,506	4.8	3.8	32	20.9	**-35%**	2,181	2,212	**1%**	1,326	1,275	**-4%**	3
gemver	5,007	4,972	6.0	4.9	30.0	24.4	**-19%**	3,366	3,093	**-8%**	2,960	2,541	**-14%**	22
sobel	4,173	2,159	7	5.9	29.2	12.7	**-57%**	2,259	2,053	**-9%**	2,107	1,533	**-27%**	6
spmv	61	46	5.2	4.5	0.3	0.2	**-33%**	1,534	1,321	**-14%**	1,359	1,124	**-17%**	3
gcd	77	76	5.1	5.3	0.4	0.4	**0%**	1,656	1,184	**-29%**	1,233	754	**-39%**	0
stencil 2d	432	433	4.2	3.8	1.8	1.6	**-11%**	869	898	**3%**	678	687	**1%**	3
matvec	15,740	15,725	5.2	4.2	81.8	66.0	**-19%**	685	694	**1%**	525	531	**1%**	4
fir	1,007	1,007	3.7	3.9	3.7	3.9	**5%**	388	383	**-1%**	356	354	**-1%**	3

the dataflow circuits for our evaluation. Dataflow circuits strongly relate to dataflow machines [1, 27, 45]. Ballance et al. [3] and Campbell et al. [8] proposed program analysis to optimize software code for these machines. Although they produced software code, their compiler representations can be interpreted as dataflow circuits. Elakhras et al. proposed a *fast token delivery* strategy [22, 24] that used those representations to improve the area and performance of dataflow circuits. Cheng et al. [15] analyzed high-level code to exploit more parallelism in statically inferrable situations. All these efforts only target better conversions of high-level code into dataflow circuits, whereas we propose formally verified circuit-to-circuit transformations that produce provably correct, optimized dataflow circuits without the need for any high-level code analysis.

Sequential synthesis [40, 51, 52, 58–60] reduces circuit complexity while keeping circuits before and after optimization sequentially equivalent. Although applicable to dataflow circuits [58], these approaches are complementary to our optimization techniques—our optimization approach alters the sequential behavior of the circuit to reduce execution latency and area cost, while still maintaining the same functionality.

Verification techniques for accelerator synthesis. Recent efforts aim at verifying the HLS compilation process [26, 29]: These tools target statically-scheduled circuits, while we focus on the HLS of dataflow circuits. Pan et al. [44] proposed a bounded strategy for checking if an RTL module behaves the same under arbitrary stalls; they do not determine a sufficient bound value for the proof, whereas ElasticMiter addresses this issue (Section 4.3). Lin et al. [37] proposed an approach for translation validation between an LLVM program and the program compiled using a dataflow compiler; yet, their approach does not apply when the compiler restructures the code. Pouchet et al. [48] introduced an equivalence checking tool for HLS source-to-source transformation. Their approach is limited to programs with static control flow and targets a different abstraction level (C code).

Our definitions of equivalent sequences and latency-insensitive equivalence are inspired by Carloni et al. [10]. They do not provide a problem encoding that is directly compatible with model checking. We take advantage of dataflow circuits' well-defined in/out ports and specify our definition over them; this facilitates the encoding of the equivalence problem (ElasticMiter) for any two circuits and enables automated model checking proofs.

Program equivalence checking is an important problem in software verification [16, 43, 49, 61]. Dataflow circuits cannot directly benefit from these techniques: at a semantic level, dataflow units can execute in arbitrary order permitted by data dependencies; yet, software equivalence checking assumes that instructions execute in a sequence.

9 Conclusions

Dataflow circuits are a promising HLS target, but their transformations lack formal verification strategies that would enable their safe and systematic usage in HLS. This is due to the fact that the notion of equivalence in dataflow circuits differs from that in ordinary digital circuits. In this work, we develop a generic framework that exploits a dataflow-specific equivalence definition to prove the equivalence of two dataflow circuits. We use our framework to verify a graph rewriting system for dataflow circuit simplification, and employ the verified graph rewrites to transform HLS-produced dataflow circuits into their equivalent, cheaper, and faster counterparts. Our work complements existing software and hardware verification strategies and enables the development of formally verified dataflow HLS compilers.

Acknowledgments

We thank the anonymous reviewers and our shepherd Guy Steele for their valuable feedback. Ayatallah Elakhras is supported by a Google PhD Fellowship in Distributed Systems and Parallel Computing. Jiahui Xu is supported by the Swiss National Science Foundation (grant number 215747) and the ETH Future Computing Laboratory (donation from Huawei Technologies).

A Artifact Appendix

A.1 Abstract

This artifact contains all the source codes and benchmarks of ElasticMiter. It utilizes a Dockerfile to set up the environment and scripts to replicate our experiments in Section 7 and generate data for Table 1, Table 2, and Figure 11.

A.2 Artifact check-list (meta-information)

- **Program:** The source code for the rewriting patterns and the verification framework is available in `elastic-miter/`. The source code for the rewriting optimization is available in `dynamatic/`. They are both available in the Zenodo archive.
- **Compilation:** Use Dockerfile to build the project.
- **Run-time environment:** The experiments run in an Ubuntu 22.04 docker (Dockerfile provided). Our experiments require Vivado 2019.1 and ModelSim 20.1 (see Appendix A.3.3).
- **Hardware:** AMD Ryzen 7 PRO 5850U (or any similar processors) with $\geq$ 32 GB memory.
- **Output:** Generate simulation reports, synthesis reports, and log files.
- **Experiments:** Run all experiments using the Bash scripts. `README.md` documents the individual experiments.
- **Disk space required:** The docker image and the software installation packages add up to $\geq$ 120 GB.
- **Time needed to prepare workflow:** Approximately 2.5 hours to setup the workflow.
- **Time needed to complete experiments:** Around 3 hours to complete all experiments.
- **Publicly available:** Yes.
- **Code licenses:** MIT license.
- **Workflow automation framework used:** Docker.
- **Archived:** Archived on Zenodo with DOI: 10.5281/zenodo.14776038.

A.3 Description

The detailed descriptions are documented in `README.md` in the Zenodo archive [25]. This subsection describes some notable hardware/software requirements (e.g., proprietary software dependencies).

A.3.1 How to access.

The artifact is publicly available at https://doi.org/10.5281/zenodo.14776038. In the archive, `asplos25fall-elastic-miter.zip` contains all the source codes, benchmarks, scripts, and environment setup files. `generated-files.zip` contains all the generated files, i.e., log files, synthesis/simulation reports, and tables.

A.3.2 Hardware dependencies.

A Ubuntu 22.04-LTS Linux machine with at least 120 GB of free disk space and 32 GB memory.

A.3.3 Software dependencies.

The experiments depend on proprietary software. Vivado (version 2019.1) [57] and ModelSim (version 20.1) [39] have free versions. `README.md` provides the instructions for downloading the software.

All the remaining dependencies are automatically configured by `Dockerfile`.

A.4 Installation

This subsection describes the steps for generating the runtime Docker environment for our experiments.

- Install Docker (https://www.docker.com/).
- Download the Zenodo archive [25] and follow the instructions in the `README.md` to download the proprietary software dependencies (see Appendix A.3.3), obtain the Gurobi license, and build the docker image.

A.5 Experiment workflow

Follow the instructions in `README.md`. Launch the container and run all experiments using `exp_gen_tab_1.sh` and `exp_gen_tab_2.sh`.

A.6 Evaluation and expected results

The produced data corresponding to the entries in Table 1, Table 2, and Figure 11 will be generated with identical or nearly identical numbers as reported (location of the files are documented in `README.md`). The produced verification runtime is not expected to be identical but is expected to be similar to the ones reported in Table 1.

References

[1] Arvind and Rishiyur S. Nikhil. 1990. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Trans. Comput.* 39 (March 1990), 300–318. https://doi.org/10.1109/12.48862

[2] David F. Bacon, Rodric Rabbah, and Sunil Shukla. 2013. FPGA Programming for the Masses. *Commun. ACM* 54, 4 (April 2013), 56–63. https://doi.org/10.1145/2436256.2436271

[3] Robert A. Ballance, Arthur B. Maccabe, and Karl J. Ottenstien. 1990. The Program Dependence Web:A Representation Supporting Control-, Data-, and Demand-Driven Interpretation of Imperative Languages. In *Proceedings of the 11th ACM SIGPLAN Conference on Programming Language Design and Implementation*. White Plains, NY, 257–71. https://doi.org/10.1145/93542.93578

[4] Aaron R. Bradley. 2011. SAT-Based Model Checking without Unrolling. In *Proceedings of the 12th International Workshop on Verification, Model Checking, and Abstract Interpretation*. Austin, TX, 70–87. https://doi.org/10.1007/978-3-642-18275-4_7

[5] Robert Brayton and Alan Mishchenko. 2010. ABC: An Academic Industrial-Strength Verification Tool. In *Proceedings of the 22nd International Conference on Computer Aided Verification*. Edinburgh, 24–40. https://doi.org/10.1007/978-3-642-14295-6_5

[6] Mihai Budiu, Pedro V. Artigas, and Seth Copen Goldstein. 2005. Dataflow: A Complement to Superscalar. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*. Austin, TX, 177–86. https://doi.org/10.1109/ISPASS.2005.1430572

[7] Dmitry Bufistov, Jordi Cortadella, Mike Kishinevsky, and Sachin Sapatnekar. 2007. A General Model for Performance Optimization of Sequential Systems. In *Proceedings of the International Conference on Computer-Aided Design*. San Jose, CA, 362–69. https://doi.org/10.1109/ICCAD.2007.4397291

[8] Philip L. Campbell, Ksheerabdhi Krishna, and Robert A. Ballance. 1993. *Refining and Defining the Program Dependence Web*. Technical Report. University of New Mexico. https://www.osti.gov/biblio/6231712

[9] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen D. Brown, and Jason H. Anderson. 2013. LegUp: An Open-Source High-Level Synthesis Tool for FPGA-Based Processor/Accelerator Systems. *ACM Transactions on Embedded Computing Systems* 13, 2 (Sept. 2013), 24:1–24:27. https://doi.org/10.1145/2514740

[10] Luca P. Carloni, Kenneth L. McMillan, and Alberto L. Sangiovanni-Vincentelli. 2001. Theory of Latency-Insensitive Design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 20, 9 (Sept. 2001), 1059–76. https://doi.org/10.1109/43.945302

[11] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. 2014. The nuXmv Symbolic Model Checker. In *Proceedings of the 26th International Conference on Computer Aided Verification*. Vienna, 334–42. https://doi.org/10.1007/978-3-319-08867-9_22

[12] Satrajit Chatterjee and Michael Kishinevsky. 2012. Automatic Generation of Inductive Invariants from High-Level Microarchitectural Models of Communication Fabrics. *Formal Methods in System Design* 40 (2012), 147–69. https://doi.org/10.1007/s10703-011-0134-0

[13] Satrajit Chatterjee, Michael Kishinevsky, and Umit Y. Ogras. 2012. xMAS: Quick Formal Modeling of Communication Fabrics to Enable Verification. *IEEE Design & Test of Computers* 29, 3 (June 2012), 80–88. https://doi.org/10.1109/MDT.2012.2205998

[14] Jianyi Cheng, Lana Josipović, George A. Constantinides, Paolo Ienne, and John Wickerson. 2020. Combining Dynamic & Static Scheduling in High-Level Synthesis. In *Proceedings of the 28th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Seaside, CA, 288–98. https://doi.org/10.1145/3373087.3375297

[15] Jianyi Cheng, Lana Josipović, George A. Constantinides, and John Wickerson. 2022. Dynamic Inter-Block Scheduling for HLS. In *Proceedings of the 32nd International Conference on Field-Programmable Logic and Applications*. Belfast, UK, 243–52. https://doi.org/10.1109/FPL57034.2022.00045

[16] Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. 2019. Semantic Program Alignment for Equivalence Checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Phoenix, AZ, 1027–1040. https://doi.org/10.1145/3314221.3314596

[17] Edmund Clarke, Kenneth McMillan, Sérgio Campos, and Vicky Hartonas-Garmhausen. 1996. Symbolic Model Checking. In *Proceedings of the 8th International Conference on Computer Aided Verification*. New Brunswick, NJ, 419–22. https://doi.org/10.1007/3-540-61474-5_93

[18] Edmund M. Clarke, Thomas A. Henzinger, and Helmut Veith. 2018. *Introduction to Model Checking*. Springer International Publishing, Cham, 1–26.

[19] Jordi Cortadella, Mike Kishinevsky, and Bill Grundmann. 2006. Synthesis of Synchronous Elastic Architectures. In *Proceedings of the 43rd Design Automation Conference*. San Francisco, CA, 657–62. https://doi.org/10.1145/1146909.1147077

[20] Giovanni De Micheli. 1994. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, New York.

[21] Stephen A. Edwards, Richard Townsend, and Martha A. Kim. 2017. Compositional Dataflow Circuits. In *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design*. Vienna, 175–84. https://doi.org/10.1145/3127041.3127055

[22] Ayatallah Elakhras, Andrea Guerrieri, Lana Josipović, and Paolo Ienne. 2022. Unleashing Parallelism in Elastic Circuits with Faster Token Delivery. In *Proceedings of the 32nd International Conference on Field-Programmable Logic and Applications*. Belfast, UK, 253–61. https://doi.org/10.1109/FPL57034.2022.00046

[23] Ayatallah Elakhras, Andrea Guerrieri, Lana Josipović, and Paolo Ienne. 2024. Survival of the Fastest: Enabling More Out-of-Order Execution in Dataflow Circuits. In *Proceedings of the 32nd International Symposium on Field-Programmable Gate Arrays*. Monterey, CA, 44–54. https://doi.org/10.1145/3626202.3637556

[24] Ayatallah Elakhras, Riya Sawhney, Andrea Guerrieri, Lana Josipović, and Paolo Ienne. 2023. Straight to the Queue: Fast Load-Store Queue Allocation in Dataflow Circuits. In *Proceedings of the 31st International Symposium on Field-Programmable Gate Arrays*. Monterey, CA, 39–45. https://doi.org/10.1145/3543622.3573050

[25] Ayatallah Elakhras and Jiahui Xu. 2025. Research Artifact of ElasticMiter: Formally Verified Dataflow Circuit Rewrites. https://doi.org/10.5281/zenodo.14776038

[26] Florian Faissole, George A. Constantinides, and David Thomas. 2019. 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, 315. https://doi.org/10.1109/FCCM.2019.00056

[27] V. G. Grafe, G. S. Davidson, J. E. Hoch, and V. P. Holmes. 1989. The Epsilon Dataflow Processor. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*. Jerusalem, 36–45. https://doi.org/10.1145/74926.74930

[28] Hagen Gädke-Lütjens. 2011. *Dynamic Scheduling in High-Level Compilation for Adaptive Computers*. Ph.D. Thesis. Technischen Universität Braunschweig, Braunschweig, Germany. https://doi.org/10.24355/dbbs.084-201105300920-0

[29] Yann Herklotz, James D Pollard, Nadesh Ramanathan, and John Wickerson. 2021. Formal Verification of High-Level Synthesis. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–30. https://doi.org/10.1145/3485494

[30] Hans M. Jacobson, Prabhakar N. Kudva, Pradip Bose, Peter W. Cook, Stanley E. Schuster, Eric G. Mercer, and Chris J. Myers. 2002. Synchronous Interlocked Pipelines. In *Proceedings of the 8th International Symposium on Advanced Research in Asynchronous Circuits and Systems*. Manchester, 3–12. https://doi.org/10.1109/ASYNC.2002.1000291

[31] Lana Josipović, Radhika Ghosal, and Paolo Ienne. 2018. Dynamically Scheduled High-level Synthesis. In *Proceedings of the 26th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Monterey, CA, 127–36. https://doi.org/10.1145/3174243.3174264

[32] Lana Josipović, Andrea Guerrieri, and Paolo Ienne. 2019. Speculative Dataflow Circuits. In *Proceedings of the 27th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Seaside, CA, 162–71. https://doi.org/10.1145/3289602.3293914

[33] Lana Josipović, Axel Marmet, Andrea Guerrieri, and Paolo Ienne. 2022. Resource Sharing in Dataflow Circuits. In *Proceedings of the 30th IEEE Symposium on Field-Programmable Custom Computing Machines*. New York, 1–9. https://doi.org/10.1109/FCCM53951.2022.9786084

[34] Lana Josipović, Shabnam Sheikhha, Andrea Guerrieri, Paolo Ienne, and Jordi Cortadella. 2020. 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, 186–96. https://doi.org/10.1145/3373087.3375314

[35] Nico Kasprzyk. 2005. *COMRADE—Ein Hochsprachen-Compiler für Adaptive Computersysteme*. Ph.D. Thesis. Technischen Universität Braunschweig, Braunschweig, Germany.

[36] Christoph Kern and Mark R Greenstreet. 1999. Formal Verification in Hardware Design: A Survey. *ACM Transactions on Design Automation of Electronic Systems* 4, 2 (April 1999), 123–93. https://doi.org/10.1145/307988.307989

[37] Zhengyao Lin, Joshua Gancher, and Bryan Parno. 2024. FlowCert: Translation Validation for Asynchronous Dataflow via Dynamic Fractional Permissions. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2, Article 289 (Oct. 2024). https://doi.org/10.1145/3689729

[38] Kenneth L. McMillan. 1993. *The SMV System*. Springer US, Boston, MA, 61–85. https://doi.org/10.1007/978-1-4615-3190-6_4

[39] Mentor Graphics. 2016. ModelSim. https://www.mentor.com/products/fv/modelsim/

[40] Alan Mishchenko, Michael Case, Robert Brayton, and Stephen Jang. 2008. Scalable and Scalably-Verifiable Sequential Synthesis. In *Proceedings of the 27th International Conference on Computer-Aided Design*. San Jose, CA, 234–241. https://doi.org/10.1109/ICCAD.2008.4681580

[41] Mahim Mishra, Timothy J. Callahan, Tiberiu Chelcea, Girish Venkataramani, Seth Copen Goldstein, and Mihai Budiu. 2006. Tartan: Evaluating Spatial Computation for Whole Program Execution. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*. 163–74. https://doi.org/10.1145/1168917.1168878

[42] Multi-Level IR Compiler Framework 2020. *https://mlir.llvm.org/*. Multi-Level IR Compiler Framework. https://mlir.llvm.org/

[43] George C. Necula. 2000. Translation Validation for an Optimizing Compiler. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*. Vancouver, British Columbia, 83–94. https://doi.org/10.1145/358438.349314

[44] Peitian Pan and Christopher Batten. 2023. Formal Verification of the Stall Invariant Property for Latency-Insensitive RTL Modules. In *Proceedings of the 21st ACM/IEEE International Conference on Formal Methods and Models for System Design*. Hamburg, Germany, 148–158. https://doi.org/10.1145/3610579.3611081

[45] Gregory Michael Papadopoulos. 1998. *Implementation of a General-Purpose Dataflow Multiprocessor*. Ph. D. Dissertation. Massachusetts Institute of Technology, Laboratory for Computer Science. http://hdl.handle.net/1721.1/27967

[46] Nir Piterman and Amir Pnueli. 2018. *Temporal Logic and Fair Discrete Systems*. Springer International Publishing, Cham, 27–73. https://doi.org/10.1007/978-3-319-10575-8_2

[47] Louis-Noël Pouchet. 2012. *Polybench: The Polyhedral Benchmark Suite*. https://sourceforge.net/p/polybench/wiki/Home/

[48] Louis-Noël Pouchet, Emily Tucker, Niansong Zhang, Hongzheng Chen, Debjit Pal, Gabriel Rodríguez, and Zhiru Zhang. 2024. Formal Verification of Source-to-Source Transformations for HLS. In *Proceedings of the 32nd ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Monterey, CA, 97–107. https://doi.org/10.1145/3626202.3637563

[49] David A. Ramos and Dawson R. Engler. 2011. Practical, Low-Effort Equivalence Verification of Real Code. In *Proceedings of 23rd International Conference on Computer Aided Verification*. Snowbird, UT, 669–685. https://doi.org/10.1007/978-3-642-22110-1_55

[50] Carmine Rizzi, Andrea Guerrieri, Paolo Ienne, and Lana Josipović. 2022. 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, 375–83. https://doi.org/10.1109/FPL57034.2022.00063

[51] Ellen. M. Sentovich, Kanwar Jit Singh, Cho Moon, Hamid Savoj, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. 1992. Sequential Circuit Design Using Synthesis and Optimization. In *Proceedings 1992 IEEE International Conference on Computer Design: VLSI in Computers & Processors*. Cambridge, MA, 328–33. https://doi.org/10.1109/ICCD.1992.276282

[52] Ellen. M. Sentovich, Horia Toma, and Gérard Berry. 1996. Latch Optimization in Circuits Generated from High-Level Descriptions. In *Proceedings of the 15th International Conference on Computer-Aided Design*. San Jose, CA, 428–35. https://doi.org/10.1109/ICCAD.1996.569833

[53] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. 2000. 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, 127–144. https://doi.org/10.1007/3-540-40922-X_8

[54] Robert Szafarczyk, Syed Waqar Nabi, and Wim Vanderbauwhede. 2023. Compiler Discovered Dynamic Scheduling of Irregular Code in High-Level Synthesis. In *Proceedings of the 33rd International Conference on Field-Programmable Logic and Applications*. Gothenburg, Sweden. https://doi.org/10.1109/FPL60245.2023.00009

[55] C. A. J. Van Eijk. 2000. Sequential Equivalence Checking Based on Structural Similarities. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 19, 7 (Aug. 2000), 814–819. https://doi.org/10.1109/43.851997

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

[57] Xilinx Inc. 2019. *Vivado Design Suite*. Xilinx Inc. http://www.xilinx.com/products/design-tools/vivado.html

[58] Jiahui Xu and Lana Josipović. 2023. Automatic Inductive Invariant Generation for Scalable Dataflow Circuit Verification. In *Proceedings of the 42nd International Conference on Computer-Aided Design*. San Francisco, CA, 1–9. https://doi.org/10.1109/ICCAD57390.2023.10323796

[59] Jiahui Xu and Lana Josipović. 2024. Suppressing Spurious Dynamism of Dataflow Circuits Via Latency and Occupancy Balancing. In *Proceedings of the 32nd ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. Monterey, CA, 188–98. https://doi.org/10.1145/3626202.3637570

[60] Jiahui Xu, Emmet Murphy, Jordi Cortadella, and Lana Josipović. 2023. 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, 27–37. https://doi.org/10.1145/3543622.3573196

[61] Anna Zaks and Amir Pnueli. 2008. CoVaC: Compiler Validation by Program Analysis of the Cross-Product. In *Proceedings of 2008 International Symposium on Formal Methods*. Turku, Finland, 35–51. https://doi.org/10.1007/978-3-540-68237-0_5