Graphiti: Formally Verified Out-of-Order Execution in Dataflow Circuits

Yann Herklotz
EPFL
Lausanne, Switzerland
yann.herklotz@epfl.ch

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

Martina Camaioni
EPFL
Lausanne, Switzerland
martina.camaioni@epfl.ch

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

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

Thomas Bourgeat
EPFL
Lausanne, Switzerland
thomas.bourgeat@epfl.ch

Abstract

High-level synthesis (HLS) tools automatically synthesise hardware from imperative programs and have seen a significant rise in adoption in both industry and academia. To deliver high-quality hardware designs for increasingly general purpose programs, HLS compilers have to become more aggressive. For the most irregular programs, HLS tools generating dataflow circuits show promising performance by adapting and specializing key ideas from processor architectures, like out-of-order execution and speculation. However, the complexity of these transformations makes them difficult to reason about, increasing the risk of subtle bugs and potentially delaying their adoption in a conservative industry where bugs can be extremely costly.

This paper introduces Graphiti, a framework embedded in the Lean 4 proof assistant designed to formally reason about and manipulate dataflow circuits at the core of these HLS tools. We develop a metatheory of graph refinement that allows us to verify a general-purpose dataflow circuit rewriting algorithm. Using this framework, we formally verify a loop rewrite that introduces out-of-order execution into a dataflow circuit. Our evaluation shows that the resulting verified optimization pipeline achieves a 2.1× speedup over the in-order HLS flow and a 5.8× speedup over a verified HLS tool generating a static state machine. We also show that it achieves the same performance compared to an existing *unverified* approach which introduces out-of-order execution.

Graphiti is a step toward a fully-verified HLS flow targeting dataflow circuits. In the interim, it can serve as an extensible, verified, optimizing engine that can be integrated into existing dataflow HLS compilers.

CCS Concepts: • **Hardware → Datapath optimization**; **Theorem proving and SAT solving**; • **Computer systems organization → Data flow architectures**; • **Software and its engineering → Software verification**.

Keywords: high-level synthesis; interactive theorem proving; Lean 4

ACM Reference Format:
Yann Herklotz, Ayatallah Elakhras, Martina Camaioni, Paolo Ienne, Lana Josipović, and Thomas Bourgeat. 2026. Graphiti: Formally Verified Out-of-Order Execution in Dataflow Circuits. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '26), March 22–26, 2026, Pittsburgh, PA, USA*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3779212.3790166

1 Introduction

The last few years have seen the integration of many hardware accelerators [16, 35, 41, 55, 57, 63] into most computing stacks, from data centers to handheld devices. While the traditional design methodology consisting of manually writing the register-transfer level (RTL) description of the hardware designs is still considered the gold standard if one can afford it, there are now viable alternatives to explore higher-level descriptions for hardware accelerators.

The automatic synthesis of hardware from imperative programs – a process known as high-level synthesis (HLS) – has seen a rise in adoption in both industry [25, 59–62] and academia [9, 10, 26, 33, 34]. For an ever-growing set of families of algorithms, the fast development speed and the decreased cost of design and verification make HLS tools viable for producing good-quality hardware. In another common scenario, the value of HLS tools resides in its ability to perform quick prototype design exploration, for example when standardization committees need to quickly acquire confidence that suggested updates to an upcoming standard are likely efficiently hardware-implementable [54].

To deliver high-quality hardware designs for increasingly general purpose programs (with less regular and obvious parallelism), HLS compilers have to become more aggressive

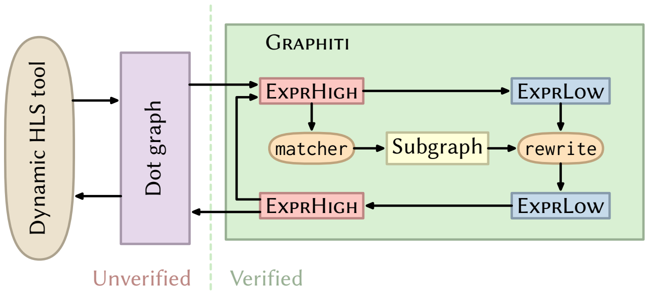


Figure 1. Tool flow, starting with a dot graph input and running rewrites by matching a subgraph on a standard graph representation we call ExprHigh, followed by performing the rewrite on an inductively defined graph representation called ExprLow.

in their optimization strategies, pushing beyond the original standard parallelization-extraction techniques required for the workloads. While most of the industrial tools have been using so-called static scheduling to transform regular C loop nests into statically scheduled circuits, for more irregular benchmarks, dynamically scheduled HLS (also known as dynamic HLS) has started to show promise.

Typically, dynamic HLS tools extract a dataflow graph from the source C program, and map this graph to a corresponding latency-insensitive circuit [8, 19, 37], which we call a *dataflow circuit*. Multiple research efforts have proposed various compilation strategies to improve the quality of the dataflow circuits obtained from dynamic HLS [6, 14, 23, 27, 37, 40, 44, 47, 68]. These strategies span a wide range of optimizations, from queue sizing and placement, to more general transformations, like identifying shareable modules. In particular, there have been recent proposals in the HLS community demonstrating promising performance benefits by introducing fine-grained out-of-order execution for particular program segments [15, 22]. This departure from strictly ordered execution represents an interesting advancement in HLS optimization. However, once operations can be reordered, ensuring correctness becomes substantially more challenging (and interesting!). Optimizations can easily become unsound (as we will see in section 6) when assumptions and guarantees are not clearly stated.

We propose to see such dataflow circuit optimizations as graph transformations, which we call dataflow graph rewrites by analogy to traditional optimizing software compilers that are performing term rewrites. Just as defining the correctness of term rewrites requires formal language semantics and a notion of program equivalence, so does proving the correctness of dataflow graph rewrites also require a formal semantics and a corresponding notion of equivalence (or, more precisely, simulation). In particular, traditional Kahnian-semantics [42, 43] are insufficient to express and study the correctness of some rewrites (section 7) due to out-of-order execution requiring *local nondeterminism*.

More concretely, reasoning about out-of-order execution is a core problem in software and hardware design. Debugging out-of-order bugs is challenging and time-consuming due to the many more subtle emerging behaviors [69]. We tackle these challenges by developing Graphiti [30], a Lean 4 [20] framework designed to ensure the correctness of dynamic HLS optimizations. The framework allows users to define, formally prove, and apply dataflow circuit rewrites to optimize dynamic HLS circuits. We demonstrate Graphiti's expressivity and practicality by implementing and formalizing a recently proposed dynamic HLS optimization. Specifically, we define a set of rewrites that introduce out-of-order execution into a dataflow circuit and formally prove that the transformation is correct, resulting in the expected notable performance gains. Additionally, we discovered that the original implementation of this optimization incorrectly optimized one of the reported benchmarks, described further in section 6. An overview of Graphiti is shown in figure 1, it accepts dataflow circuits, rewrites them using a verified rewriting framework, and can either continue iterating or output the dataflow circuit. Our key contributions are:

- A *verified rewriting framework for dataflow circuits*, Graphiti, that provides a sound foundation for defining, verifying and applying dataflow circuit optimizations, described in section 4.
- A proof of correctness in Graphiti for a core rewrite which is part of a novel graph-rewriting-based optimization that introduces out-of-order execution to a sequential dataflow circuits, described in section 3.
- An evaluation of our optimization introducing out-of-order execution compared to the state-of-the-art dynamic HLS tool called Dynamatic [39] and with the *unverified* approach that also introduces out-of-order execution [22]. We also compare with an existing *verified* statically scheduled high-level synthesis tool called Vericert [31, 32]. We demonstrate that formal verification can be integrated into a practical HLS compilation flow and can be competitive with state-of-the-art unverified flows.

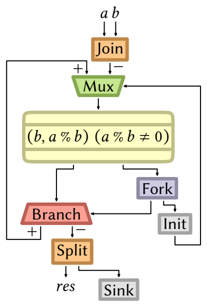
2 Out-of-order execution in dataflow circuits

In this section, we present a simple example that serves two purposes: we informally introduce dataflow circuits and their semantics and we demonstrate the advantage of out-of-order execution in dataflow circuits generated by dynamic HLS.

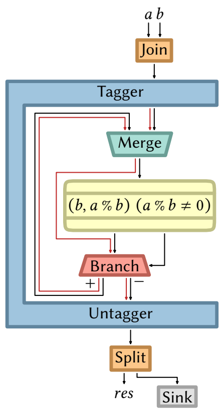
Figure 2a shows an inlined implementation of a GCD computation, where the outer loop iterates over two arrays and computes their GCD. A dataflow circuit implementing the inner `do-while` loop which computes the GCD is shown in figure 2b. This dataflow circuit is the type of circuit that

```
for(int i = 0; i < N; i++) {
  a = arr1[i];
  b = arr2[i];
  do {
    int temp = b;
    b = a % b;
    a = temp;
  } while (b != 0);
  result[i] = a;
}
```

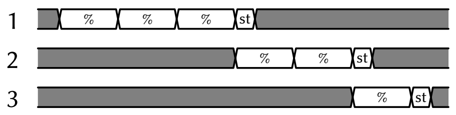
(a) Inlined implementation of a GCD calculation.



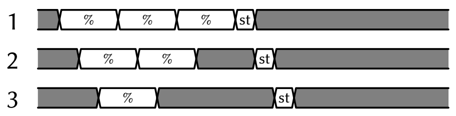
(b) Circuit implementing the inlined GCD function, following standard techniques of dynamic HLS [21, 37]. This circuit executes sequentially, i.e., it cannot accept new inputs a and b before storing the current result.



(c) Circuit implementing the inlined GCD function, following the technique of a recent work [22] that introduces tagged out-of-order execution to dataflow circuits.



(d) Execution trace for three loop executions of the in-order circuit, showing that the modulo operation cannot be pipelined.



(e) Execution trace for three loop executions of the out-of-order circuit, showing that the modulo operation is pipelined.

Figure 2. An example of how out-of-order execution in dataflow circuits increases the performance of the resulting circuit, at a reasonable resource cost.

Table 1. Description of dataflow circuit components used in dynamic HLS.

Component	Description
Branch	A Branch passes the input to the left or right if the condition is true or false respectively.
Mux	A Mux emits the left or right input if the condition is true or false respectively.
Merge	A Merge passes the first token on the left or the right to the output.
Fork	A Fork duplicates the input token.
Init	An Init component produces an initial false token and acts like a queue thereafter.
op	An arbitrary circuit component implementing an operator *op*.
Tagger / Untagger	A Tagger ensures that the outputs are emitted in the order in which they were accepted, even though they might have been reordered inside the tagger region.
Split	Splits a tuple into the left and right parts.
Join	Creates a tuple, synchronising the inputs.
Reorg	Reorganises values in a tuple according to the type signatures on the ports.
Pure *f*	Application of the pure function *f*.

would have been generated by a dynamic HLS tool [21, 37]. All components we are describing are dataflow components, also known as elastic components, and provide a latency-insensitive interface. This includes the Mux node, for example, which does not behave like a traditional combinational component. The loop is guarded by Mux and Branch components, having common conditions, at the boundaries of the loop. Together, the Muxes and Branches govern the loop initialization, loop iterations, and loop exit: every Mux takes one initialization input (i.e., from the Join in figure 2b), sends it to operators in the loop body, which passes a new value to a Branch that in turn either sends the new value out, signaling loop termination, or sends it back to the Mux signaling a new loop iteration. The yellow operation block takes in the loop input from the Mux and computes two outputs: (1) the data output for the next iteration, which includes the GCD operation, and (2) the termination condition, which is when the modulo operation returns 0. The condition gets duplicated by the Fork node, which sends the condition to the Branch as well as the Mux node. For the latter, the condition passes through an Init component, which can be thought of as a queue with an initial true token in it. Such a circuit structure results in a purely sequential implementation of the loop, which follows the sequential execution of the C program. The trace of the execution is shown in figure 2d, where each loop iteration can only start when the previous one has finished computing. Table 1 describes the other dataflow circuit components that are commonly used [37], and are present in our circuit examples.

However, assuming that the component implementing the modulo operator in the body of the inner `do-while` loop is pipelined, the sequential execution of figure 2b described above fails to make use of this pipelining opportunity because it can only process one instance of loop execution at a time. Elakhras et al. [22] showed that we can simply replace a Mux guarding a loop with a Merge that is unconditional; in doing so, they allow multiple instances of loop executions to overlap and fill the pipeline slots of the modulo component. Since the inner loop has a variable bound, we might run into a situation similar to that explained above where the output of the inner loop arrives to the store component out of the program order. To ensure that the stores are done correctly, a Tagger/Untagger is introduced. This component tags tokens it receives at the input of the Tagger, so that when the Branch sends the results to the Untagger, they can be reordered correctly. Within the Tagger/Untagger region, the circuit is responsible for correctly propagating the tag. With this new out-of-order circuit, the execution can take advantage of the pipelined modulo operation, which results in the much more efficient execution trace shown in figure 2e.

3 Graph rewriting and its application

The main goal of Graphiti is to provide an environment in which one can define and reason about rewrites on these dataflow circuits represented as graphs. In this section we will give a high-level overview of what a rewrite is, and how it is applied to a graph. We will be referencing specific subfigures that appear in later pages, any other figures can be ignored until they are described in later sections. One example of a rewrite is shown in figure 3a, which replaces two Mux nodes with a single Mux node. The rewrite is specified by a pair of graphs, with a left-hand side (lhs) and a right-hand side (rhs). This rewrite is applied by a rewriting function, together with a matching function which finds the exact spot to apply the rewrite, and replaces the lhs with the rhs. For example, we can apply this rewrite on the graph shown in figure 4a, which is an expanded version of the graph from section 2. The matching function identifies a region of the graph, highlighted with a dashed red region, that should be rewritten, and which should match the lhs of the rewrite. The rewriting function then replaces the lhs by the rhs in this graph, producing the graph in figure 4b.

The rewriting function also ensures correctness, assuming that the rewrite itself has been proven correct. What do we mean by correctness? At a high-level, we want to guarantee that the rewriting function never introduces new behaviors in the graph. Intuitively, behaviors can be thought of as traces of input/output values of a graph.

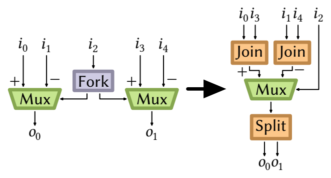
In particular, if we have a rewrite where we prove that the rhs $\sqsubseteq$ lhs, i.e. the rhs has less or the same behaviors as the lhs, then if we apply the rewrite to graph g, producing g', we guarantee that g' will also have less or the same behaviors as g ($g' \sqsubseteq g$). By chaining such rewrites, we are sure that the final optimized graph still has a subset of behaviors compared to the input graph. The general architecture of Graphiti is shown in figure 1. C++ input programs are transformed into dataflow graphs by the dynamic HLS tool front-end, which are then parsed by Graphiti into an ExprHigh graph. This is a higher-level graph-based language similar to the input dot graph language [1]. A rewrite is selected and its matcher function is run to find a subgraph to be replaced. The rewrite is applied on a lower-level graph representation called ExprLow which is more suited to verification, and the resulting graph is lifted back up to ExprHigh, where it can be output or rewritten further.

3.1 Out-of-order execution using rewrites

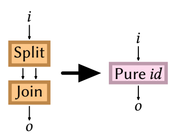
To demonstrate the practicality and effectiveness of Graphiti, we present a set of rewrites implemented in Graphiti that perform the loop optimization presented in section 2. All these rewrites are applied by the verified rewriting function, providing an overall verified transformation assuming each rewrite is verified. In total there are 20 rewrites that are needed to perform the transformation, out of those there is one core rewrite which performs the out-of-order transformation, whereas the remaining 19 are minor rewrites used to normalize the structure of the loop. We verify the core rewrite that introduces the out-of-order execution, described in detail in section 5, whereas the other minor rewrites are not verified. The rewriting strategy is independent from the correctness theorem, so an arbitrary unverified oracle can be used to guide where the rewrites are applied. We rely on Elakhras et al. to mark the loops that should be executed out-of-order. We then design an oracle to apply the rewrites in a certain order to complete the optimization, which we explain further in this section.

Figure 4 shows a complete working example that optimizes the graph from figure 2b to the graph described by figure 2c. The loop in figure 4a is composed of two Muxes and two Branches and should be made to execute out of order. There are five phases in the rewrite procedure, where most phases exhaustively apply the applicable rewrites in that phase.

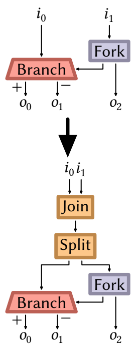
1. The loop is normalized by combining the Muxes and Branches by exhaustively applying rewrites like the one shown in figure 3a, producing the graph shown in figure 4b.
2. Additional nodes may have been introduced by these rewrites, which have to be eliminated by exhaustively applying rewrites like the ones shown in figure 3b.
3. Next, we need to prove that the loop body is acting like a pure function; if the loop body is performing side-effects, then performing them out-of-order may be incorrect. We do this using more rewrites, incrementally turning the body into a single Pure component,



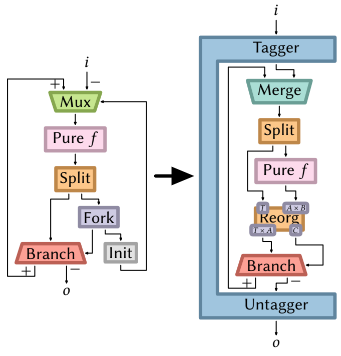
(a) Combining two Muxes into a single Mux.



(b) Reducing consecutive Split and Join into a wire.



(c) Introducing a Join that joins the input of a Fork and adding a Split right after.



(d) Out-of-order loop rewrite that overlaps the execution of the loop's iterations. The datapath in the loop is modeled by the pure function f that has a single input and output.

Figure 3. Our dataflow circuit rewrites, separated into categories according to their purpose and effect.

which is a component with one input and one output, that simply applies a function on the input. These rewrites are guided by an external oracle, described further in section 3.2.

4. We can then apply the main out-of-order loop rewrite in figure 3d, and the resulting graph is shown in figure 4d. To apply this rewrite, we also have to carefully introduce additional nodes like the Split node, using rewrites like the one shown in figure 3c.
5. Finally, the pure node has to be turned back into the graph that it is modeling. This can be simply done by applying the exact rewrites that generated it in reverse, getting back the original loop body.

This set of rewrites faithfully implements the original optimization and achieves the goal of correctly introducing out-of-order execution in the loop by allowing multiple independent loop iterations to run simultaneously and overtake each other.

3.2 Pure generation rewrites

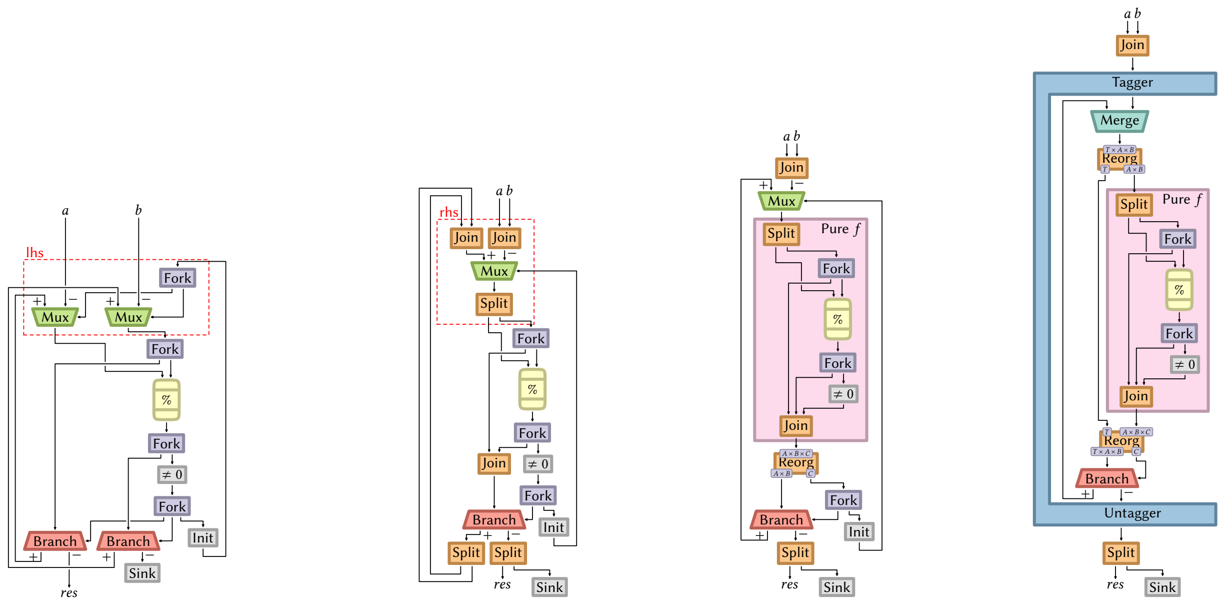
The body of the loops we are interested in rewriting could be arbitrary. However, to be able to perform the out-of-order loop rewrite, one has to prove that the loop body has two essential properties: (1) for every input, exactly one output is produced, and (2) computations are done in order. Instead of trying to prove a side-condition which states this property using analysis passes, we can reuse the rewrite engine to do so. The Pure component, which is a component that applies a function to its inputs, encapsulates precisely these properties. By showing that we can rewrite the loop body into an instance of a Pure component, we prove it has the required properties. The loop rewrite shown in figure 3d can then use a Pure component as the body of the rewrite instead of becoming a conditional rewrite on properties of the body.

The fact that Pure components can only have one input makes Pure generation, in general, nontrivial on arbitrary loop bodies, due to the presence of Forks that duplicate values, Sinks that discard values, and various operations that require multiple inputs. Taking the loop body of figure 4b as an example, we demonstrate the stages of Pure generation in figure 5. The initial loop body is shown in figure 5a. The first step is to replace each operator by a Pure implementation, adding Join nodes if the operator required more inputs. The resulting graph is shown in figure 5b. Next, Fork nodes are moved to the top of the graph, duplicating any components above the fork. This is shown in figure 5c. Next, we eliminate Fork nodes by replacing them with a Pure implementation as well, followed by a Split node, as shown in figure 5d. Finally, we move Pure components as far up and down as possible, leaving only a graph of Split and Join nodes, as shown in figure 5e.

We are left with a subgraph of Split and Join nodes that still needs to be eliminated. This should always be possible by applying the associativity, commutativity and elimination of Split and Join nodes. The order in which they need to be applied to reduce this subgraph is unknown though. For this we use egg [66] as an oracle. Egg is an e-graph rewriting tool, that allows us to identify which rewrite to perform to minimise the size of this Join and Split graph. This gives us a sequence of rewrites that can be replayed in Graphiti to eliminate this subgraph.

3.3 Main out-of-order loop rewrite

The rewrites presented above all serve as a preprocessing step towards making the main out-of-order loop rewrite applicable. This rewrite searches for a loop structure composed of a single Mux and a single Branch with a Pure function and a Split in between, as shown in figure 3d. It converts the Mux to a *tagged* Merge and inserts a Tagger/Untagger like the one in figure 2c. It is fed from the input of the Merge that comes from outside of the loop, and the output of the Branch that goes outside of the loop. Converting a Mux to a Merge introduces out-of-order execution by overlapping multiple instances of execution of the loop, and adding the Tagger/Untagger component reorders the output of the loop before passing it to the rest of the circuit, as explained in section 2.



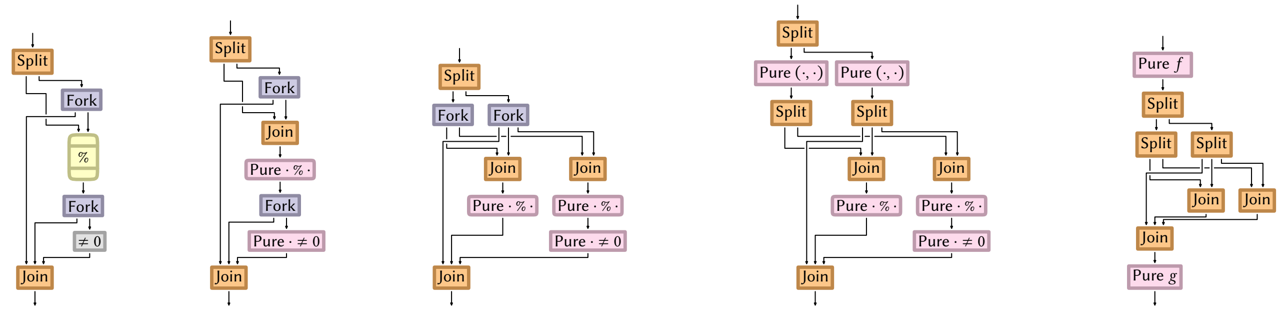
(a) Original circuit with multiple Muxes and Branches for all loop-carried variables.

(b) Join duplicate Muxes and Branches.

(c) Generate a Pure module for the body of the loop.

(d) Parallelize the loop using the main rewrite.

Figure 4. Complete example applying our rewrites on the inner loop of the simple array GCD program presented in Section 2.



(a) Initial.

(b) Op elimination.

(c) Move forks.

(d) Eliminate forks.

(e) Move pure modules.

Figure 5. Pure module generation from an arbitrary datapath. One more step is required after the last rewrite to remove all the final Split and Join components and obtain a single Pure component.

4 Semantics and refinement of graphs

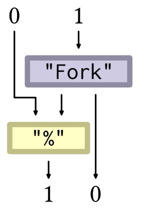
This section describes the syntax of the lower-level graph language, which we call ExprLow, as well as its semantics and how we define inclusion of behaviors in this semantics. Semantics for ExprHigh are defined in terms of ExprLow, by describing translations between these representations.

Notation. We denote a product of two types A and B as $A \times B$. Dependent sum types, describing a pair between a type T and a dependent function f from type to type, are denoted as $\sum_T f(T)$. We define $\mathcal{P}(\cdot)$ as the powerset operation. For a relation $r \subseteq A \times B$, we use the notation $r(a, b)$ to denote $(a, b) \in r$. Finally, we denote finite maps from A to B as $A \mapsto B$.

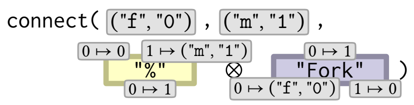
4.1 Formal definition of ExprLow syntax

One peculiarity of our graph language is that our graphs have to support inputs and outputs, which are represented as dangling wires. Due to this, port names (I) are either an I/O port identified by a single natural number (Nat), or a local name, which is identified by a pair of Str, representing an instance name paired up with a wire name which will be connected to another port in the graph.

In ExprLow the graph is defined inductively using a product ($\otimes$) constructor as well as a connect constructor. Defining a graph structure inductively instead of, for example, defining it using an adjacency matrix, is a much more suitable data structure for reasoning in an interactive theorem prover like Lean 4. The purpose for this syntax is described further



(a) ExprHigh representation of the GCD loop body.



(b) ExprLow representation of the GCD loop body.

Figure 6. Dataflow implementation of the loop body of a GCD operation, represented as both ExprHigh and ExprLow.

in section 4.5, when we describe its semantics. However, by using these global names associated with each port, we can define the same fork and modulo graph in ExprLow, shown in figure 6b. Formally, this can be described using the following grammar.

$$\begin{array}{lcll} I & \triangleq & \textsc{Nat} \mid \textsc{Str} \times \textsc{Str} & \text{(port name)} \\ P & \triangleq & (I \mapsto I) \times (I \mapsto I) & \text{(port maps)} \\ C_{\mathrm{L}} & \triangleq & P \times \textsc{Str} & \text{(ExprLow component)} \end{array}$$

Components can then be represented visually as in figure 6b. Next, ExprLow is defined inductively as follows:

$$\begin{array}{lcll} \textsc{ExprLow} & \triangleq & C_{\mathrm{L}} & \text{(base)} \\ & \mid & \textsc{ExprLow} \otimes \textsc{ExprLow} & \text{(product)} \\ & \mid & \texttt{connect}(I, I, \textsc{ExprLow}) & \text{(connect)} \end{array}$$

The base of the expression represents a component, which can be combined using the product or connect constructor.

4.2 Rewriting on ExprLow

We can then define a simple rewriting function recursively on ExprLow by relying on simple equality to find an lhs e_{lhs} in the expression and replace it by e_{rhs} in e: $e[e_{\mathsf{lhs}} := e_{\mathsf{rhs}}]$.

- $e_{\mathsf{lhs}}[e_{\mathsf{lhs}} := e_{\mathsf{rhs}}] \triangleq e_{\mathsf{rhs}}$
- $(e_1 \otimes e_2)[e_{\mathsf{lhs}} := e_{\mathsf{rhs}}] \triangleq$ $e_1[e_{\mathsf{lhs}} := e_{\mathsf{rhs}}] \otimes e_2[e_{\mathsf{lhs}} := e_{\mathsf{rhs}}]$
- $(\texttt{connect}(o, i, e))[e_{\mathsf{lhs}} := e_{\mathsf{rhs}}] \triangleq$ $\texttt{connect}(o, i, e[e_{\mathsf{lhs}} := e_{\mathsf{rhs}}])$
- $c[e_{\mathsf{lhs}} := e_{\mathsf{rhs}}] \triangleq c$

This rewriting function is simple to express and to reason about. However, it is inflexible, as subgraphs can only be matched syntactically. Instead, assuming we know which subgraph nodes we want to replace, we define ways to modify expressions locally so the subgraph in e matches e_{lhs}. For example, we prove the correctness of moving base components over products and connections, and use these transformations to isolate the subgraph that is to be replaced in a ExprLow expression. When defining the correctness theorem over ExprHigh, these transformations have to be taken into account.

4.3 Semantics of components

This section describes the semantics of ExprLow, followed by a description of the notion of refinement with which these semantics are compatible.

We call nodes in these graphs *components*. ε is a mapping from component types to semantics objects $\mathcal{M}$. $\mathcal{M}$ gives semantics to components by defining relations describing how inputs are consumed, how outputs are emitted, and potential internal transitions of the component. We name such a semantic object $\mathcal{M}$ a *module*.

For example, let's assume that we have functions called enq_n to add an element to the front of the n^{th} list (leaving other lists unchanged), deq_n to remove an element from the end of the n^{th} list (leaving other lists unchanged) and $first_n$ to check the last element of the n^{th} list. We can then define all the relations necessary to define the semantics object $\mathcal{M}$ – a module – for a fork component using the following three relations.

$$\begin{array}{lcl} \texttt{fork.in0}(l, e, l') & \triangleq & l' = enq_1(enq_2(l, e), e) \\ \texttt{fork.out0}(l, e, l') & \triangleq & e = first_1(l) \wedge l' = deq_1(l) \\ \texttt{fork.out1}(l, e, l') & \triangleq & e = first_2(l) \wedge l' = deq_2(l) \end{array}$$

Each relation is relating l, e and l', where l and l' correspond to the input and output state of the transition relation and the element e corresponds to the input for input relations and to the output for output relations. In these examples, both l and l' – corresponding to the state of the component before and after a transition – are pairs of lists. The `fork.in0` relation processes a new input e by relating the new state l' to the old state l with e enqueued to both lists in the pair. The `fork.out0` relation emits the last element in the old state l and removes it to define the new state l'.

We can also define the relation necessary to define the module for the modulo operator in a similar fashion, where the state also comprises two queues.

$$\begin{array}{lcl} \texttt{mod.in0}(l, e, l') & \triangleq & l' = enq_1(l, e) \\ \texttt{mod.in1}(l, e, l') & \triangleq & l' = enq_2(l, e) \\ \texttt{mod.out0}(l, e, l') & \triangleq & e = first_1(l) \,\%\, first_2(l) \\ & & \wedge\, l' = deq_1(deq_2(l)) \end{array}$$

Here, the modulo operation is only applied in the output transition, when both lists have at least one element.

We can then finally define the modules for the fork and modulo components, and define an environment in which we can denote circuits that refer to these components.

A module $\mathcal{M}$ is composed of: a map from identifiers to input transitions, a map from identifiers to output transitions, a collection of internal transitions which do not produce or consume an external value, and finally an initial state. For example, the modules for the fork component would be

$$\begin{array}{lcll}
\mathcal{R}_\mathrm{i}(S) & \subseteq & S \times S & \text{(internal transition)} \\
\mathcal{R}_\mathrm{e}(S,T) & \subseteq & S \times T \times S & \text{(external transition)} \\
\mathcal{M}(S) & \triangleq & (I \mapsto \sum_T \mathcal{R}_\mathrm{e}(S,T)) & \text{(module inputs)} \\
 & \times & (I \mapsto \sum_T \mathcal{R}_\mathrm{e}(S,T)) & \text{(module outputs)} \\
 & \times & \mathcal{P}(\mathcal{R}_\mathrm{i}(S)) & \text{(internal transition)} \\
 & \times & \mathcal{P}(S) & \text{(initial state)} \\
\varepsilon \in \mathsf{Env} & \triangleq & \textsc{Str} \mapsto \sum_S \mathcal{M}(S) & \text{(environment)}
\end{array}$$

Figure 7. Formal definition of a module and its environment.

written as (similarly for the mod component):

$$\begin{aligned}
\mathcal{M}_{\texttt{fork}} \triangleq\ & (\{\, 0 \mapsto \texttt{fork.in0} \,\}, \\
& \{\, 0 \mapsto \texttt{fork.out0}; 1 \mapsto \texttt{fork.out1} \,\}, \\
& \varnothing, \{\, ([], []) \,\})
\end{aligned}$$

In this case, all input and output transitions emit a value of the same type (an integer). However, we need *heterogeneous* maps and sets to store relations with different input, internal state and output types. In practice, this is done by parameterizing the input and output type by a dependent type inside of a dependent pair. We summarize in figure 7 the formal definition of a module and the environment, giving the types of the objects involved. S denotes the (dependent) type of the internal state of the component, while T denotes the (dependent) types of the input/output element of each transition. Finally, we can define an environment ε for these components: $\varepsilon \triangleq \{\, \texttt{"Fork"} \mapsto \mathcal{M}_{\texttt{fork}}; \texttt{"\%"} \mapsto \mathcal{M}_{\texttt{mod}} \,\}$.

4.4 Refinement of graphs and verified rewrites

We mentioned previously that behavior of a graph is based on traces of inputs and outputs. However, more precisely we are using a weaker notion of behavior inclusion, namely *refinement* between two modules m_1 and m_2, written as $m_1 \sqsubseteq m_2$, which states the existence of a weak simulation relation (φ) relating states in m_1 and m_2, such that a certain simulation diagram between transitions holds, detailed below. We prove that refinement implies the trace-based notion of behavior. More precisely, refinement is defined for each individual part of a module, and ensures that inputs, outputs and internal transitions refine each other.

Definition 4.1 (Input transition refinement)**.** Refinement between the input transitions of two modules m and m' for port i can be defined using the simulation diagram below: if two state s and t are related by $\varphi(s,t)$, and there is an input transition in m.in at i which transitions from s to s', then there must exist a resulting state t', an input transition at i in m'.in, and a set of internal transition executions in m'.int which transition from t to t', such that $\varphi(s',t')$. This is also depicted by the simulation diagram below.

$$m.\mathrm{in} \sqsubseteq^{\mathrm{i}}_{\varphi} m'.\mathrm{in} \triangleq \begin{tikzcd}
s \arrow[dd, "m.\mathrm{in}[i]"', "v"] \arrow[r, phantom, "\varphi"] & t \arrow[d, "v"', "m'.\mathrm{in}[i]"] \\
 & \cdot \arrow[d, dashed, "m'.\mathrm{int}", "*"'] \\
s' \arrow[r, phantom, "\varphi"] & \exists t'
\end{tikzcd}$$

Refinement for output transitions $m.\mathrm{out} \sqsubseteq^{\mathrm{o}}_{\varphi} m'.\mathrm{out}$ is defined in a similar way, except that the internal steps have to be performed *before* the output transition. The asymmetry between these definitions is due to the way connections are formed, which will be shown in section 4.5. In short, forming a connection between an output transition and an input transition removes the possibility of performing internal steps in between executing the output transition and executing the input transition. Therefore, to prove that forming connections is sound according to our definition of refinement, we forbid internal transitions from executing after output transitions and before input transitions.

Definition 4.2 (Output transition refinement)**.** Refinement between the output transitions of two modules m and m' for port o:

$$m.\mathrm{out} \sqsubseteq^{\mathrm{o}}_{\varphi} m'.\mathrm{out} \triangleq \begin{tikzcd}
s \arrow[dd, "m.\mathrm{out}[o]"', "v"] \arrow[r, phantom, "\varphi"] & t \arrow[d, dashed, "m'.\mathrm{int}", "*"'] \\
 & \cdot \arrow[d, "v"', "m'.\mathrm{out}[o]"] \\
s' \arrow[r, phantom, "\varphi"] & \exists t'
\end{tikzcd}$$

Finally, refinement of internal steps $m.\mathrm{int} \sqsubseteq^{\mathrm{int}}_{\varphi} m'.\mathrm{int}$ is defined using a similar diagram without any externally visible transitions.

Definition 4.3 (Internal transition refinement)**.** Refinement between the internal transitions of two modules m and m':

$$m.\mathrm{int} \sqsubseteq^{\mathrm{int}}_{\varphi} m'.\mathrm{int} \triangleq \begin{tikzcd}
s \arrow[d, "m.\mathrm{int}"'] \arrow[r, phantom, "\varphi"] & t \arrow[d, dashed, "m'.\mathrm{int}", "*"'] \\
s' \arrow[r, phantom, "\varphi"] & \exists t'
\end{tikzcd}$$

Definition 4.4 (Refinement with simulation relation φ)**.**

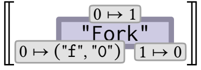
$$\begin{aligned}
m \sqsubseteq_{\varphi} m' \triangleq\ & m.\mathrm{in} \sqsubseteq^{\mathrm{i}}_{\varphi} m'.\mathrm{in} \land m.\mathrm{out} \sqsubseteq^{\mathrm{o}}_{\varphi} m'.\mathrm{out} \\
& \land m.\mathrm{int} \sqsubseteq^{\mathrm{int}}_{\varphi} m'.\mathrm{int} \\
& \land (\forall i.\, m.\mathrm{init}(i) \rightarrow \exists s.\, m'.\mathrm{init}(s) \land \varphi(i,s))
\end{aligned}$$

Definition 4.5 (Refinement)**.** The top-level definition of refinement is therefore given as: $m \sqsubseteq m' \triangleq \exists \varphi.\, m \sqsubseteq_{\varphi} m'$.

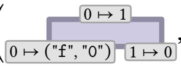
4.5 Denoting ExprLow to modules

We now want to denote the ExprLow graph definition from figure 6b into a module $\mathcal{M}$, to give a semantics to the full graph. We start with denoting base components, such as `f` and `m` in figure 6b. We denote ExprLow expression e in environment ε as $[\![e]\!]_{\varepsilon}$, producing a module object $\sum_S \mathcal{M}(S)$.

Denoting the "Fork" component from figure 6b can be done as follows, assuming we have a rename function which can rename input and output port names given a map from global to local names:



$$[\![\cdot]\!]_{\varepsilon} \triangleq \texttt{rename}\left(\cdot , \varepsilon[\texttt{"Fork"}]\right)$$



The module for the component is retrieved from the environment, and the input and output ports of the module are renamed according to the local name map for inputs and outputs included in the module definition.

Denoting a product of two ExprLow graphs is simply done by creating a module whose state is the product of the states of both ExprLow modules, and whose input, output and internal transitions are a concatenation of the transitions of each underlying module. For example, in the case where modules $\mathcal{M}_{\texttt{mod}}$ and $\mathcal{M}_{\texttt{fork}}$ are combined using a product in ExprLow, the new state that each transition needs to act on is now a product of the two lists from the $\mathcal{M}_{\texttt{mod}}$ module and the product of the two lists from the $\mathcal{M}_{\texttt{fork}}$ module. The transitions then first have to be lifted from acting on a product of two lists to act on a product of four lists. For example, the original `fork.in0` transition would be updated to the following definition, where the input is now enqueued to the third and fourth list in the state instead of the first and second.

$$\begin{aligned} \texttt{modfork.out1}(l, e, l') &\triangleq \upharpoonright\texttt{fork.out0} \\ &\triangleq l' = enq_3(enq_4(l, e), e) \end{aligned}$$

In general we can define $\upharpoonleft\cdot$ and $\upharpoonright\cdot$, which will lift a transition acting on state S to act on states $S \times S'$ and $S' \times S$ respectively, where S' is arbitrary. Merging of the transitions can then be defined using the union operator below:

$$\begin{aligned} \cdot \uplus \cdot \quad &\in \mathcal{M}(S) \to \mathcal{M}(S') \to \mathcal{M}(S \times S') \\ m_1 \uplus m_2 &\triangleq (\upharpoonleft m_1.\text{in} \cup \upharpoonright m_2.\text{in}, \\ &\quad\ \upharpoonleft m_1.\text{out} \cup \upharpoonright m_2.\text{out}, \\ &\quad\ \upharpoonleft m_1.\text{int} \cup \upharpoonright m_2.\text{int}, \\ &\quad\ \lambda\,(x, y).\, m_1.\text{init}(x) \wedge m_2.\text{init}(y)) \end{aligned}$$

This union operator is a module combinator and takes two modules and produces a new module that includes the behaviors of the two previous modules with a larger state. Therefore the denotation of the ExprLow product can be defined as: $[\![e_1 \otimes e_2]\!]_{\varepsilon} \triangleq [\![e_1]\!]_{\varepsilon} \uplus [\![e_2]\!]_{\varepsilon}$. Finally, we can also define how we denote connections in ExprLow. For example, connecting the output ("f", "0") with the input ("m", "1") can be done by removing transitions associated with these outputs and inputs and creating an internal transition that propagates the output from the first transition to the input of the next transition. The semantics associated with ("f", "0") and ("m", "1") are `modfork.out1` defined above and $\upharpoonleft$`mod.in1`. Connecting them would produce the following internal transition, which does not emit or consume data anymore, and hence does not have an e parameter:

$$\begin{aligned} &\texttt{modforkconn}(l, l') \triangleq \\ &\qquad \exists e\ l_{int}.\ \texttt{modfork.out1}(l, e, l_{int}) \wedge \upharpoonleft\texttt{mod.in1}(l_{int}, e, l') \end{aligned}$$

This can also be defined as a general combinator for modules, where the input and output transitions are removed from the compound module and are added as joined together in an internal transition r:

$$\begin{aligned} \cdot[\cdot \rightsquigarrow \cdot] \quad &\in \quad \mathcal{M}(S) \to I \to I \to \mathcal{M}(S) \\ m[o \rightsquigarrow i] \quad &\triangleq \quad (m.\text{in} - \{i\}, m.\text{out} - \{o\}, m.\text{int} + [r], m.\text{init}) \\ &\qquad \text{where} \\ &\qquad\quad r(s, s') \text{ iff } \exists v\, s''.\ m.\text{out}[o](s, v, s'') \\ &\qquad\qquad \wedge m.\text{in}[i](s'', v, s') \end{aligned}$$

Note that there are no internal transitions which are allowed to fire between executing $m.\text{out}[o](s, v, s'')$, the output transition, and $m.\text{in}[i](s'', v, s')$, the input transition, leading to the asymmetry in the definition of refinement discussed earlier.

Finally, the denotation of a connection can therefore be described as: $[\![\texttt{connect}(o, i, e))]\!]_{\varepsilon} \triangleq [\![e]\!]_{\varepsilon}[o \rightsquigarrow i]$.

4.6 Correctness of the rewriting function

We then show that some core properties hold about refinement, for example that it is a preorder, giving us transitivity and reflexivity. Additionally, we show that refinement is preserved over connections and product. These properties are used to prove correctness of all transformations on ExprLow expressions by induction on the structure of the expression. For example, the correctness of the rewriting function on an expression is verified in this way.

Theorem 4.6 (Replacement refines). *If the graph corresponding to the right-hand side of the rewrite* $\mathsf{e}_{\mathsf{rhs}}$ *refines the graph corresponding to the left-hand side of the rewrite* $\mathsf{e}_{\mathsf{lhs}}$*, then a graph where we applied the rewrite* $\mathsf{e}[\mathsf{e}_{\mathsf{lhs}} := \mathsf{e}_{\mathsf{rhs}}]$ *refines the original graph* e*.*

$$[\![\mathsf{e}_{\mathsf{rhs}}]\!]_{\varepsilon} \sqsubseteq [\![\mathsf{e}_{\mathsf{lhs}}]\!]_{\varepsilon} \ \to\ [\![\mathsf{e}[\mathsf{e}_{\mathsf{lhs}} := \mathsf{e}_{\mathsf{rhs}}]]\!]_{\varepsilon} \sqsubseteq [\![\mathsf{e}]\!]_{\varepsilon}$$

There are two versions of this theorem, one operating on ExprLow, and the top-level rewriting correctness theorem operating on ExprHigh.

5 Proof of the parametric loop rewrite

This section describes the refinement proof for a particular fairly sophisticated rewrite: the core out-of-order loop rewrite that applies to arbitrary loop bodies, which is parameterized by the function $f \in T \to T \times \text{Bool}$, and is shown in figure 3d. The function f takes values of type T and returns values of type T in addition to a Boolean, which determines if the loop should terminate or not. We want to show that the out-of-order loop execution refines a sequential loop execution. This would imply that it is safe to rewrite the sequential loop into the out-of-order loop. We will use $\mathcal{I} \in \mathcal{M}(I)$ for the module with a state I which represents the right-hand side (out-of-order) loop and $\mathcal{S} \in \mathcal{M}(S)$ for the module with state S which represents the left-hand side (sequential) loop in the rewrite shown in figure 3d.

The top-level theorem we are interested in proving is $\mathcal{I} \sqsubseteq \mathcal{S}$, which states that the out-of-order loop $\mathcal{I}$ refines the sequential loop $\mathcal{S}$. To prove this theorem, we have to find a relation $\varphi \subseteq I \times S$, such that the simulation diagrams in section 4.4 hold. The key idea in this proof is that to simulate the out-of-order loop with the sequential loop, it suffices to know that every computation in the out-of-order loop ($\mathcal{I}$) is contained in the input queue of the sequential loop ($\mathcal{S}$), i.e. the queue associate with input i in the diagram. Then, when an output transition is called on $\mathcal{I}$, we execute the entire loop with the oldest input of the input queue until it finishes iterating, followed by executing the output transition of $\mathcal{S}$.

Fortunately, we can define the output value o for an input i for both loops quite simply. We define $o = f^n(i)$ to mean applying a function f to i n times. If it returned an output value o where the associated Boolean is false, this terminates the loop with result o, and it states that input i is terminating. Of course, a loop might not be terminating if the function f never returns a value where the associated Boolean is false, meaning the loop execution would diverge. Coming back to the refinement, we can therefore prove it as long as we know that: (1) if we have an input i, such that $\exists o\ n.\, (o, \text{false}) = f^n(i)$, then $\mathcal{S}$ must terminate with value o, and (2) if we output a value o in $\mathcal{I}$, then $\exists i\ n.\, (o, \text{false}) = f^n(i)$, such that i is the *oldest* input that is currently being processed. We will therefore prove these two statements in section 5.1 and lemma 5.2 respectively.

5.1 Executing the sequential loop

We first need to show that the sequential loop actually implements the multiple application of function f correctly, i.e. that if an input terminates, then the sequential loop circuit will take input i and produce the value returned by $f^n(i)$. To do this, we define a new relation $\omega \subseteq S$ which states that all components in $\mathcal{S}$ are empty except for the input queue. We will use states $s_{\text{start}}, s_{\text{inter}}, s_{\text{end}} \in S$ in the following sections to describe states of $\mathcal{S}$.

Lemma 5.1 (Flushing invariant)**.** For any state s_{start} of $\mathcal{S}$, if the property $\omega(s_{\text{start}})$ holds and the next input i to the loop terminates, then there exists a state s_{end} that is reachable from s_{start} by applying internal transitions of $\mathcal{S}$, followed by applying an output transition which will produce the value $f^n(i)$.

$$\forall s_{\text{start}}\ i\ o\ n.\ \ \omega(s_{\text{start}}) \wedge deq(s_{\text{start}}.\text{inp}) = i \wedge o = f^n(i) \ \rightarrow$$
$$\exists s_{\text{inter}}\ s_{\text{end}}.\ \ s_{\text{start}} \xrightarrow[\mathcal{S}.\text{int}]{}^{*} s_{\text{inter}} \wedge s_{\text{inter}} \xrightarrow[\mathcal{S}.\text{out}]{o} s_{\text{end}} \wedge \omega(s_{\text{end}})$$

Proof sketch. We generalize the statement, and instead of considering i, we consider $f^m(i)$, where that value is already in the loop. The proof is then performed by induction on the number of loop iterations m, such that $m < n$, and remembering that the current value in the loop is $f^m(i)$. For the inductive case:

- If $m + 1 < n$, we execute a single loop iteration symbolically by applying internal rules of $\mathcal{S}$ until the value $f^{m+1}(i)$ is back at the top of the loop and then apply the inductive hypothesis.
- If $m + 1 = n$, then we know that function f returned a Boolean false, and we can terminate the loop.

□

5.2 Output corresponds to oldest input in out-of-order loop

Now that we know that the sequential loop produces the values that we expect, we need to show that the out-of-order loop maintains certain properties during its execution so that we can link the state of $\mathcal{I}$ with the state of $\mathcal{S}$. In the following we will use $i_{\text{start}}, i_{\text{end}} \in I$.

We define a relation $\psi \subseteq I$ comprising three main aspects of the state of module $\mathcal{I}$. *No-duplication* guarantees that for each input i the tag that $\mathcal{I}$ assigns to it is unique, and that each computed value associated with i appears at most once in the entire state I. Next, we define another property called *in-order*, which maintains that the tags which were allocated in the Tagger/Untagger component remain in the order in which they were accepted into the loop, and that the tags are always linked with the correct value. Next, we define $\theta \in I$ as being the state associate with the circuit inside of the tagger and untagger region of $\mathcal{I}$. Thus, each value can be either be an input i, be in the loop body θ or be an output o. Finally, the *iterate* property tracks how many iterations n of f are needed until the final output value o is produced (or does not exist if the input diverges). For example, if a value $v \in \theta$ associated with $f^{n-1}(i)$ is in the Merge component, when $f(v)$ will reach the Split component, $f(v)$ can be untagged, because $f^n(i) = (o, \text{false})$, becoming a final output o.

Lemma 5.2 (State invariant)**.** For all possible states i_{start} of $\mathcal{I}$ and all possible single steps $i_{\text{start}} \xrightarrow[\mathcal{I}.\text{int}]{} i_{\text{end}}$ from i_{start} to i_{end} if ψ holds for i_{start}, it will also hold for i_{end}.

$$\forall i_{\text{start}}\ i_{\text{end}}.\ \ i_{\text{start}} \xrightarrow[\mathcal{I}.\text{int}]{} i_{\text{end}} \wedge \psi(i_{\text{start}}) \ \rightarrow\ \psi(i_{\text{end}})$$

Proof sketch. The proof considers all possible internal transitions of $\mathcal{I}$ and checks whether ψ is preserved, and therefore is an invariant. □

Once we have shown that ψ is preserved over internal steps, this gives the property that each value that is computing in the loop corresponds to an input that was fed to to it, and that the tags remain in order. We can now develop the properties we need to link it to the specification.

5.3 Refinement between $\mathcal{I}$ and $\mathcal{S}$

The final theorem is the refinement between $\mathcal{I}$ and $\mathcal{S}$. We first need to define $\varphi(i_{\text{start}}, s_{\text{start}}) \triangleq \psi(i_{\text{start}}) \land \omega(s_{\text{start}}) \land \text{match}(i_{\text{start}}, s_{\text{start}})$, where the match function maps all the values that are currently in i_{start} to the input queue of s_{start} in the order in which they were received. Given two states, i_{start} of $\mathcal{I}$ and s_{start} one of $\mathcal{S}$, the relation φ is satisfied if $\psi(i_{\text{start}})$ and $\omega(s_{\text{start}})$ hold, and if the values in the $\mathcal{I}$ can be mapped to the input queue in $\mathcal{S}$.

Theorem 5.3 (Refinement between $\mathcal{I}$ and $\mathcal{S}$)**.** $\mathcal{I} \sqsubseteq_{\varphi} \mathcal{S}$

Proof sketch. The proof is by induction on the step function of $\mathcal{I}$, which generates three different cases. In each case, we prove that φ is preserved.

Input transition This implies that an input i is added to the state i_{start} of $\mathcal{I}$, and it is also added to the specification.

Output transition This implies that an output o is emitted from i_{start}. For the specification, due to $\psi(i_{\text{start}})$, we know that the output satisfies $(o, \text{false}) = f^n(i)$, where i is the oldest input in the input queue of $\mathcal{S}$. We then apply lemma 5.1 to execute the specification and emit the same output in $\mathcal{S}$.

Internal transition In this case we know that ψ is preserved in $\mathcal{I}$. We also do not perform any transitions in $\mathcal{S}$, so ω is also preserved. Finally, no values were emitted, so match must also still hold.

□

6 Evaluation

In this section, we demonstrate the practical effectiveness of our rewrites presented in section 3 and show that our framework can produce verified circuits whose quality matches that of the unverified circuits produced by Elakhras et al. [22], which we will call DF-OoO. We also compare against the in-order circuits generated by Elakhras et al. [21], which we will call DF-IO in the benchmarks. Finally, we also compare against Vericert [31, 32], a verified but statically scheduling HLS tool. Vericert therefore generates drastically different kinds of circuits when compared to Graphiti, however, it being the only other verified HLS transformation makes it an interesting comparison point. In practice, dynamic HLS tools and statically scheduling HLS tools are complementary to each other. Statically scheduling HLS tools perform well with predictable memory accesses, generating more area efficient hardware, whereas dynamic HLS tools produce circuits with higher throughput at the cost of area.

6.1 Methodology

In Lean 4 we implemented the tool flow shown in figure 1, and evaluated it using the benchmarks used by DF-OoO. We took the input dot graphs from *Dynamatic* [39], an open-source C-to-dataflow circuits tool based on LLVM [64], implementing the *fast token delivery* [21] dataflow circuit generation strategy that produces untagged circuits, prior to buffer placement. We applied the rewrites of figure 3 on the input dot graphs, transforming the same loops as those transformed by DF-OoO, and exported the results to dot graphs that we passed back to Dynamatic for buffer placement and VHDL netlist generation. We used the modified version of Dynamatic's buffer placement strategy [40], as explained by Elakhras et al., to prevent deadlocks. We employed the same number of tags specified by Elakhras et al. for each benchmark, and used the rest of the flow unmodified.

We synthesized the generated VHDL netlists using Vivado [67] with a clock-period constraint of 4 ns, targeting a Kintex-7 FPGA. We simulated the designs with ModelSim [50] to obtain cycle count. We then measure the clock period (CP) and the resource usage (i.e., LUT, FF, and DSP counts) reported from Vivado after placement and routing. Note that all three tools use the same component implementations to make the results more comparable.

The benchmarks we evaluate use floating point operations, which Vericert does not support. Therefore, to evaluate Vericert we extended the scheduler and the back-end with floating-point operation support, using the same floating point units as the other tools. Furthermore, during the development of Graphiti, we found better timing characteristics for tagging components, improving the clock period for most benchmarks. To make the comparison more fair, we manually annotated the original DF-OoO dot graphs with the same timing characteristics.

About the benchmarks. We use the same benchmarks as those which were evaluated by Elakhras et al., because these benchmarks exhibit properties that make them especially difficult to synthesise optimally using traditional HLS techniques. Firstly, there are benchmarks that inherently have a high initiation interval (II) for the inner loop due to long-latency loop-carried dependencies, whereas the outer loop has independent iterations. Most of these benchmarks come from the PolyBench/C suite [53] (**bicg**, **mvt** and **gemm**). Another benchmark is a floating-point matrix-vector multiplication (**matvec**). Next, there are benchmarks with multiple conditional paths within the inner loop which also limit the II of the loop. These comprise two versions of **gsum** [12], one inherently sequential version in **gsum-single** with loop-carried dependencies in the outer loop, and one where there are multiple independent invocations of the **gsum** kernel in **gsum-many**. Finally, **img-avg** is a benchmark that we omit in this evaluation, as the out-of-order optimization being performed for this benchmark is by reordering branch body executions instead of loop iterations, which is not a rewrite that we implemented.

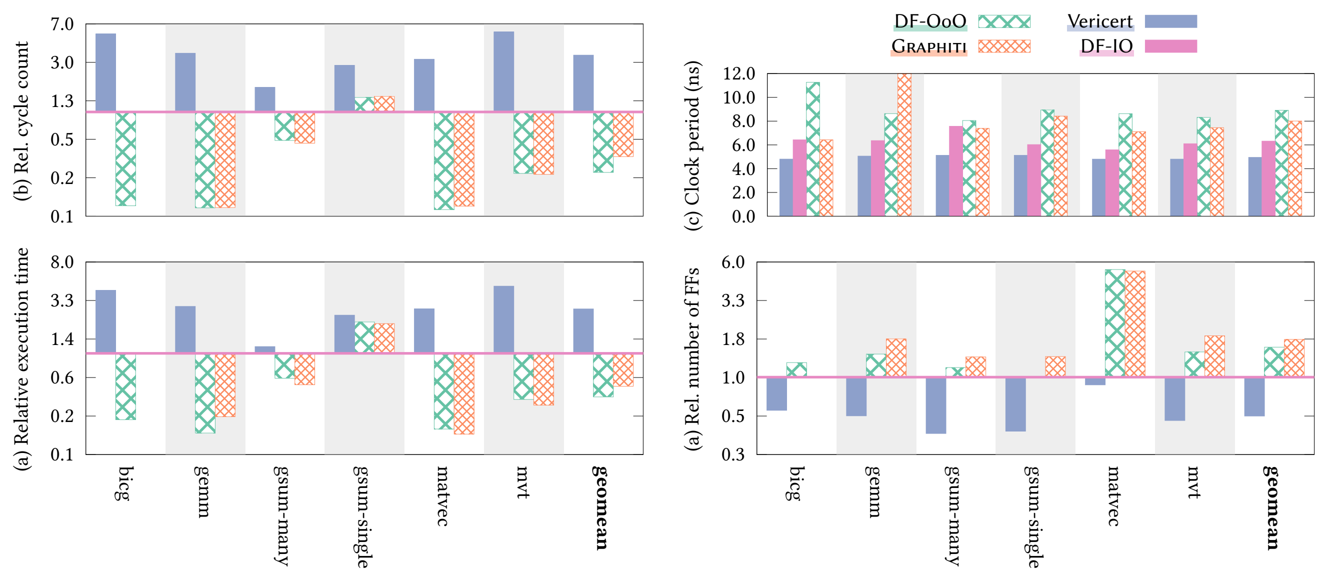


Figure 8. Graphs showing the relative performance of Graphiti, as well as prior work [21] which we will call DF-IO compared to prior work [22] which we will call DF-OoO.

6.2 Results of our transformed dataflow circuits

Figure 8 records our results (Graphiti) compared to the reproduced results of DF-IO [21], updated results of DF-OoO [22] and Vericert [31, 32]. More detailed results are shown in tables 2 and 3. We show the effectiveness of our rewrites by calculating the factor of reduction that Graphiti circuits have on the execution time compared to the untagged circuits, and show how it differs from DF-OoO.

Compared to the untagged circuits, both Graphiti circuits and the ones produced by DF-OoO report factors of improvement in the cycle count, and consequently in the execution time; this is due to an increase in loop throughput by allowing multiple outer loop iterations to execute simultaneously. In some cases, our circuits report a slightly higher cycle count than DF-OoO; this is due to our rewrites imposing extra synchronization as a result of rewrites which combine Mux and Branchcomponents, synchronizing their data paths. This synchronization is not inherently necessary for correctness, however, a more general, asynchronous, Tagger/Untagger component would be needed to lift this limitation. The approach by DF-OoO is less synchronized as it leaves the Muxes and Branches uncombined, and only synchronizes the different Muxes by providing common conditions. Nevertheless, we still have significant improvements in comparison to the untagged circuits. The exception is **gsum-single** which does not benefit from this optimization, so the result is expected. Vericert, as expected due to the irregular loops present in benchmarks, has worse cycle count throughout compared to DF-IO, but the designs can achieve a better max clock period. The resulting execution time of Graphiti is around 5.8× that of Vericert. One exception is **gsum-many**, where Vericert is on par with DF-IO, however, this is due to the data provided by the testbench for the example, which happens to fit the static schedule well. Changing the data will affect Vericert a lot more than the dataflow circuits.

Our circuits and the ones produced by DF-OoO consistently increase resources (look-up tables (LUTs) and flip-flops (FFs)) and worsen the critical path when compared to DF-IO due to the additional components managing the tagging and reordering, and the increased buffer slots to accommodate the increased parallelism, as noted in the discussion of results of DF-OoO. This is especially obvious for **matvec**, where tagged circuits use nearly 6× the number of FFs, due to the allocation of 50 tags. Here Vericert is the clear winner, with consistently better area and critical path. This is due to the fine-grained resource sharing it can perform, using less floating-point units, and also not needing handshaking circuitry due to the static schedule. In general, it's quite positive that even with additional synchronization in the circuits produced by Graphiti, this does not seem to affect critical path or area compared to DF-OoO.

Overall, our framework produces circuits that introduce out-of-order loop executions to untagged dataflow circuits using a verified loop rewrite, as-well-as a verified rewriting engine, in addition to simpler but yet unverified rewrites. It achieves a significant improvement in the circuit's execution time, in doing so. Besides, it achieves a comparable quality to that of the unverified circuits.

Dynamatic bug. For the **bicg** benchmark Graphiti performs the same as DF-IO, instead of DF-OoO. When formalising the rewrites, we could not justify doing the transformation for the **bicg** benchmark, as it had a store operation in the loop body, which led to an inconsistent memory state. This allowed us to *identify a bug in the original compilation scheme* that was turning loops out-of-order too aggressively.

6.3 Lean 4 development

In total, the size of the development is 15806 lines of Lean 4 code, taking around one person-year of time to write. However, only 1600 of those lines were needed to verify the loop rewrite. As the rewriting algorithm is written in Lean 4, it can be extracted to C, producing a command-line program that interfaces with the Dynamatic dot graph format. The rewriting algorithm is practical and can handle a thousand rewrites on graphs with a couple of hundred nodes in a reasonable amount of time. For example, optimizing **matvec** (90 nodes) required 1650 rewrites and took 9.76s, while the largest graph, **gemm** (180 nodes), required 4416 rewrites and took 81.49s. The main bottleneck is the sheer number of rewrites that are applied, which are mainly present to produce the Pure component.

One particular difficulty we encountered is linking a parametric loop rewrite refinement shown in section 5 with the concrete rewriting function correctness proof shown in theorem 4.6. This is because the environment in which this loop rewrite is interpreted is parameterized, i.e. when verifying the loop rewrite we are verifying it relative to a family of environments parameterized by f and the data type T. When applying this parameterized rewrite to a concrete input graph with a concrete environment, the rewrite needs to be specialized based on the subgraph that will be rewritten. The main issue is that this concrete environment cannot be equivalent to the parameterized environment used to prove the loop rewrite correct. For example, a `"Fork"` component denoted in the concrete environment for a specific graph can only have one specific interpretation, i.e. the denoted $\mathcal{M}_{\texttt{fork}}$ may refer to a fork of Boolean values, but cannot at the same time implement a fork for integers. In the rewrite, however, the environment containing the $\mathcal{M}_{\texttt{fork}}$ module can be parameterized by an arbitrary type T, making it possible for the `"Fork"` component to denote a module which forks values of any type T.

To unify these two types of environments, we move this parametricity into the environment. We do not expose the internal type on which the module associated with the `"Fork"` component operates. This means that the module existentially quantifies over the type of its state when it is defined in the environment. For example, when denoting this `"Fork"` component in this environment, one therefore cannot assume anything about the type of its state, or the type of the input and output ports. To be able to meaningfully reason about the loop rewrite, one does have to be able to know the type of each component though. We solve this by adding a notion of well typed graphs, which simply states that connections have to have the same type. This allows us to deduce the types of the whole graph with respects to the new, more general environment that was introduced.

Limitations. To demonstrate the usefulness and expressivity of the framework, we have verified the loop out-of-order rewrite described in section 5, as well as the rewrite function itself. However, we have not provided a proof of refinement for most of the minor rewrites, like those shown in figures 3a to 3c.

7 Related Work

Formalizations of dataflow circuits in theorem provers. Recent work [45] presented Cigr and Cilan, two languages formalized in Rocq, to reason about dataflow graphs. They have extensively validated their semantics against Dynamatic and have proven interesting meta-properties about their semantics. These meta-properties focus on determinate circuits or on safe use of nondeterministic components. Cigr and Cilan have small-step operational semantics, which are well-adapted to providing precise semantics, however, it is unclear whether the semantics exhibits enough modularity to implement a rewriting engine. Additionally, the work does not yet present a notion of equivalence between two graphs, which would be necessary to reason about transformations.

Vélus [4] is a verified compiler for Lustre [28], a synchronous dataflow language. Additionally, Paulin-Mohring [52] formalized a model of Kahn process networks [42, 43]. The main limitation of these two works are that they only model deterministic systems, so cannot reason about local nondeterminism and out-of-order execution.

Dataflow circuit optimizations. Many research efforts explored generating dataflow circuits from imperative code [6, 27, 37, 44]; this includes Dynamatic [21, 37], the source of the dataflow circuits that we used in our evaluation. Prior work aims to increase the parallelism of the circuit through queue sizing [40, 56], building memory interfaces for irregular parallelism [7, 23, 36], advancing computations via speculation [38], and increasing spatial parallelism between independent circuit constructs [14, 21, 47, 68].

Previous work [29, 49] employed loop-specific compiler transformations to execute inner loops of a loop nest in parallel and reorder at the loop exit. Elakhras et al. [22] presented a methodology to enable out-of-order execution in generic dataflow circuit structures, including loops, and its loop out-of-order transformation inspired our rewrites. Yet, none of these works formally verify the correctness of their transformations or employ a rewrite-based approach.

Verification of dataflow circuit optimizations. Other work on loop-specific out-of-order execution optimizations implement a checker for this transformation in Boogie [46], an intermediate verification language [13, 15]. The program

Table 2. Cycle count, clock period and execution time results.

	Cycle count				Clock period (ns)				Execution time (ns)			
benchmark	DF-IO	DF-OoO	Graphiti	Vericert	DF-IO	DF-OoO	Graphiti	Vericert	DF-IO	DF-OoO	Graphiti	Vericert
bicg	7936	1000	7936	44557	6.43	11.27	6.43	4.807	51028	11270	51028	214185
gemm	68825	8278	8338	252013	6.361	8.631	12.439	5.059	437796	71447	103716	1274934
gsum-many	68523	36537	34363	118096	7.57	8.052	7.388	5.127	518719	294196	253874	605478
gsum-single	6703	9234	9436	18798	6.026	8.937	8.421	5.127	40392	82524	79461	96377
matvec	7936	919.	993	25447	5.589	8.628	7.114	4.805	44354	7929	7064	122273
mvt	7940	2044	2002	46538	6.101	8.31	7.45	4.805	48442	16986	14915	223615
geomean	15842	4168	5911	55593	6.32	8.91	8.01	4.95	100095	37160	47335	275336

Table 3. Results relating to the area.

	LUT count				FF count				DSP count			
benchmark	DF-IO	DF-OoO	Graphiti	Vericert	DF-IO	DF-OoO	Graphiti	Vericert	DF-IO	DF-OoO	Graphiti	Vericert
bicg	2051	3229	2051	838	2182	2737	2182	1302	10	10	10	5
gemm	3248	5564	6282	940	2709	3880	4908	1484	11	11	11	5
gsum-many	3028	3867	4438	1151	3319	3855	4546	1381	22	22	22	5
gsum-single	2648	2541	3862	1042	3110	3101	4283	1342	22	22	22	5
matvec	1400	6027	6107	613	1282	6839	6680	1137	5	5	5	5
mvt	2980	5084	5656	936	2721	4028	5179	1386	10	10	10	5
geomean	2462	4190	4437	903	2443	3896	4396	1334	11.77	11.77	11.77	5

is translated into a Boogie representation which exhibits the same memory operations as the original program, and it can be used to find memory conflicts at a certain depth for out-of-order execution. However, the approach is specialized to this optimization and the resulting circuit is not modeled or verified to be equivalent to the input circuit.

Recent work [24] presented ElasticMiter, a framework for formally verifying the equivalence of dataflow circuits using model checking [5, 18, 58], and used it to formally prove the correctness of a set of rewrites that optimize the steering logic in dataflow circuits. Yet, their approach is confined to in-order execution and does not support tagging.

Finally, FlowCert [48] is a framework for translation validation between LLVM and the RipTide dataflow language which targets a dataflow CGRA architecture. Again, this is limited to in order dataflow circuits.

Verification of rule-based hardware languages. Verification efforts on rule-based hardware languages like Bluespec [51] have explored notions of refinement reminiscent of our own. These languages, while heavily influenced by dataflow literature, depart from input/output connections by using guarded atomic actions to compose an arbitrary number of hardware methods simultaneously. Related work [3, 17, 65] has proposed Rocq-mechanized semantics for this type of languages, and defined notions of refinement (forms of weak simulation) demonstrated to be useful by verifying specific designs like processors, accelerators or cache designs. These frameworks have not been used to describe or verify the kind of compiler-level transformations we studied.

Verified rewrite frameworks. Lean-MLIR [2] is also a rewrite framework, but focuses on software and peephole rewrites which can be interpreted using arithmetic formulas.

8 Conclusions

In conclusion, we have developed Graphiti, a rewriting framework for dataflow circuits formalized in Lean 4 and general enough to reason about transformations introducing out-of-order execution. We integrate the framework with Dynamatic, an existing state-of-the-art dynamic HLS tool, and show that we can optimize dataflow graphs in practice on existing benchmarks. Graphiti also provides an environment to verify new rewrites, which can then be plugged into the top-level rewriting loop to produce a verified top-level transformation. As rewrite based optimizations are becoming more popular [11], we hope to develop Graphiti further to reason about more tricky transformations in existing tools.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd Konstantinos Kallas for their helpful feedback. We would also like to thank Gurvan Debaussart, Hamza Remmal, John Wickerson and Tz-Ching Yu, for comments on earlier drafts. Thomas Bourgeat and Yann Herklotz were partially supported by the Swiss State Secretariat for Education, Research, and Innovation (SERI) through the SwissChips research project. Ayatallah Elakhras is supported by a Google PhD Fellowship in Distributed Systems and Parallel Computing.

A Artifact appendix

A.1 Abstract

The main contribution of the paper is a verified rewriting algorithm based on an abstract graph semantics, and an implementation of the out-of-order optimization, together with a verified loop rewrite. This artifact describes and documents the Graphiti artifact, and show that the rewriting algorithm, as well as the loop rewrite, have been fully verified without admitting any additional axioms in the Lean 4 theorem prover. Additionally, the artifact reproduces the results in the evaluation section.

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

- **Compilation:** Lean v4.26.0-rc2, rustc 1.86.0 and uv 0.7.13
- **Run-time environment:** Docker image for experiments
- **How much disk space required (approximately)?:** 45GB for docker image
- **How much time is needed to prepare workflow (approximately)?:** 30 mins
- **How much time is needed to complete experiments (approximately)?:** 1h30
- **Publicly available?:** yes
- **Code licenses (if publicly available)?:** Apache-2.0
- **Archived (provide DOI)?:** 10.5281/zenodo.18328388 [30]

A.3 Description

A.3.1 How to access. The code for the Graphiti rewriting framework, as well as the loop rewrite proof can be found on GitHub:

github.com/VCA-EPFL/graphiti/releases/tag/ASPLOS'26

A build of Graphiti, with a version of Dynamatic, is also available as a Docker image:

https://zenodo.org/records/18328388

A.3.2 Software dependencies.

- Python 3.12
- docker
- Gurobi 13.0 (https://www.gurobi.com/)
- Vivado 2019.1 (or a more recent version)
- gnuplot

A.4 Installation

Download the Zenodo archive at https://zenodo.org/records/18328388. Follow the `README.pdf` in the archive after having installed the software dependencies described in appendix A.3.2.

A.5 Evaluation and expected results

To reproduce the main results, assuming the software dependencies are installed and the artifact has been downloaded:

```
$ bash run-all.sh
```

Follow the `README.pdf` for more detailed instructions related to the artifact.

References

[1] The Graphviz Authors. [n. d.]. DOT Language. https://graphviz.org/doc/info/lang.html

[2] Siddharth Bhat, Alex Keizer, Chris Hughes, Andrés Goens, and Tobias Grosser. 2024. Verifying Peephole Rewriting in SSA Compiler IRs. In *15th International Conference on Interactive Theorem Proving (ITP 2024) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 309)*, Yves Bertot, Temur Kutsia, and Michael Norrish (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 9:1–9:20. doi:10.4230/LIPIcs.ITP.2024.9

[3] Thomas Bourgeat, Jiazheng Liu, Adam Chlipala, and Arvind. 2025. Making Concurrent Hardware Verification Sequential. *PLDI* 9, Proc. ACM Program. Lang. (2025), 2007–2031. doi:10.1145/3729331

[4] Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. 2017. A Formally Verified Compiler for Lustre. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 586–601. doi:10.1145/3062341.3062358

[5] 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. doi:10.1007/978-3-642-18275-4_7

[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. doi:10.1109/ISPASS.2005.1430572

[7] Mihai Budiu and Seth Copen Goldstein. 2003. Optimizing Memory Accesses for Spatial Computation. In *Proceedings of the 1st International ACM/IEEE Symposium on Code Generation and Optimization.* San Francisco, CA, 216–27. doi:10.1109/CGO.2003.1191547

[8] L.P. Carloni, K.L. McMillan, and A.L. Sangiovanni-Vincentelli. 2001. Theory of latency-insensitive design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 20, 9 (2001), 1059–1076. doi:10.1109/43.945302

[9] David Castells-Rufas, Santiago Marco-Sola, Juan Carlos Moure, Quim Aguado, and Antonio Espinosa. 2022. FPGA Acceleration of Pre-Alignment Filters for Short Read Mapping With HLS. *IEEE Access* 10 (2022), 22079–22100. doi:10.1109/ACCESS.2022.3153032

[10] Hongzheng Chen, Niansong Zhang, Shaojie Xiang, Zhichen Zeng, Mengjia Dai, and Zhiru Zhang. 2024. Allo: A Programming Model for Composable Accelerator Design. *Proc. ACM Program. Lang.* 8, PLDI, Article 171 (June 2024), 28 pages. doi:10.1145/3656401

[11] Jianyi Cheng, Samuel Coward, Lorenzo Chelini, Rafael Barbalho, and Theo Drane. 2024. SEER: Super-Optimization Explorer for High-Level Synthesis using E-graph Rewriting. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 1029–1044. doi:10.1145/3620665.3640392

[12] 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. doi:10.1145/3373087.3375297

[13] Jianyi Cheng, Lana Josipović, John Wickerson, and George A. Constantinides. 2023. Parallelising Control Flow in Dynamic-Scheduling High-Level Synthesis. *ACM Transactions on Reconfigurable Technology and Systems* 16, 4 (2023), 55:1–55:32. doi:10.1145/3599973

[14] Jianyi Cheng, Lana Josipović, George A. Constantinides, and John Wickerson. 2022. Dynamic Inter-Block Scheduling for HLS. In *32nd International Conference on Field-Programmable Logic and Applications (FPL)*. 243–252. doi:10.1109/FPL57034.2022.00045

[15] Jianyi Cheng, John Wickerson, and George A. Constantinides. 2022. Dynamic C-Slow Pipelining for HLS. In *IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 1–10. doi:10.1109/FCCM53951.2022.9786096

[16] Monica Chiosa, Fabio Maschi, Ingo Müller, Gustavo Alonso, and Norman May. 2022. Hardware acceleration of compression and encryption in SAP HANA. *Proc. VLDB Endow.* 15, 12 (Aug. 2022), 3277–3291. doi:10.14778/3554821.3554822

[17] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. 2017. Kami: a platform for high-level parametric hardware specification and its modular verification. *Proc. ACM Program. Lang.* 1, ICFP (2017), 24:1–24:30. doi:10.1145/3110268

[18] 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. doi:10.1007/3-540-61474-5_93

[19] Jordi Cortadella, Mike Kishinevsky, and Bill Grundmann. 2006. Synthesis of synchronous elastic architectures. In *Proceedings of the 43rd Annual Design Automation Conference* (San Francisco, CA, USA) *(DAC '06)*. Association for Computing Machinery, New York, NY, USA, 657–662. doi:10.1145/1146909.1147077

[20] Leonardo de Moura, Sebastian Ullrich, and AMD. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28*, André Platzer and Geoff Sutcliffe (Eds.). Springer International Publishing, Cham, 625–635. doi:10.1007/978-3-030-79876-5_37

[21] 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. doi:10.1109/FPL57034.2022.00046

[22] 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, USA) *(FPGA '24)*. Association for Computing Machinery, New York, NY, USA, 44–54. doi:10.1145/3626202.3637556

[23] 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 ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, CA, USA) *(FPGA '23)*. Association for Computing Machinery, New York, NY, USA, 39–45. doi:10.1145/3543622.3573050

[24] Ayatallah Elakhras, Jiahui Xu, Martin Erhart, Paolo Ienne, and Lana Josipović. 2025. *ElasticMiter: Formally Verified Dataflow Circuit Rewrites*. Association for Computing Machinery, New York, NY, USA, 293–308. https://doi.org/10.1145/3676641.3715993

[25] Google. 2020. XLS: Accelerated HW Synthesis. https://google.github.io/xls/

[26] Yijin Guan, Hao Liang, Ningyi Xu, Wenqiang Wang, Shaoshuai Shi, Xi Chen, Guangyu Sun, Wei Zhang, and Jason Cong. 2017. FP-DNN: An Automated Framework for Mapping Deep Neural Networks onto FPGAs with RTL-HLS Hybrid Templates. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 152–159. doi:10.1109/FCCM.2017.25

[27] Nico Gädke-Lütjens. 2011. *Dynamic Scheduling in High-Level Compilation for Adaptive Computers*. Ph.D. Thesis. Technischen Universität Braunschweig, Braunschweig, Germany.

[28] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. 1991. The Synchronous Data Flow Programming Language LUSTRE. *Proc. IEEE* 79, 9 (1991), 1305–1320. doi:10.1109/5.97300

[29] Robert J. Halstead and Walid Najjar. 2013. Compiled Multithreaded Data Paths on FPGAs for Dynamic Workloads. In *Proceedings of the International Conference on Compilers Architectures and Synthesis for Embedded Systems*. Montreal, Canada, 3:1–3:10. doi:10.1109/CASES.2013.6662507

[30] Yann Herklotz, Ayatallah Elakhras, Martina Camaioni, Paolo Ienne, Lana Josipović, and Thomas Bourgeat. 2026. *Graphiti Artefact ASPLOS 2026*. doi:10.5281/zenodo.18328388

[31] 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 (10 2021). doi:10.1145/3485494

[32] Yann Herklotz and John Wickerson. 2024. Hyperblock Scheduling for Verified High-Level Synthesis. *Proceedings of the ACM on Programming Languages* 8, PLDI, Article 225 (6 2024), 25 pages. doi:10.1145/3656455

[33] Yuwei Hu, Yixiao Du, Ecenur Ustun, and Zhiru Zhang. 2021. GraphLily: Accelerating Graph Linear Algebra on HBM-Equipped FPGAs. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 1–9. doi:10.1109/ICCAD51958.2021.9643582

[34] Qijing Huang, Christopher Yarp, Sagar Karandikar, Nathan Pemberton, Benjamin Brock, Liang Ma, Guohao Dai, Robert Quitt, Krste Asanovic, and John Wawrzynek. 2019. Centrifuge: Evaluating full-system HLS-generated heterogenous-accelerator SoCs using FPGA-Acceleration. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8. doi:10.1109/ICCAD45719.2019.8942048

[35] Intel. [n. d.]. Intel QAT: Accelerating data compression and encryption. https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/what-is-intel-qat.html

[36] Lana Josipović, Philip Brisk, and Paolo Ienne. 2017. An Out-of-Order Load-Store Queue for Spatial Computing. *ACM Transactions on Embedded Computing Systems* 16, 5s (Sept. 2017), 125:1–125:19. doi:10.1145/3126525

[37] Lana Josipović, Radhika Ghosal, and Paolo Ienne. 2018. Dynamically Scheduled High-level Synthesis. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Monterey, CALIFORNIA, USA) *(FPGA '18)*. Association for Computing Machinery, New York, NY, USA, 127–136. doi:10.1145/3174243.3174264

[38] 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.

[39] Lana Josipović, Andrea Guerrieri, and Paolo Ienne. 2022. From C/C++ Code to High-Performance Dataflow Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* CAD-41, 7 (July 2022), 2142–55. doi:10.1109/TCAD.2021.3105574

[40] Lana Josipović, Shabnam Sheikhha, Andrea Guerrieri, Paolo Ienne, and Jordi Cortadella. 2021. Buffer Placement and Sizing for High-Performance Dataflow Circuits. *ACM Trans. Reconfigurable Technol. Syst.* 15, 1, Article 4 (Nov. 2021), 32 pages. doi:10.1145/3477053

[41] Norman P. Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B. Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, Thomas Norrie, Nishant Patil, Sushma Prasad, Cliff Young, Zongwei Zhou, and David Patterson. 2021. Ten Lessons From Three Generations Shaped Google's TPUv4i: Industrial Product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 1–14. doi:10.1109/ISCA52012.2021.00010

[42] G. Kahn. 1974. The Semantics of a Simple Language for Parallel Programming. In *Information Processing 74: Proceedings of the IFIP Congress 74, Stockholm, Sweden, August 1974*, J.L. Rosenfeld (Ed.). North-Holland, Amsterdam, Netherlands, 471–475.

[43] Gilles Kahn and David MacQueen. 1976. *Coroutines and Networks of Parallel Processes*. Research Report IRIA-RR-202. IRIA. 21 pages. https://inria.hal.science/hal-04716357

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

[45] Tony Law, Delphine Demange, and Sandrine Blazy. 2025. A Mechanized Semantics for Dataflow Circuits. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 98 (April 2025), 27 pages. doi:10.1145/3720432

[46] K Rustan M Leino. 2008. This is Boogie 2. *manuscript KRML* 178, 131 (2008), 9. https://audentia-gestion.fr/MICROSOFT/PDF/krml178.pdf

[47] Rui Li, Lincoln Berkley, Yihang Yang, and Rajit Manohar. 2021. Fluid: An Asynchronous High-level Synthesis Tool for Complex Program Structures. In *2021 27th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*. 1–8. doi:10.1109/ASYNC48570.2021.00009

[48] 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). doi:10.1145/3689729

[49] Gai Liu, Mingxing Tan, Steve Dai, Ritchie Zhao, and Zhiru Zhang. 2015. Architecture and Synthesis for Area-Efficient Pipelining of Irregular Loop Nests. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 36, 11 (Feb. 2015), 1817–1830. doi:10.1109/TCAD.2017.2664067

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

[51] Rishiyur S. Nikhil. 2004. Bluespec System Verilog: Efficient, Correct RTL from High Level Specifications. In *Proceedings of the Second ACM/IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE '04)*. IEEE Computer Society, Washington, DC, USA, 69–70. doi:10.1109/MEMCOD.2004.1459818

[52] Christine Paulin-Mohring. 2009. A Constructive Denotational Semantics for Kahn Networks in Coq. In *From Semantics to Computer Science*, Yves Bertot, Gérard Huet, Jean-Jacques Lévy, and Gordon Plotkin (Eds.). Cambridge University Press, 383–413. https://inria.hal.science/inria-00431806

[53] Louis-Noël Pouchet. 2012. *Polybench: The polyhedral benchmark suite*. http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/

[54] David Pursley. [n. d.]. "Great" Hardware Design in a Wireless World. https://community.cadence.com/cadence_blogs_8/b/di/posts/great-hardware-design-in-a-wireless-world

[55] Parthasarathy Ranganathan, Daniel Stodolsky, Jeff Calow, Jeremy Dorfman, Marisabel Guevara, Clinton Wills Smullen IV, Aki Kuusela, Raghu Balasubramanian, Sandeep Bhatia, Prakash Chauhan, Anna Cheung, In Suk Chong, Niranjani Dasharathi, Jia Feng, Brian Fosco, Samuel Foss, Ben Gelb, Sara J. Gwin, Yoshiaki Hase, Da-ke He, C. Richard Ho, Roy W. Huffman Jr., Elisha Indupalli, Indira Jayaram, Poonacha Kongetira, Cho Mon Kyaw, Aaron Laursen, Yuan Li, Fong Lou, Kyle A. Lucke, JP Maaninen, Ramon Macias, Maire Mahony, David Alexander Munday, Srikanth Muroor, Narayana Penukonda, Eric Perkins-Argueta, Devin Persaud, Alex Ramirez, Ville-Mikko Rautio, Yolanda Ripley, Amir Salek, Sathish Sekar, Sergey N. Sokolov, Rob Springer, Don Stark, Mercedes Tan, Mark S. Wachsler, Andrew C. Walton, David A. Wickeraad, Alvin Wijaya, and Hon Kwan Wu. 2021. Warehouse-scale video acceleration: co-design and deployment in the wild. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 600–615. doi:10.1145/3445814.3446723

[56] 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. doi:10.1109/FPL57034.2022.00063

[57] Kiran Seshadri, Berkin Akin, James Laudon, Ravi Narayanaswami, and Amir Yazdanbakhsh. 2022. An Evaluation of Edge TPU Accelerators for Convolutional Neural Networks. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*. 79–91. doi:10.1109/IISWC55918.2022.00017

[58] 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. doi:10.1007/3-540-40922-X_8

[59] Siemens. 2015. *Google Develops WebM Video Decompression Hardware IP Using High-Level Synthesis*. White Paper. Siemens EDA, Siemens Digital Industries Software. https://resources.sw.siemens.com/en-US/white-paper-google-develops-webm-video-decompression-hardware-ip-using-high-level/

[60] Siemens. 2020. *High-Level Synthesis for Autonomous Drive*. White Paper. Siemens EDA, Siemens Digital Industries Software. https://resources.sw.siemens.com/en-US/white-paper-high-level-synthesis-for-autonomous-drive/

[61] Siemens. 2022. *STMicroelectronics Quickly Brings Automotive Image Signal Processing to Market with High-Level Synthesis*. White Paper. Siemens EDA, Siemens Digital Industries Software. https://resources.sw.siemens.com/en-US/white-paper-stmicroelectronics-brings-automotive-image-signal-processing-to-market-with-high-level-synthesis/

[62] Siemens. 2025. *How NVIDIA Uses High-Level Synthesis Tools for AI Hardware Accelerator Research*. Webinar. Siemens EDA, Siemens Digital Industries Software. https://webinars.sw.siemens.com/en-US/how-nvidia-uses-high-level-synthesis-tools-for-ai-hardware-accelerator-research/

[63] Anthony T. Sofia, Matthias Klein, Brad D. Stilwell, Simon Weishaupt, Qin Yue Chen, and Robert W. St. John. 2020. Integration of z15 processor-based DEFLATE acceleration into IBM z/OS. *IBM J. Res. Dev.* 64, 5/6 (2020), 10:1–10:8. doi:10.1147/JRD.2020.3008101

[64] The LLVM Compiler Infrastructure 2018. *http://www.llvm.org*. The LLVM Compiler Infrastructure. http://www.llvm.org

[65] Muralidaran Vijayaraghavan, Adam Chlipala, Arvind, and Nirav Dave. 2015. Modular Deductive Verification of Multiprocessor Hardware Designs. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 9207)*, Daniel Kroening and Corina S. Pasareanu (Eds.). Springer, 109–127. doi:10.1007/978-3-319-21668-3_7

[66] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.* 5, POPL, Article 23 (Jan. 2021), 29 pages. doi:10.1145/3434304

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

[68] Ali Mustafa Zaidi and David Greaves. 2014. A New Dataflow Compiler IR for Accelerating Control-Intensive Code in Spatial Hardware. In *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*. 122–131. doi:10.1109/IPDPSW.2014.18

[69] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. 2020. SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine. In *Fourth Workshop on Computer Architecture Research with RISC-V (CARRV 2020)*. https://carrv.github.io/2020/papers/CARRV2020_paper_15_Zhao.pdf