Thesis Proposal: Verifying Asynchronous Dataflow Compilers

ZHENGYAO LIN, Carnegie Mellon University, USA

The paradigm of *asynchronous dataflow circuits*, in which parallel operators are dynamically scheduled and data-driven, unlocks substantial energy efficiency and performance gains in reconfigurable dataflow architectures (RDAs) and dynamic high-level synthesis (HLS) toolchains. A key challenge hindering their mainstream adoption is the difficulty of *correct*, *efficient*, and *general-purpose* compilation towards asynchronous dataflow. In this proposal, I present my efforts to apply formal verification to asynchronous dataflow, with the goal of developing an end-to-end, provably correct dataflow compilation pipeline.

1 Introduction

An asynchronous dataflow circuit consists of a set of parallel operators, each peforming a simple arithmetic, routing, or memory operation, and they are interconnected with communication channels. In recent years, asynchronous dataflow has seen a re-emergence in reconfigurable dataflow architectures (RDAs) [1, 6, 8, 10, 14, 31, 33] and dynamic high-level synthesis (HLS) [11, 15, 25]. In RDAs, the data locality of asynchronous dataflow unlocks significant improvements in energy efficiency [9, 10], while in HLS, the asynchrony (i.e., dynamic scheduling) enables better performance and improves general-purpose programmability [15]. These architectures and techniques cover a wide range of applications, from ultra-low-power edge computing [9, 10] to high-performance machine learning [31]. They have the potential to improve energy efficiency by orders of magnitude [10, 31] compared to conventional approaches, making asynchronous dataflow a promising paradigm for future accelerators.

A key common challenge in both RDAs and dynamic HLS is *dataflow compilation*, i.e., the process of turning a high-level, sequential, and imperative user program into an optimized asynchronous dataflow circuit that preserves its behavior. The exact benefits of asynchronous dataflow, namely parallelism and dynamic scheduling, become challenges in dataflow compilation. Every instruction in the original sequential program is converted into a parallel operator in the dataflow circuit that communicates with others through channels; to enable realistic applications, these operators also have access to a *shared memory*. As a result, the simple sequential source program suddenly becomes a highly distributed system with both message passing and shared memory, along with the potential concurrency bugs of deadlocks and data races, all at the hardware level!

Further complicating this challenge, dataflow also comes in different variants. One important distinction in prior architecture work is the difference between *ordered* and *tagged* dataflow. In ordered dataflow [9, 10, 31, 33], channels have the semantics of FIFO queues and the dataflow circuit has a static topology. In the more complex model of tagged dataflow [1, 3, 12, 30], in order to enable more parallelism and expressive control flow, messages can reorder in channels and the dataflow circuit has a dynamic topology.

While the asynchronous dataflow paradigm has seen significant growth in the architecture community, efforts in formalizing and verifying dataflow architectures have been sparse over the last few decades. The most notable historical work that provided theoretical underpinning for dataflow includes Kahn's process networks [16] and Lee's dataflow semantics with firing [18]. Recently, in the context of Dynamatic [15], a dynamic HLS framework, there have been efforts to mechanize the dataflow models [17] and verify dataflow graph rewrites [7, 13]. All of this prior formalization work misses two important aspects of modern dataflow architectures: (1) the compilation from sequential source programs to parallel dataflow circuits, which is the first, and arguably the most error-prone step, and (2) the shared memory/state between dataflow operators, which is a key and

Author's Contact Information: Zhengyao Lin, zhengyal@cmu.edu, Carnegie Mellon University, Pittsburgh, PA, USA.

necessary divergence [1, 10, 31] from classical denotational models of dataflow [16, 18] to support practical applications, and it is also an important source of concurrency bugs.

Therefore, this proposal develops formal compiler verification techniques for the model of *asynchronous dataflow with shared memory*, and seeks to establish the following thesis:

> A combination of translation validation and modular compiler verification techniques makes it practical to formally guarantee semantics preservation and the absence of concurrency bugs in asynchronous dataflow compilers.

As an overview of this proposal, I will discuss the following finished and ongoing work:

- Section 2: FlowCert, a translation validation approach for ordered dataflow compilation.
- Section 3: Wavelet, a formally verified dataflow compiler in the Lean theorem prover, with the goals of modular proofs and supporting the pipelining optimization for ordered dataflow.
- Section 4: Ongoing work to extend Wavelet to support tagged dataflow compilation.

Finally, I conclude with a planned timeline of my thesis work in Section 5.

2 (Completed) FlowCert: Translation Validation for Asynchronous Dataflow

As a first step towards the thesis goal, we hope understand how state-of-the-art dataflow compilers work and the challenges in verifying them without specializing our proofs for a particular compiler. Therefore, we developed FlowCert [22], which is a translation validation technique for certifying compilation from source sequential imperative programs to asynchronous dataflow circuits. In our case, translation validation amounts to checking that the output dataflow circuit from the compiler has the same behaviors as the source program. Compared to traditional testing, translation validation provides a much stronger guarantee, as it checks that the imperative program and dataflow circuit behave the same on all possible inputs. Additionally, in contrast to full compiler verification, translation validation allows the use of unverified compilation toolchains, which is crucial for the fast-developing area of RDAs.

The main challenge of this work (Section 2.1) is that our source program is a single-threaded imperative program, while the target is a highly concurrent dataflow circuit *with shared memory*. Therefore, the target dataflow has the potential to introduce unwanted behaviors (deadlocks and data races) due to concurrency.

To tackle this challenge, we use a two-phase translation validation technique (Section 2.2). We first prove the forward simulation from the source program to the dataflow circuit using symbolic execution. Then, by augmenting the symbolic execution with affine memory permissions, we show that any possible schedule of the dataflow circuit must be "equivalent" to the execution schedule explored during the forward simulation check.

In our evaluation [22], we implement this technique to target the state-of-the-art RDA RipTide [10], which has an LLVM-based compiler to dataflow circuits. Using our tool, we found 8 confirmed compiler bugs where the RipTide dataflow compiler generated incorrect dataflow circuits, including a data race bug that is hard to discover via testing.

The full presentation this work can be found in our OOPSLA paper [22], advised Joshua Gancher and Bryan Parno.

2.1 Background and Challenges

A *dataflow circuit* is a graph in which the nodes are called *operators* and edges are called *channels*. Semantically, operators can be thought of as stateful, recursive processes running in parallel, which communicate through channels, or FIFO queues of messages. Repeatedly, each operator: waits to dequeue (a subset of) input channels to get input values; performs a local computation, optionally

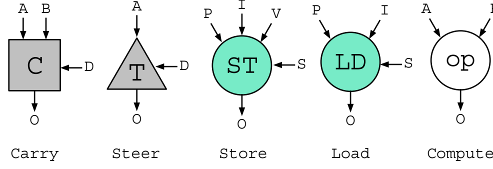
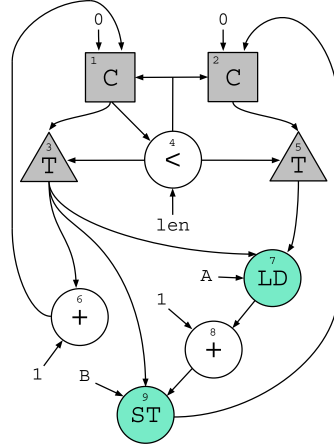


Fig. 1. Examples of operators. `Carry` and `Steer` manage control flow, `Store` and `Load` manipulate the main memory, and op ∈ {+, ∗, <, . . . } executes pure arithmetic operations.

```
1   define void @test(i32* %A, i32* %B, i32 %len) {
2   entry:
3     br label %header
4   header:
5     %i = phi i32 [ 0, %entry ], [ %i_inc, %body ]
6     %cond = icmp slt i32 %i, %len
7     br i1 %cond, label %body, label %end
8   body:
9     %idx1 = getelementptr i32, i32* %A, i32 %i
10    %A_i = load i32, i32* %idx1
11    %A_i_inc = add i32 %A_i, 1
12    %idx2 = getelementptr i32, i32* %B, i32 %i
13    store i32 %A_i_inc, i32* %idx2
14    %i_inc = add i32 %i, 1
15    br label %header
16  end:
17    ret void
18  }
```

(a) Input LLVM function.



(b) Output dataflow circuit.

Fig. 2. An example of input/output programs from the RipTide dataflow compiler. Most dataflow operators correspond to LLVM instructions: the comparison operator corresponds to line 6, the add operators correspond to lines 11 and 14, while the load and store operators correspond to lines 10 and 13, respectively.

changing its local state or global memory; and pushes output values to (a subset of) output channels. When an iteration of this loop is done, we say that the operator has executed or *fired*.

In Figure 1, we show the main operators in RipTide [10]. The carry operator (`C`) is used to support loop variables. It has two local states: in the initial state, it waits for a value from `A`, outputs it to `O`, and transitions to the loop state. In the loop state, it waits for values from `B` and `D` (decider); if `D` is true, it outputs `O` = `B`; otherwise it discards `B` and transitions to the initial state. The steer operator (`T`) is used for conditional execution. It waits for values from `A` and `D`. If `D` is true, it outputs `O` = `A`; otherwise, it discards the value and outputs nothing. To utilize the global memory, we have the *load* (`LD`) and *store* (`ST`) operators. The load operator waits for values from `P` (base), `I` (offset), and an optional signal `S` for memory ordering; reads the memory location `P[I]`; and outputs the read value to `O`. Similarly, the store operator waits for values from `P` (base), `I` (offset), `V` (value), and an optional signal `S` for memory ordering; stores `P[I]` = `V` in the global memory; and optionally outputs a finish signal to `O`. All other *compute* operators (e.g., op ∈ {+, ∗, <}) wait for values from input channels and output the result to output channels.

Compilation from LLVM to Dataflow. RipTide's dataflow compiler is an LLVM-based compiler that takes in a sequential and imperative program in LLVM IR [24] and emits a dataflow circuit for the RipTide RDA. Figure 2 shows an example of such a compilation pass.

The input LLVM function `@test` in Figure 2a contains a single loop with loop header `%header` and loop body `%body`. For each `%i` from `0` to `%len`, the function updates `%B[%i] = %A[%i] + 1`.

The LLVM function `@test` is compiled to the dataflow circuit in Figure 2b. In addition to compute and memory operators that mostly correspond to their counterpart LLVM instructions in the source program, the compiler also inserts operators to faithfully implement the sequential semantics of the LLVM program. The steer operators enforce that the operators corresponding to the loop body only execute when the loop condition is true. The carry operator 1 is used for the loop variable `%i` (line 5). The carry operator 2 is used for a loop variable inserted by the compiler to enforce a memory dependency: since arrays `A` and `B` could overlap, we have an additional dataflow path along operators $9 \to 2 \to 5 \to 7$ in order to enforce that the load in the next iteration waits until the previous store has finished.

This also illustrates the challenge of dataflow compilation. The additional data dependency to enforce that the load operator waits for the previous store operator is necessary because the array `A` can *potentially alias* with `B`. If a buggy compiler omits this dependency in the dataflow circuit, then on certain inputs, the dataflow circuit may have a data race that violates the read-after-write dependency in the original source program. On the other hand, we should not be too conservative by inserting unnecessary dependencies between any pair of memory operators. One challenge of our translation validation technique is then to formally check that the dataflow compiler has made the right decision to insert enough dependencies to prevent data races.

2.2 Our Two-Phase Validation Technique

On the same example in Figure 2, we now illustrate our translation validation technique.

Our translation validation approach consists of two steps. First, we perform a simulation check to validate that there is a *forward simulation* from the input LLVM program to the output dataflow circuit. Through the forward simulation, the execution of the input LLVM program induces a particular execution schedule of the dataflow circuit, which we call the *canonical schedule*.

Next, in a *determinacy check*, we prove that any possible schedule of the dataflow circuit must converge to the same final state as the canonical schedule, or that any schedule can synchronize with the canonical schedule infinitely often. For our setting, this requires showing that the dataflow circuit does not contain any data races. Inspired by fractional permissions [5], we augment our symbolic execution with *affine permission tokens* to track ownership of memory locations.

Combining the guarantees from these two checks, we can conclude that the dataflow circuit is equivalent to the LLVM program on all possible schedules.

Simulation Check. The first step in FlowCert's translation validation procedure is to check that the LLVM program and dataflow circuit are equivalent on a canonical schedule of dataflow operators, where the dataflow operators are executed in the same order as their LLVM instruction counterparts.

This is done using a *simulation relation* between the states of the LLVM program and dataflow circuit. FlowCert constructs this relation by placing corresponding *cut points* on both the LLVM and the dataflow sides. A cut point symbolically describes a set of pairs of LLVM and dataflow configurations satisfying a correspondence condition; a list of cut points together describes the simulation relation. Figure 4 shows the list of cut points for the example. FlowCert places cut point 1 for the initial configurations, cut point 2 for the loop structure, and a final cut point $\bot$ for all final configurations. For each pair of LLVM and dataflow cut points, FlowCert also infers a list of equations expressing the correspondence of symbolic variables in these cut points.

To automatically check that the proposed relation in Figure 4 is indeed a simulation, we perform *symbolic execution* [4] from all pairs of cut points (except for $\bot$) until they reach another pair of cut points. On the LLVM side, we first symbolically execute cut point 1, which turns into two symbolic

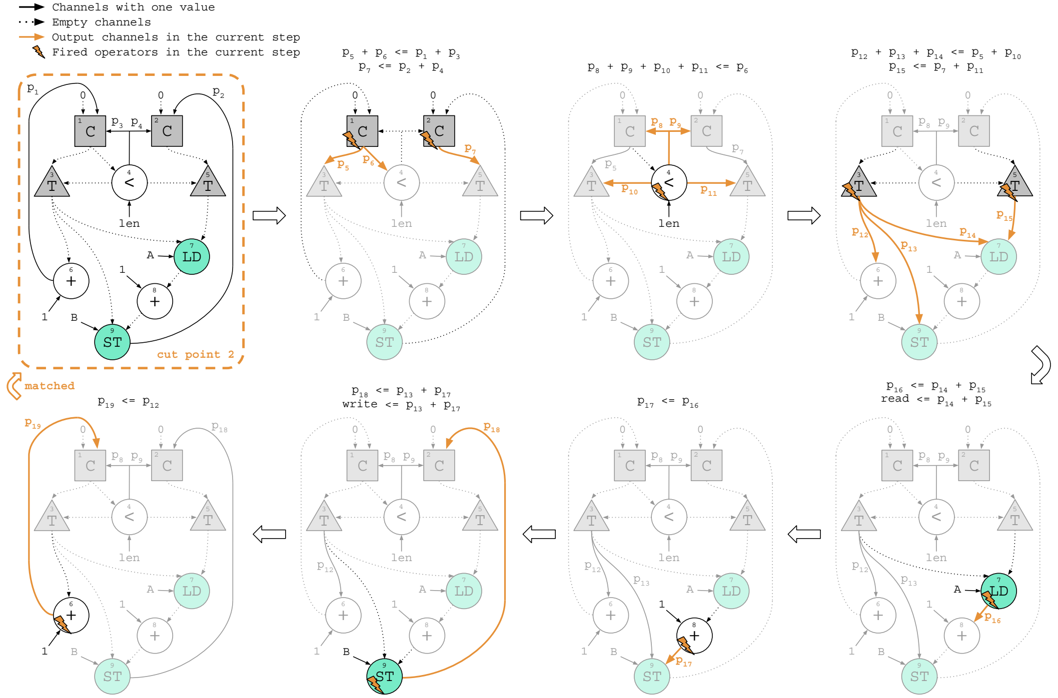


Fig. 3. This plot shows a trace of execution from dataflow cut point 2 (upper left corner). The operators fired in this trace are 1, 2, 4, 3, 5, 7, 8, 9, 6 (which are inferred from the LLVM trace from cut point 2 to cut point 2). At the end of the trace, the configuration matches cut point 2 again. In each configuration, we mark the fired operators with orange lightning symbols. The emptiness of each channel after firing the operator(s) is represented by dashed vs solid lines. Before the firing of an operator, the input channels of the operator should be non-empty and marked with a solid line. After firing, the input channels are emptied, and the modified output channels will become non-empty and marked with a solid, orange line. The symbolic values in each channel are omitted, but their attached permission variables (starting with p) are marked. At the top of each configuration, we also indicate the permission constraints added at each step.

Index	LLVM Program Point	Dataflow Configuration	Correspondence
1	`%entry`	Initial config	$\mathsf{Mem}_{\mathrm{df}} = \mathsf{Mem}_{\mathrm{df}}$
2	`%header` (from `%body`)	First configuration in Figure 3	$\mathsf{Mem}_{\mathrm{df}} = \mathsf{Mem}_{\mathrm{df}} \wedge \texttt{\%i} = \texttt{i}$
$\perp$	`%end`	Final config	$\mathsf{Mem}_{\mathrm{df}} = \mathsf{Mem}_{\mathrm{df}}$

Fig. 4. Cut points for the example in Figure 2. For the second LLVM cut point, FlowCert places it at the program point at the back edge from `%body` to `%header`. The correspondence equality $\mathsf{Mem}_{\mathrm{df}} = \mathsf{Mem}_{\mathrm{df}}$ means that the memory state should be the same at each cut point. The equality `%i` = `i` means that the LLVM variable `%i` at cut point 2 should be equal to the dataflow variable `i` referring to the value in the channel from operator 6 to 1 in Figure 3. We implicitly assume that all function parameters are equal (i.e., `%A` = `A` ∧ `%B` = `B` ∧ `%len` = `len`).

branches: reaching `%header` after one loop iteration (cut point 2), or failing the loop condition and reaching `%end` (cut point ⊥). For LLVM cut point 2, we also have two branches: one reaching cut point 2 again after a loop iteration, the other failing the loop condition and reaching cut point ⊥.

For the dataflow side, we have a correspondence between a subset of LLVM instructions and a subset of dataflow operators, which is automatically generated by the dataflow compiler. We use this mapping to infer the order in which we fire the dataflow operators, i.e., the canonical schedule. For example, from LLVM cut point 2, there are two branches reaching cut point 2 and cut point $\bot$ respectively. These two branches have two traces of LLVM instructions (excluding `br`):

- For the first branch to cut point 2: lines 5, 6, 9 - 14;
- For the second branch to cut point $\bot$: lines 5, 6.

For the first branch, the LLVM instructions map to dataflow operators 1, 2, 4, 7, 8, 9, 6 (excluding ones that do not have a corresponding operator; operator 2 corresponds to a `phi` instruction implicitly inserted during compilation for memory ordering). LLVM instructions in the second branch map to dataflow operators 1, 2, and 4. Therefore, from dataflow cut point 2, we execute these two traces of dataflow operators (and also any steer operator that can be executed), which gives us two dataflow configurations, similar to the LLVM side. Figure 3 shows the trace of dataflow execution of the first branch from cut point 2 to itself.

At each step of the execution, we try to match the pair of LLVM and dataflow configurations to a cut point. If there is a match, we check the validity of the correspondence equations at the target cut points (given the assumptions made in the source cut point). If starting from all cut points, the symbolic execution eventually reaches another cut point with the correspondence equations satisfied, we soundly conclude that we have obtained a simulation relation between the LLVM program and the dataflow circuit on the canonical schedule.

Determinacy Check. Once we have established a simulation relation between the LLVM program and a canonical schedule of the dataflow circuit, we perform a *determinacy check* to ensure that the dataflow circuit is free of data races. As a result of race freedom, if the LLVM program terminates, then any other schedule will terminate in the same final state as the canonical schedule.

We achieve this by using *affine permission tokens*. For this example, let us consider a simplified set of permission tokens $\{0, \texttt{read}_1, \texttt{read}_2, \texttt{write}\}$, where 0 means no permission to read or write, and `read`/`write` means read/write permission to the entire memory, respectively. Denote $p_1 + p_2$ as the disjoint sum of two permission tokens p_1 and p_2. The disjoint sum is a partial function satisfying $p + 0 = 0 + p = p$, $\texttt{read}_1 + \texttt{read}_2 = \texttt{write}$. However, $\texttt{write} + \texttt{write}$ is undefined. Moreover, we partially order these permission tokens by $0 < \texttt{read}_1, \texttt{read}_2 < \texttt{write}$. We will write `read` in place of $\texttt{read}_1$ or $\texttt{read}_2$ if the subscript is irrelevant.

In general, we allow `write` to split into k tokens of `read`, for a predetermined value of k, and we have one permission token for every memory region, such as the array `A` or `B`. The intuition is that to write to a memory location, an operator needs to have exclusive write ownership of that location and no other operator should have `write` or `read` permissions; while to read a memory location, we allow potentially k parallel reads to happen independently. If all reads are done, and a write needs to occur, we merge all `read` tokens into one `write` token.

These permission tokens are passed between operators via channels. Permission tokens can be dropped, but they cannot be created or duplicated (i.e., they are an affine resource). We attach permissions to values flowing through the dataflow circuit, instead of creating separate channels for communicating permission tokens.

To determine the exact assignment and flow of permission tokens, we first attach a permission variable (representing a permission token that we do not know a priori) to each value in the channels of dataflow configurations in each symbolic branch. Figure 3 shows an example of this attachment of permission variables and permission constraints. At the starting state cut point 2, we attach free permission variables p_1, p_2, p_3, p_4 to values in the configuration. After the first step in which operators 1 and 2 fire, these permission variables are consumed by the two carry

operators and we generate fresh permission variables p_5, p_6, p_7 for the new output values. We require that the permission variables are used in an affine fashion: operators cannot duplicate or generate permission tokens; instead, they have to be obtained from the inputs. Hence, for the first step in Figure 3, we add two constraints $p_5 + p_6 \leq p_1 + p_3$ and $p_7 \leq p_2 + p_4$ to say that the output permissions have to be contained in the input permissions. Furthermore, if the operator is a load or a store, such as in Figure 3 steps 4 and 6, we require that the suitable permission (`read` for load, and `write` for store) is present in the input permissions (e.g., $\texttt{write} \leq p_{13} + p_{17}$ at step 6 in Figure 3).

Finally, when the execution is finished and has either hit a final state or another cut point, we have collected a list of constraints involving a set of permission variables. If the configuration is matched to a cut point, we add additional constraints to make sure that the assignment of tokens at the cut point is consistent. For example, in Figure 3, since the last configuration is matched again to the same cut point 2, we would add these additional constraints: $p_1 = p_{19}$, $p_2 = p_{18}$, $p_3 = p_8$, $p_4 = p_9$. We then solve for the satisfiability of these constraints with respect to the permission algebra $(\{0, \texttt{read}_1, \texttt{read}_2, \texttt{write}\}, +, \leq)$ that we have defined above.

If there is a satisfying assignment of the permission variables with permission tokens, then intuitively, the operators in the dataflow circuit are consistent in memory accesses, and there will not be any data races. In our full paper [22], we show that this implies that any possible schedule will converge to the canonical schedule we use.

3 (Completed) Wavelet: A Verified Pipelining Compiler for Asynchronous Dataflow

Our prior work FlowCert (Section 2) has two main limitations. First, as a translation validation approach, it can only certify individual compilation instances, and the heuristics for discovering the simulation relation may not always succeed—there could be false positives where FlowCert is unable to prove the correctness of compilation. Second, FlowCert uses a simple permission algebra where the permission for an entire array must be transferred as a whole. This precludes an important optimization called *pipelining* [33], where the execution of different iterations of a loop can be interleaved to improve performance and reduce circuit size.

To improve upon these limitations, we present Wavelet [21], a pipelining compiler for asynchronous dataflow, with core passes formally verified in the Lean theorem prover [28]. To modularly verify the determinacy of the compiled dataflow circuit while enabling pipelining, we use a combination of typing and compiler verification techniques.

Our frontend language $\mathbb{L}^*_{let}$ is equipped with a novel capability type system to turn implicit sequential memory dependencies into explicit data dependencies through *affine permission variables*. Our core verified passes then compile an elaborated $\mathbb{L}^*_{let}$ program into dataflow circuits represented in the $\mathbb{L}_{flow}$ calculus, with forward simulation formally verified in Lean.

The crucial determinacy property of the dataflow circuit is modularly propagated from the frontend type system to the backend without modifying the forward simulation proofs at all. We interpret $\mathbb{L}^*_{let}$ and $\mathbb{L}_{flow}$ programs in a common semantic framework, and formulate the soundness guarantee of the type system as label restrictions on the semantics. Such label restrictions are preserved by forward simulations, and therefore satisfied by the compiled dataflow circuit. Finally, we show that any dataflow circuit satisfying the label restrictions must be determinate.

For evaluation, we compare Wavelet with two unverified dataflow compilers in RipTide [10] and LLVM CIRCT [25], and show that Wavelet produces dataflow circuits with comparable quality, while providing formal guarantees of correctness and determinacy.

In the following, we start with an overview of Wavelet's design and proof structure in Section 3.1 and then present evaluation results in Section 3.2. More details about Wavelet can also be found in our upcoming PLDI paper [21], which is a collaboration with Yi Cai, advised by Milijana Surbatovich. My main contribution is the Lean formalization, the design of the overall compilation pipeline,

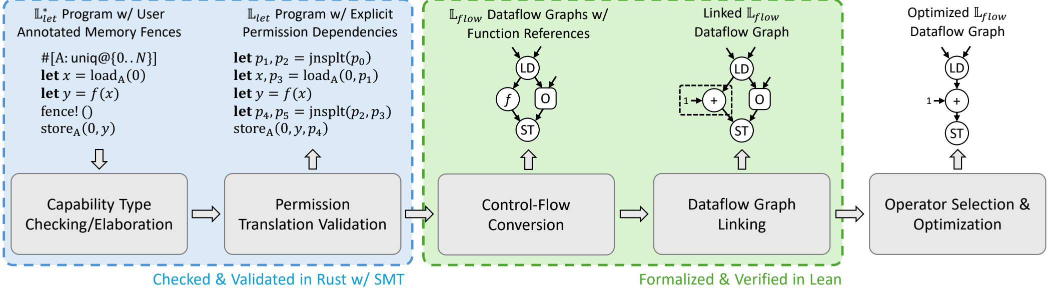


Fig. 5. Compiler pipeline of Wavelet. The compilation transforms the frontend language $\mathbb{L}^*_{let}$ through the elaborated IR $\mathbb{L}_{let}$ to the dataflow calculus $\mathbb{L}_{flow}$.

and the evaluation, and Yi's contribution is the frontend, including the capability type system and permission validation.

3.1 System and Proof Overview

We present an overview of Wavelet's compiler pipeline in Figure 5. At a high level, Wavelet takes in a sequential program decorated with *capability types* for accessing arrays and user-inserted *fences* indicating synchronization points, and translates it into a lower-level dataflow circuit that is guaranteed to be race- and deadlock-free, while still allowing pipelining. Wavelet uses the following five main passes: (1) capability type checking, to ensure that the source program correctly uses fences to mark conflicting memory accesses; (2) elaboration, to lower fences to explicit manipulation of affine memory permissions using the jnsplt operator; (3) control flow conversion, to convert sequential control flow to dataflow; (4) linking, to modularly connect individually compiled functions into a single dataflow circuit; and (5) operator selection and optimization, to rewrite and lower the graph to an optimized, architecture-specific representation. The soundness of the frontend passes is certified with translation validation implemented in Rust, while the core control flow conversion and linking passes are formally verified in Lean. The last operator selection/optimization pass is currently unverified.

The Frontend: Capability Type Checking and Elaboration. A programmer using Wavelet writes their application in a sequential language, $\mathbb{L}^*_{let}$, which is implemented as a DSL embedded in Rust. The goal of the frontend design is to give the core compiler hooks for safely determining when operations can proceed in parallel and when they must be sequentialized to remain sound, without the programmer reasoning about details of dataflow execution. To that end, we design: (1) a system of *capability types* and *fences* that ensure a program only type checks if potentially racing accesses are in separate fenced regions, and (2) an elaboration pass that lowers fences into explicit permission tokens that the backend uses to provably preserve the typing guarantee.

The main idea of the capability types is that the user can type an array region as uniq for exclusive write access, or shrd for shared read accesses. Crucially, these types are *index-dependent*—the region R need not be the entire array but instead a fine-grained slice, allowing the same array to be accessed in both exclusive and shared patterns. This allows the compiler to identify operations on disjoint array indices and prove that they can execute in parallel, enabling safe pipelining, a key goal of Wavelet. When the array access patterns are inherently sequential (e.g., sequential reads/writes to the same address), Wavelet asks the programmer to insert *fences* to express ordering constraints explicitly. Once a program decorated with capabilities and fences type-checks, it is guaranteed to execute without data races, assuming the fences are respected.

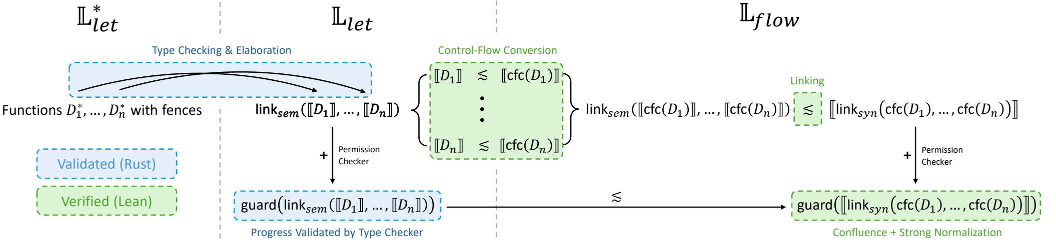


Fig. 6. Proof structure overview of Wavelet. $\lesssim$ denotes simulation; cfc is control flow conversion; link_{sem} is semantic linking of function definitions to dataflow circuits; link_{syn} is syntactic linking of compiled dataflow circuits; guard imposes permission checking on $\mathbb{L}_{let}$ and $\mathbb{L}_{flow}$ semantics.

Since fences are not directly supported by the backend dataflow architecture, we lower $\mathbb{L}^*_{let}$ programs with fences to an intermediate representation $\mathbb{L}_{let}$, equipped with the explicit synchronization operator jnsplt. Wavelet elaborates the abstract fences in $\mathbb{L}^*_{let}$ programs into manipulation of *ghost permission variables* in $\mathbb{L}_{let}$, reifying the fence's ordering constraints as explicit data dependencies between ghost variables, similar to control signals in the underlying architecture. To ensure that the lowering process is sound, we adopt a *translation validation* approach, where we validate that permission tokens present in the lowered $\mathbb{L}_{let}$ program reflect the original typing constraints and satisfy the soundness property assumed by Wavelet's backend.

The Backend: Verified Control Flow Conversion and Linking. The elaborated $\mathbb{L}_{let}$ program still uses sequential control flow. As the next step, Wavelet performs two formally verified core passes (the right half of Figure 5) to compile an $\mathbb{L}_{let}$ program to the dataflow calculus $\mathbb{L}_{flow}$.

The first is *control flow conversion*, which individually compiles each function in the elaborated $\mathbb{L}_{let}$ program to an $\mathbb{L}_{flow}$ process by translating sequential control-flow constructs (e.g., branching and tail-recursion) into pure dataflow operators (e.g., steer and carry in Figure 1). While this pass is standard in dataflow compilers dating back to TTDA [3], it is particularly error-prone and challenging to verify. The core difficulty stems from the mismatched semantics of variables in a sequential program and channels in a dataflow circuit. Unlike in a sequential program, when a "variable" in dataflow is used in, e.g., both branches of a conditional, each use requires a *fresh* communication channel to receive the value. Thus, to formally relate variables to channels, our simulation proofs need to carefully track unused variables and the current path condition. Additionally, for a dataflow circuit to be safely invoked multiple times, Wavelet ensures that all unused values are properly consumed at the end of execution to avoid deadlocks.

The second verified core pass is *linking*, which connects individually compiled dataflow circuits into a single graph representing the entire program. Function calls in dataflow pose unique challenges for linking. Unlike sequential programs with an explicit call stack, dataflow circuits distribute local variables across channel buffers that may hold values from multiple concurrent function invocations. To tackle these challenges, we place static restrictions on $\mathbb{L}^*_{let}$ to facilitate linking, and use a modular linking semantics inspired by interaction semantics [34].

To aid formal verification, we implement these core passes in Lean. This implementation is parametric in an abstract set of operators, making our compiler framework applicable to dataflow architectures with different instruction sets. We then prove the following final correctness results: (a) *forward simulation*: the output dataflow circuit has at least one deadlock-free schedule with the same behavior as the input sequential program; and (b) *determinacy*: given a terminating $\mathbb{L}_{let}$

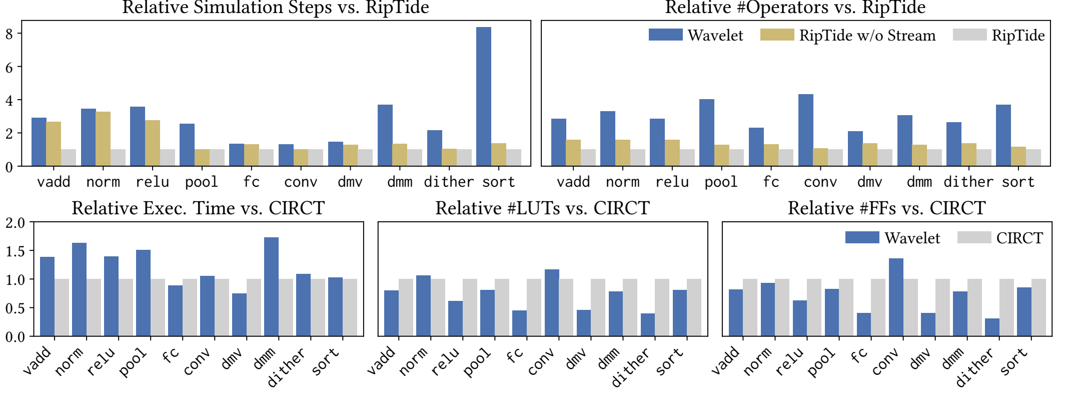


Fig. 7. Compilation quality: the first row compares relative simulation performance and graph size of Wavelet, RipTide without streamification, and RipTide; the second row compares relative execution time and resource usage of Wavelet and CIRCT's dynamic HLS pipeline. The execution time is computed from simulated cycle counts from Verilator [35] and estimated clock periods from nextpnr [36].

program that satisfies our type system's soundness property, the output $\mathbb{L}_{flow}$ process is confluent and strongly normalizing, producing the same result regardless of execution schedule.

Figure 6 gives a proof structure overview of Wavelet, showing how different guarantees of each component connect to establish our final correctness results. In particular, we designed the mechanized portion of the proofs to be modular in a few key ways to improve maintainability. First, we separate the simulation proofs for control flow conversion and linking, simplifying both and enabling modular replacement and verification of functions with optimized dataflow circuits. We then propagate our type system's soundness property as *label restrictions* on the labeled transition systems of $\mathbb{L}_{let}$ and $\mathbb{L}_{flow}$, and prove determinacy independently from the core simulation proofs.

Final Operator Selection and Optimizations. At this point, we have constructed a dataflow circuit in $\mathbb{L}_{flow}$ that simulates the input $\mathbb{L}_{let}$ program and is determinate. To interface with actual backend hardware (e.g., RDA [23] or FPGAs), we lower certain pseudo-operators to architecture-specific operators (instruction selection), and perform unverified optimizations to improve the quality of the dataflow circuit. We define these transformations as graph rewrites in $\mathbb{L}_{flow}$. Combined with the memory ordering strategy in the type checker/elaborator and the core control-flow passes of the compiler, Section 3.2 shows that the produced dataflow circuits have comparable quality to unverified dataflow compilers in RipTide [10] and LLVM CIRCT [25].

3.2 Evaluation

The dataflow circuits produced by Wavelet are intended to be either configured directly on an RDA (e.g., RipTide [10]), or further lowered to RTL and deployed on FPGAs (i.e., dynamic high-level synthesis [15, 25]). Therefore, the performance and resource usage of the compiled dataflow circuit are two crucial metrics of compilation quality.

In Figure 7, we compare Wavelet with two unverified dataflow compilers in RipTide and LLVM CIRCT [25] on 10 benchmark programs from RipTide, and we discuss the results in detail below.

Comparison with RipTide. The RipTide project [10] features an LLVM-based dataflow compiler targeting the RipTide RDA. The output dataflow circuits of Wavelet and the RipTide compiler are at a similar level of abstraction, so we directly compare their simulation performance and graph

sizes in the first row of Figure 7. Simulators of both Wavelet and RipTide eagerly fire all operators that are ready in each simulation step. While this is not cycle-accurate, the relative performance of Wavelet and RipTide dataflow circuits should still be informative.

Compared to RipTide, Wavelet's dataflow circuits are 2.62× slower and have 3.04× more operators in geometric mean. The main downside of Wavelet is that it produces significantly more control-flow operators (e.g., carry, steer, etc.). In the benchmarks, Wavelet-compiled graphs have 78% control-flow operators on average, compared to 51% in RipTide's graphs. We attribute this issue to the memory synchronization strategy in Wavelet. Wavelet passes permission tokens *separately* from concrete values, extending all operators with additional permission token inputs/outputs. While this makes the type system more principled—permission variables are affine and do not interfere with ordinary variables—it introduces many permission variables and larger graphs.

On the other hand, RipTide *mixes* synchronization signals with values and optimizes away unnecessary memory ordering when data dependencies already enforce them. However, this analysis is error-prone and harder to verify compositionally. In fact, in the `sort` benchmark, we find that RipTide is missing a memory ordering, causing a data race and incorrect simulation results.

Finally, pipelining in Wavelet does improve performance even with larger graphs. For example, in benchmarks `vadd`, `norm`, and `fc`, even though Wavelet produced graphs with 2× more operators than RipTide (without streamification), the final performance is still similar due to pipelining.

Comparison with CIRCT. CIRCT [25] is an MLIR-based compiler framework aiming to bring modularity to hardware design and high-level synthesis. We compare Wavelet against a dynamic HLS pipeline in CIRCT, which compiles MLIR's structured control-flow dialect (`scf` [27]) to dataflow circuits (CIRCT's `handshake` dialect [26]). For Wavelet, we implement an unverified lowering pass from $\mathbb{L}_{flow}$ to the `handshake` dialect, where most operators in Wavelet have a direct correspondence.

We then lower the produced `handshake` dataflow circuits to SystemVerilog RTL designs (via CIRCT), and use Verilator [35] for simulation. To estimate cycle periods and resource utilization, we synthesize the designs targeting the Lattice ECP5 FPGA [32] using Yosys [37] and nextpnr [36].

The resulting comparison is shown in the second row of Figure 7, including relative execution time and resource utilization of the generated designs from Wavelet and CIRCT. The results show that Wavelet's dataflow circuits, when used in HLS, are comparable with CIRCT's compilation results in both performance and area, with some benchmarks even outperforming CIRCT's output. On average, Wavelet's results are 1.2× slower in execution time, and have 0.69× fewer LUTs and 0.67× fewer FFs compared to CIRCT's output.

During evaluation, we discovered two issues [19, 20] in CIRCT, which further highlights the challenge of building a correct dataflow compiler.

4 (Proposed) Verifying Tagged Dataflow Compilation

The compiler verification techniques in Sections 2 and 3 have exclusively focused on *ordered dataflow*, in which channels have FIFO semantics and the dataflow circuit has a static topology. A complementary and equally influential model is *tagged dataflow*, adopted by architectures such as TTDA [3], Monsoon [30], and more recently, Tyr [1].

Tagged dataflow differs from ordered dataflow in two key aspects. First, every value carries a *tag* that distinguishes parallel invocations of a particular region of the dataflow circuit (e.g., different loop iterations, function calls, or "threads"). Operators can therefore fire as soon as matching tags are available on their inputs, regardless of arrival order. Second, the graph topology is dynamic: operators may determine their consumers at runtime or non-deterministically choose among inputs.

Together, these relaxations trade the simplicity of ordered dataflow for greater parallelism and expressiveness. Tags remove the requirement that operators consume inputs in order; parallelism

in ordered dataflow is typically limited to prevent deadlocks (e.g., values from an earlier iteration block those of later iterations), but tags lift this restriction by letting operators prioritize whichever invocation is ready first. Dynamic topology, in turn, enables more expressive control flow: Id [29], a language targeting tagged dataflow, supports higher-order functions, whereas most ordered dataflow architectures [9, 10, 31] are restricted to first-order programs.

In this proposed work, we plan to extend the Wavelet framework to the more expressive model of tagged dataflow, with two goals:

(1) Formally verify tagged dataflow compilation targeting Tyr [1] and TTDA [3].
(2) Unify the IRs and compiler proofs for ordered and tagged dataflow.

For goal (1), we plan to follow the same proof strategy as Wavelet (Section 3): first establish forward simulation from the source language to the compiled tagged dataflow, then verify determinacy separately. The main challenge for forward simulation in tagged dataflow is *tag management*—the runtime mechanism that allocates, exchanges, and frees tags. To keep the simulation proof modular, we plan to base our tagged dataflow model on the Tyr architecture [1], whose tag management system is both local and tunable: each region of the dataflow circuit operates in a disjoint, independently sized tag space.

Determinacy poses a different challenge. In the ordered dataflow model of Wavelet (Section 3), shared memory is the only source of non-determinism; without it, the model falls within Kahn's process networks [16] and is always determinate. Tagged dataflow, however, introduces operators that dynamically determine their consumers or non-deterministically choose among inputs, creating additional sources of non-determinism even in the absence of shared memory. To tame this dataflow-level non-determinism, we plan to identify invariants on the dataflow state that preserve determinacy. Similar invariants have been studied in Cilan [17] for ordered dataflow with non-deterministic merges; we plan to extend that approach to the tagged setting.

For goal (2), we observe that despite their semantic differences, ordered and tagged dataflow graphs share significant structural similarity. UDIR [2] exploits this observation to unify compiler infrastructure across both models. Inspired by UDIR, we plan to explore whether the compiler proofs can similarly be unified within the formal setting of Wavelet.

5 Timeline

The following is an estimated timeline for completing my thesis.

- Thesis Proposal: Apr 30, 2026
- Tagged Dataflow Verification: Summer and Fall 2026
- Thesis Writing: Spring 2027
- Thesis Defense: End of Spring 2027

References

[1] Nikhil Agarwal, Mitchell Fream, Souradip Ghosh, Brian C. Schwedock, and Nathan Beckmann. 2024. The TYR Dataflow Architecture: Improving Locality by Taming Parallelism. In *Proceedings of the 2024 57th IEEE/ACM International Symposium on Microarchitecture* (Austin, TX, USA) *(MICRO '24)*. IEEE Press, 1184–1200. doi:10.1109/MICRO61859.2024.00089

[2] Nikhil Agarwal, Mitchell Fream, Souradip Ghosh, Brian C. Schwedock, and Nathan Beckmann. 2024. UDIR: Towards a Unified Compiler Framework for Reconfigurable Dataflow Architectures. *IEEE Computer Architecture Letters* 23, 1 (2024), 99–103. doi:10.1109/LCA.2023.3342130

[3] Arvind and R.S. Nikhil. 1990. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Trans. Comput.* 39, 3 (1990), 300–318. doi:10.1109/12.48862

[4] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3, Article 50 (may 2018), 39 pages. doi:10.1145/3182657

[5] John Boyland. 2013. *Fractional Permissions.* Springer Berlin Heidelberg, Berlin, Heidelberg, 270–288. doi:10.1007/978-3-642-36946-9_10

[6] Jinyi Deng, Xinru Tang, Jiahao Zhang, Yuxuan Li, Linyun Zhang, Boxiao Han, Hongjun He, Fengbin Tu, Leibo Liu, Shaojun Wei, Yang Hu, and Shouyi Yin. 2023. Towards Efficient Control Flow Handling in Spatial Architecture via Architecting the Control Flow Plane. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture* (Toronto, ON, Canada) *(MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 1395–1408. doi:10.1145/3613424.3614246

[7] 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

[8] Souradip Ghosh, Yufei Shi, Brandon Lucia, and Nathan Beckmann. 2025. Ripple: Asynchronous Programming for Spatial Dataflow Architectures. *Proc. ACM Program. Lang.* 9, PLDI, Article 157 (June 2025), 28 pages. doi:10.1145/3729256

[9] Graham Gobieski, Ahmet Oguz Atli, Kenneth Mai, Brandon Lucia, and Nathan Beckmann. 2021. Snafu: an ultra-low-power, energy-minimal CGRA-generation framework and architecture. In *Proceedings of the 48th Annual International Symposium on Computer Architecture* (Virtual Event, Spain) *(ISCA '21)*. IEEE Press, New York, NY, USA, 1027–1040. doi:10.1109/ISCA52012.2021.00084

[10] Graham Gobieski, Souradip Ghosh, Marijn Heule, Todd Mowry, Tony Nowatzki, Nathan Beckmann, and Brandon Lucia. 2023. RipTide: A Programmable, Energy-Minimal Dataflow Compiler and Architecture. In *Proceedings of the 55th Annual IEEE/ACM International Symposium on Microarchitecture* (Chicago, Illinois, USA) *(MICRO '22)*. IEEE Press, New York, NY, USA, 546–564. doi:10.1109/MICRO56248.2022.00046

[11] Google. 2025. XLS: Accelerated HW Synthesis. https://github.com/google/xls.

[12] J.R. Gurd. 1985. The Manchester dataflow machine. *Computer Physics Communications* 37, 1 (1985), 49–62. doi:10.1016/0010-4655(85)90135-3

[13] 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* (Pittsburgh, PA, USA) *(ASPLOS 2026)*. Association for Computing Machinery, New York, NY, USA.

[14] Olivia Hsu, Maxwell Strange, Ritvik Sharma, Jaeyeon Won, Kunle Olukotun, Joel S. Emer, Mark A. Horowitz, and Fredrik Kjølstad. 2023. The Sparse Abstract Machine. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 710–726. doi:10.1145/3582016.3582051

[15] 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

[16] Gilles Kahn. 1974. The semantics of a simple language for parallel programming. *Information processing* 74, 471-475 (1974), 15–28.

[17] 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

[18] Edward A. Lee and Eleftherios Matsikoudis. 2009. *The semantics of dataflow with firing.* Cambridge University Press, Cambridge, UK, 71–94.

[19] Zhengyao Lin. 2026. CIRCT Issue 9807. https://github.com/llvm/circt/issues/9807.

[20] Zhengyao Lin. 2026. CIRCT PR 9587. https://github.com/llvm/circt/pull/9587.

[21] Zhengyao Lin, Yi Cai, and Milijana Surbatovich. 2026. Let it Flow: A Formally Verified Compilation Framework for Asynchronous Dataflow. *Proc. ACM Program. Lang.* PLDI (June 2026).

[22] Zhengyao Lin, Joshua Gancher, and Bryan Parno. 2024. FlowCert: Translation Validation for Asynchronous Dataflow via Dynamic Fractional Permissions. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 289 (Oct. 2024), 28 pages. doi:10.1145/3689729

[23] Leibo Liu, Jianfeng Zhu, Zhaoshi Li, Yanan Lu, Yangdong Deng, Jie Han, Shouyi Yin, and Shaojun Wei. 2019. A Survey of Coarse-Grained Reconfigurable Architecture and Design: Taxonomy, Challenges, and Applications. *ACM Comput. Surv.* 52, 6, Article 118 (Oct 2019), 39 pages. doi:10.1145/3357375

[24] LLVM. 2024. LLVM Language Reference Manual. https://llvm.org/docs/LangRef.html.

[25] LLVM. 2025. CIRCT: Circuit IR Compilers and Tools. https://circt.llvm.org/.

[26] LLVM. 2026. CIRCT: 'handshake' dialect. https://circt.llvm.org/docs/Dialects/Handshake/.

[27] LLVM. 2026. MLIR: 'scf' dialect. https://mlir.llvm.org/docs/Dialects/SCFDialect/.

[28] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings.* Springer-Verlag, Berlin, Heidelberg, 625–635. doi:10.1007/978-3-030-79876-5_37

[29] Rishiyur S. Nikhil Arvind. 1992. Id: a language with implicit parallelism. In *A Comparative Study of Parallel Programming Languages*, John T. FEO (Ed.). North-Holland, Amsterdam, 169–215. doi:10.1016/B978-0-444-88135-9.50010-3

[30] Gregory M. Papadopoulos and David E. Culler. 1990. Monsoon: an explicit token-store architecture. In *Proceedings of the 17th Annual International Symposium on Computer Architecture* (Seattle, Washington, USA) *(ISCA '90)*. Association for Computing Machinery, New York, NY, USA, 82–91. doi:10.1145/325164.325117

[31] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2017. Plasticine: A Reconfigurable Architecture For Parallel Patterns. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. Association for Computing Machinery, New York, NY, USA, 389–402. doi:10.1145/3079856.3080256

[32] Lattice Semiconductor. 2026. Lattice ECP5 FPGA family. https://www.latticesemi.com/en/Products/FPGAandCPLD/ECP5.

[33] Nathan Serafin, Souradip Ghosh, Harsh Desai, Nathan Beckmann, and Brandon Lucia. 2023. Pipestitch: An energy-minimal dataflow architecture with lightweight threads. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture* (Toronto, ON, Canada) *(MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 1409–1422. doi:10.1145/3613424.3614283

[34] Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. 2015. Compositional CompCert. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 275–287. doi:10.1145/2676726.2676985

[35] Verilator. 2026. Verilator open-source SystemVerilog simulator and lint system. https://github.com/verilator/verilator.

[36] Yosys. 2026. nextpnr portable FPGA place and route tool. https://github.com/YosysHQ/nextpnr.

[37] Yosys. 2026. Yosys Open SYnthesis Suite. https://github.com/YosysHQ/yosys.